SESSION

CASE STUDIES

Chair(s)

TBA
Application of Open Hypermedia to Military Software

Kenneth M. Anderson
Department of Computer Science, University of Colorado, Boulder, CO, USA

Abstract - Maintaining large legacy avionics software systems is a complex task. Retaining understanding of a complex system given changes to the system and to its support organization is critical. A significant capability when supporting understanding of these software systems is managing the complex (typically implicit) relationships that exist between their artifacts. In this paper, we examine the use of an approach called open hypermedia to address large-scale relationship management issues. Two key technical features characterize open hypermedia: (1) externally managed relationships and (2) integration of hypermedia services into third-party applications. We ground our insights with our experience in supporting change-impact assessment tasks on a large avionics subsystem. We compare open hypermedia with other approaches to relationship management and argue that it is superior along several dimensions. We conclude with a critical assessment of the approach and suggest improvements for future generations of open hypermedia technology.

Keywords: open hypermedia, link types, military software

1 Introduction

A complex military software system is defined, developed, and maintained using a large collection of interrelated software work-products. These artifacts include planning documents, requirements specifications, design descriptions, test plans, source code and data files. Each artifact may be developed or revised by different organizations or personnel, using different tools, over the lifetime of a major software system. Each artifact may describe the software system from a different perspective. The relationships between these perspectives are significant but are not always represented explicitly. These relationships represent a critical component of the knowledge about the system being developed by the avionics organization.

The explicit representation of relationships may change based on the tools used to create or use the software artifacts. For example cross references between document segments, diagrams, and figures are maintained by many document publishing tools (e.g. FrameMaker). Code dependencies are represented between modules by integrated development environments. Individual structures may be defined to specify the relationships between data components (as in a relational database schema). These relationship representations are useful in a variety of contexts.

However scalability issues must be addressed when applying tool-unique relationship management mechanisms within large software development organizations. To be effective, an approach to relationship management must cover a wide range of artifacts, efficiently handling relationships between internal sections of a given artifact and relationships across a set of artifacts. The process for creating artifacts implies that each succeeding artifact refines a system concept or implements that concept at some additional level of detail. A significant process relationship ensures the traceability of requirements definition to requirements implementation and validation. Other relationships capture interface specification and change management issues. Each of these relationships must be considered when evaluating the impact of changes to a major avionics system (see Fig. 1).

1.1 Relationship Types

The extent of relationships between artifacts can be characterized. Within an artifact, structural relationships define the organization, sequence, or sub-component structure of an artifact. These individual components (e.g., paragraphs) may be cross-referenced in other parts of the structure (to reduce document size and redundancy). Component mappings or tables may be created to capture some of the critical or contractually-required relationship visibility defined in Fig. 1. Finally, textual references may indicate a relationship between domain specific concepts defined or discussed in different sections of an artifact set.

The extent of this relationship network exceeds the capabilities of most artifact development and maintenance tools. Maintaining relationships represented using different tool-specific mechanisms can be expensive and error prone. Incrementally adding ubiquitous relationship management capabilities to these local mechanisms is needed. From such an approach, an organization requires effective support for task driven navigation and information organization. These requirements indicate that open hypermedia techniques may be a useful technology for supporting this domain. However, other factors must be considered. For instance, the choice of a relationship management technology within a legacy avionics development organization may be driven more by organizational/infrastructure issues than by technical issues.

1.2 Organizational Characteristics

Over the extended life of an avionics software development project, historical events leave a support organization with a heterogeneous environment of computing resources, network infrastructures, and documentation systems. Over time, different development groups may have used different tools to develop their work-products. Software tool licenses and tailoring of these tools to each group’s processes may have required an investment by the organization that impedes the transition to common tools and environments.
As an avionics system progresses to a maintenance phase, opportunities exist for injecting new software tools. However, the extent of this retooling is limited by several factors, e.g., limited maintenance personnel, tightly constrained schedules and budget, and the need for baseline management. As maintenance begins, the design staff is quickly reduced; retraining of the remaining staff to use new tools is risky. Delivery schedules are fixed or may be compressed. Experimentation with new tools cannot impede this schedule. The cost of new tools, their licenses, infrastructure, and training must also be considered within limited maintenance budgets. In most cases, the application of new tools requires the reformatting of existing documentation to enable the benefits of these new support tools.

Within a maintenance organization, changes to the baseline must be explicitly controlled; documentation changes are tracked at the page level. Automated conversion from one document format to a new format required by a new tool has not been particularly successful at maintaining the exact representation of the original document. Automated conversion of tables, diagrams, and cross-references has induced many errors that directly impact baseline traceability. The effort required to correct these errors has not proven to be cost effective for major legacy software projects.

1.3 **Open Hypermedia Approach**

Several choices exist for addressing the relationship management problem. In this paper, we argue for the use of open hypermedia technology. Open hypermedia systems combine many of the benefits of database technology (access control, query, versioning, scalability) and the World Wide Web (openness, distribution, scalability) while addressing the infrastructure and organizational issues identified above. The open hypermedia approach is characterized by two technical features: externally-managed relationships, allowing for a separation of concerns between relations and the information being related, and support for third-party applications, allowing for the use of familiar tools. Open hypermedia provides environment-wide hypermedia services (such as navigation, guided tours, and arbitrary link granularity) over a heterogeneous set of document types and legacy applications.

The application of open hypermedia technology within the maintenance environment of a legacy avionics organization must provide a solution within the constraints mentioned above. The benefit of improved access to related information must be provided using existing documentation formats and systems. The extraction or recovery of relationships between work-products must be automated and performed incrementally, external to the original artifacts. The remainder of this paper discusses a project in which open hypermedia techniques were used to support change-assessment tasks over a large legacy avionics subsystem. The Chimera open hypermedia system [1–2] provided the supporting infrastructure for this pilot study.

The rest of the paper is organized as follows. We describe the characteristics of the software system used in the pilot study, and describe how open hypermedia was applied to manage its relationships. Next, we reflect on the lessons learned and critically evaluate the open hypermedia technology employed. We then discuss our future research plans and consider alternative techniques for relationship management before presenting our conclusions.
2 Pilot Study Description

In this section, we characterize the nature of the software system used as a basis for our pilot project. A major focus of this characterization is to identify the types of artifacts and relationships contained in this software system. Our pilot study was executed in the context of a major avionics software program. This program involves millions of lines of avionics software and maintains thousands of pages of documentation all of which may be revised or reviewed for each incremental upgrade to the avionics system.

2.1 Pilot Study Artifacts

Table 1 shows the size (in pages) of the total number of artifacts created for the avionics system that is the target of our pilot study as well as the size of the subset of artifacts actually used in the study. The artifacts used in the study were related to two subsystems of the overall avionics system.

Table 1. Artifact Size for Avionics System

<table>
<thead>
<tr>
<th>Components</th>
<th>Used in Study</th>
<th>Total Artifacts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>1,442</td>
<td>11,583</td>
</tr>
<tr>
<td>Design</td>
<td>185</td>
<td>3,471</td>
</tr>
<tr>
<td>Test</td>
<td>101</td>
<td>9,385</td>
</tr>
<tr>
<td>Interface Specs.</td>
<td>5,117</td>
<td>12,064</td>
</tr>
<tr>
<td>Product Specs.</td>
<td>0</td>
<td>60</td>
</tr>
<tr>
<td>Source Code</td>
<td>651 (files)</td>
<td>Not Disclosed</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>7,496</strong></td>
<td><strong>36,563</strong></td>
</tr>
</tbody>
</table>

2.2 Pilot Study Relationship Descriptions

As mentioned in Sec. 1, there are four types of relationships that can exist between the artifacts of a software system. Table 2 displays the statistics for these relationship types over the entire pilot study. We now describe each type of relationship in detail.

Table 2. Number of Relationship Instances

<table>
<thead>
<tr>
<th>Relationship Type</th>
<th>Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structural</td>
<td>99,000</td>
</tr>
<tr>
<td>Cross Reference</td>
<td>180,000</td>
</tr>
<tr>
<td>Process</td>
<td>64,000</td>
</tr>
<tr>
<td>Concept Associations</td>
<td>201,000</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>544,000</strong></td>
</tr>
</tbody>
</table>

2.2.1 Structural Relationships

Military documents are formally structured as a sequence of document volumes, sections, and paragraphs. Within this structure, each component of the software system to be built is specified in terms of requirements, design, or validation. The meaning of a textual description is defined by its location (document and paragraph level) within this formal structure. The organization of these descriptions into volumes and sections represents the first category of relationships evaluated in our study. This can be viewed as the creation of a detailed table of contents that spans the entire artifact set.

For our pilot study an exhaustive identification of this link type was not pursued. However, an examination of a subset of the pilot study artifacts produced the data in Table 3.

Table 3. Structural Relationship Data

<table>
<thead>
<tr>
<th>Total Documents</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total TOC Links</td>
<td>10,276</td>
</tr>
<tr>
<td>Total Pages</td>
<td>3,772</td>
</tr>
<tr>
<td>Average TOC Links Per Page</td>
<td>2.72</td>
</tr>
</tbody>
</table>

2.2.2 Cross Reference Relationships

The next category of relationships is driven by consistency requirements within specification artifacts, typical of military or manufacturing domains. In this context, detailed descriptions need to be created in one place and then referenced within other artifacts. Two examples of this type of cross-referencing relationship were reviewed.

The first involved locating explicit cross-references within example documents. These were defined as tables or lists and provided mappings between document sections. Additional references were embedded using tool specific or textual cross-referencing mechanisms.

For the second, an evaluation of the interface specification between avionics subsystems was used. This specification typically defines communications between system components (messages); the sub-structures (words) which comprise these messages; and the format, restrictions, and uses of each field within each word of these messages. These interface specifications are then referenced by requirements or design artifacts, or by source code which implements or conforms to these specifications. The number of these relationships existing in the avionics system we studied is presented in Table 4.

Table 4. Interface Specification Data

<table>
<thead>
<tr>
<th>Data</th>
<th>Complete Database</th>
<th>Study Database</th>
</tr>
</thead>
<tbody>
<tr>
<td>Messages</td>
<td>4,259</td>
<td>676</td>
</tr>
<tr>
<td>Words</td>
<td>7,805</td>
<td>4,441</td>
</tr>
<tr>
<td>Message to Word</td>
<td>58,258</td>
<td>35,332</td>
</tr>
<tr>
<td>Message to</td>
<td>6,520</td>
<td>Not Available</td>
</tr>
<tr>
<td>Message</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pages</td>
<td>12,064</td>
<td>5,117</td>
</tr>
<tr>
<td>Code to Message</td>
<td>Not Available</td>
<td>851</td>
</tr>
<tr>
<td><strong>Links Per Page</strong></td>
<td>5.37</td>
<td>6.9</td>
</tr>
</tbody>
</table>

Based on the results of three studies, a summary estimate for cross-reference relationships is presented in Table 5.

Table 5. Cross Reference Data

<table>
<thead>
<tr>
<th>Artifact Set</th>
<th>References Per Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Support</td>
<td>1.06</td>
</tr>
<tr>
<td>Interface Specification</td>
<td>5.37</td>
</tr>
<tr>
<td>VSD Design Document</td>
<td>8.33</td>
</tr>
<tr>
<td><strong>Average References Per Page</strong></td>
<td>4.92</td>
</tr>
</tbody>
</table>
2.2.3 Process Relationships

Software work-products describe the components of a system being built from different perspectives. Software source code represents one perspective: the implementation. Individual sections of other work-products define for each component what it is required to do, how it will do it, and how it will be tested. This sequence of design refinement represents one critical relationship to be maintained across the various products. The solid arrows in Fig. 1 illustrate the relationships collected during the pilot study and are summarized in Table 6. We discuss the approach for identifying these relationships within the artifact set below.

Table 6. Process Relation Data

<table>
<thead>
<tr>
<th>Relationship Type</th>
<th>References Per Page/File</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements to Design</td>
<td>653/185 = 3.5</td>
</tr>
<tr>
<td>Design to Code</td>
<td>468/185 = 2.5</td>
</tr>
<tr>
<td>Requirements to Formal Test</td>
<td>145/101 = 1.4</td>
</tr>
<tr>
<td>Interface Specification to Code</td>
<td>851/5117 = 0.17</td>
</tr>
</tbody>
</table>

2.2.4 Concept Associations

Typical narrative text contains a significant number of associations between concepts. The identification of these associations and the meaning of each association is typically a natural language understanding problem. The concept being defined by the narrative and the intent of references to other concepts can only be inferred. Without stylized references this problem is even more pronounced, where abbreviations, synonyms, and local contextual references must be resolved before the association can be defined. In typical military documents this type of reference is extremely dense. While immature, textual extraction mechanisms have been applied in this domain as a means to automate the recovery of these implicit relationships. One such result identified 1017 terms from 185 pages of documentation.

2.2.5 Extrapolation to Entire Avionics Program

Based on the analysis of the pilot study artifacts, and some extrapolation to the remaining avionics software artifacts, Table 7 summarizes the relationship characterization for the entire artifact set (refining the data of Table 2). This represents a conservative estimate for the number of unidirectional relationships across such a set of artifacts.

Table 7. Summary Relationship Statistics

<table>
<thead>
<tr>
<th>Relationship Type</th>
<th>Estimate Factors</th>
<th>Total References</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structural</td>
<td>2.72 * 36563</td>
<td>99,451</td>
</tr>
<tr>
<td>Cross Reference</td>
<td>4.92 * 36563</td>
<td>179,890</td>
</tr>
<tr>
<td>Process</td>
<td>3.5 * 11583 + 2.5 *</td>
<td>64,407</td>
</tr>
<tr>
<td></td>
<td>3471 + 1.4 * 9385 + 0.17 * 12064</td>
<td></td>
</tr>
<tr>
<td>Concept Associations</td>
<td>5.5 * 36563</td>
<td>201,097</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>544,845</td>
</tr>
</tbody>
</table>

3 Application of Open Hypermedia

The main goal of the pilot study was to evaluate the suitability of open hypermedia technology in addressing the relationship management problem. We used the Chimera open hypermedia system [1–2] as our relationship management and tool integration infrastructure. Our process included identifying document interactions (relationships), creating text and relationship extraction procedures, instantiating a hypermedia network, and, finally, prototyping an interface for navigating relationships.

3.1 Identify Documented Topics

For each document format within our pilot study, an extraction routine was created to locate the textual sections responsible for defining unique topics within the context of each document. For military documents, fixed paragraph numbering schemes are used to identify these topics. For example paragraphs numbered 3.2 and below (i.e. 3.2.1, 3.2.2) identify “Functional Requirements” within a requirements document. Paragraphs numbered 3.1 and below identify software configuration items within a design document. Additional processing was required to isolate the actual topic (e.g., a required function) from sections describing other issues (the inputs/outputs of a required function). The unique textual string to use as the topic name had to be refined for each document format. Naming conventions or prefixes were used to divide the collection of topics into functional areas.

3.2 Identify Documented Relationships

Extraction routines were created to locate mappings between document sections. In particular, “tables” are included in the documentation to maintain the traceability of requirements to design components and test descriptions. The layout in these tables implies the type and direction of the process relationships. Within these tables, references to related topics were specified using paragraph numbers or acronyms. The ambiguity allowed by these narrative references had to be resolved by the extraction routines.

3.3 Relation Naming Conventions

At the end of topic extraction, a collection of textual descriptions has been identified for each topic, as well as a collection of relations between these topics. During this extraction process a unique name and a type were associated with each topic/relationship. The name string concatenated both a type and a unique string. For topics, the unique string was the paragraph number, component name, or label found in the document. For relationships, the unique string was the name of the source topic for the relationship. For the pilot study, relationships were modeled as one to many, allowing this name to be unique.
3.4 Hypermedia Anchors

Depending on the tool responsible for managing a particular artifact a unique set of parameters are required to identify the textual section defining a topic or a relationship. This textual location identifies a relationship end-point, or anchor, within the hypermedia network. For the pilot study, three document browsing tools were used: Framemaker, XEmacs, and Netscape.

Framemaker defines a unique paragraph identifier for each component of a document. This identifier is not position dependent and can be referenced independent of document modifications or presentation. These references can be created by the tool’s API and used by the tool’s cross-reference mechanisms. Adding references to these identifiers does not modify the document content or presentation.

For the XEmacs tool, textual hotspots are maintained using line and column numbers; these anchors are updated by XEmacs as the document is modified. As in the Framemaker case, XEmacs provides an API through which Chimera can create, remove, and retrieve these parameters for each anchor.

In the current Netscape integration, Chimera monitors web page references sent to Netscape. Other hypermedia systems have extended this integration [3–4]. These references (URLs) identify a complete document, but with the addition of fragment identifiers, targets can be defined inside of a document. These location identifiers are not position independent, however. This approach worked well for source code artifacts, where a fragment identifier was added to each line of code as the code was converted to HTML for display.

3.5 Hypermedia Database

Once the topics, relations, and anchor parameters were all identified, this information had to be loaded into the Chimera database to initialize the hypermedia network. Chimera provides an API for creating and manipulating this data interactively. However, for the pilot study, a different Chimera mechanism was used: the database import and export capability. Chimera defines its database structure in terms of XML files. These files can be used to create a structure for disjoint databases (a website), the objects referenced in the hypermedia network (anchors), and relationship sets (links). As the size of the hypermedia network grew, the ability to merge incrementally defined structures was important. The process of parsing the pilot study’s artifacts and totally rebuilding the Chimera database using incrementally defined sets of database import files allowed for reliable development and maintenance of the hypermedia database as a static operation. As a side effect, this file-based export/import mechanism provided a convenient format for update of Chimera concepts required to track document movement.

3.6 Relationship Navigation

Having created the hypermedia network, the next task was to create an effective user interface for understanding and navigating the relationships between software artifacts. Chimera provides an infrastructure for relationship management and for integrating software tools. It focuses on maintaining relationships and traversing between the end points of these relationships. However, the end user needed an operational model that focused on the domain of the document’s topics (not relation end-points) and intelligent models for selecting related information. A second tool—MediaDoc—provided this capability.

MediaDoc [6] defined a menu of topics, organized by type, and allowed the user to select from these. A specialized layout of related topics was created based on the selected topic using the domain model shown in Fig. 1. In addition to this context-specific information tailoring, MediaDoc also provided mechanisms for adjusting the information presentation based on user-role, user-history, or user-task models. As each topic was selected, MediaDoc passed the anchor reference to Chimera, which initiated a traversal to the topic, using the tool responsible for presenting the associated document format. This division of responsibilities proved to be effective. Chimera provided the underlying infrastructure for efficient relationship management, while MediaDoc focused on intelligently presenting the managed information based on domain knowledge.

4 Lessons Learned

Chimera’s ability to maintain external relationships (without requiring format changes to existing documents) allowed legacy documentation to be enhanced with critical hypermedia relationships and allowed improved access to be provided to task-related information. An approach requiring that anchor and link information be embedded in legacy documentation would be untenable due to the organizational constraints discussed in Sec. 1.

Chimera’s abstractions and approach supported a useful integration of intelligent document navigation tools. For instance, Chimera enabled the integration of the MediaDoc tool via a simple extension of its client architecture. In particular, Chimera provides an extensible mechanism for modifying the behavior of link traversal. This mechanism modified Chimera’s default link traversal algorithm to pass the source anchor of a traversal to MediaDoc. MediaDoc uses its domain knowledge to query Chimera for relationship types relevant to a user’s current task. A user can select important artifacts from the MediaDoc display and Chimera traverses to the selected destinations.

The extraction of relationships can be automated for a significant number of critical associations between document sections, enabling open hypermedia to provide relationship
management capabilities at a scale sufficient to provide tangible benefits over the existing manual process.

The mechanisms for importing the Chimera database structure and contents, directly supports the concept that document baselines can be analyzed off-line in order to generate a hypermedia network for efficient navigation. This approach maps well to the concept of document baselines and check-in processes for modern software development. The application of this approach meant that document loading and anchor loading were critical performance constraints. As such, Chimera provides mechanisms for loading document anchors as collections, allowing large sets of anchors to be retrieved en masse from the hypermedia database.

Chimera provides the mechanism for a hierarchy of relationship sets (hyperwebs). This mechanism was not explored in terms of the overlapping nature of these sets, but the concept of having separate hyperwebs for each system configuration or sub-configuration seems an obvious use of this capability.

5 Future Work

The pilot study met with some degree of success. Chimera’s scalability was sufficient to handle the approximately 500,000 relationships extracted from the artifact set. The open hypermedia approach allowed the explicit representation of these relationships and provided rapid navigational capabilities across the artifacts using applications familiar to the software engineers and leaving the original legacy documents unmodified. However our experience has identified several areas for improvement of Chimera and open hypermedia technology in general.

5.1 Support for Additional Views

The open hypermedia approach directly supports the creation of navigational links. These links are accessed by viewing an artifact, selecting one of its anchors, and traversing along the associated relationships to other artifacts. This “view” is not conducive to establishing in a software engineer’s mind the global structure of the hypermedia relationships. Thus, additional views onto the hypermedia network are required. Example visualizations could employ different techniques and perspectives. For instance, an engineer may want to see all of the relationships created by a particular user or during a particular task. An engineer may also want to see a graph that displays all the destinations reachable from an artifact by two or more traversals. Finally, if the hypermedia network consists of links of explicit types, a useful visualization would be to present the network based on a schema of link types.

5.2 First Class Link/Anchor Types

As such, we have plans in place to improve support for first-class link/anchor types in future versions of Chimera. In particular, this would enable the automated management of standard attributes and constraints on anchors/links. In addition, first-class link/anchor types would allow schema-based query mechanisms to be implemented so that clients can obtain information about the hypermedia network based on the link/anchor types most relevant to their domain. Indeed, we have started to explore these issues in our work on requirements traceability using the TraceM environment [7].

5.3 Relocation and Configuration Issues

One problem encountered during the pilot study was the issue of storing names in the hypermedia network. In particular, names that have a tendency to change, such as absolute paths to documents, required changing information stored in the hypermedia database. This process was supported by Chimera’s XML import and export capabilities. When a path name had to be changed, the hyperweb would be exported to an XML document and another tool was used to seek each instance of the old path name and replace it with the new one. The hyperweb would then be imported, updating Chimera with the new information. A long-term solution to this issue requires a mechanism by which a document is given a logical name that is mapped at run-time to its actual name.

6 Related Work

In Sec. 1, we discussed how relationship management is currently addressed with manual ad hoc techniques. These techniques include cross references, document indexes, and even manually written notes on an outdated paper document. These techniques are prone, of course, to consistency and completeness errors. In this section, we restrict our discussion to other techniques for supporting relationship management. Our goal is to show where these techniques breakdown and how the open hypermedia approach is superior.

6.1 Relational Databases

A traditional approach to managing relationships is relational databases. Relational databases support the scalable and reliable management of information and relationships for a particular application domain. For any particular domain, a schema is developed that specifies how the critical elements of the domain are related to each other. A relational database server then manages the creation of and access to databases that store information in the specified format. Applications that understand the schema can then make use of the information to directly support the identified domain.

It is this focus on domains that represents relational databases greatest weakness with respect to relationship management. The sheer heterogeneity of the artifact set of a typical large-scale software project makes specifying a single schema that captures all of the relevant relationships a difficult task. This is especially true when considering the fact that the set of useful link types for a software project is open.
For a particular task, a developer may need an ad hoc relationship type to rapidly navigate to a part of the information space that was unexpectedly useful in completing a task. After that task is finished, the developer may delete the relationship or formalize it into a distinct link type. Creating and deleting links and link types in an open hypermedia system is an efficient operation, allowing developers to manipulate them without being distracted from the task at hand. With relational databases, changing the underlying schema of the database is considered an expensive operation not undertaken lightly.

Another aspect of the heterogeneous artifact set that causes problems for relational databases is that too much of the data is managed by applications that do not make use of a database, and changing a legacy application to store its contents in a database is not a trivial task. Open hypermedia’s integration approach to legacy applications is again more powerful in addressing relationship management needs.

### 6.2 Configuration Management / Versioning

Two important relationship sets exist in the construction of software systems. The first is the relationships which exist between versions of specifications, documents, or code modules. The second is the set which represents a single version of the software system as a whole. It should be noted that open hypermedia is not the appropriate technology to manage these relationships; that task is best addressed by versioning and configuration management systems.

The interesting intersection with open hypermedia and configuration management lies in managing the evolution of open hypermedia relationships. For instance, when an engineer retrieves a previous configuration of a software system from a configuration management system, the engineer’s open hypermedia system must re-establish the open hypermedia links that were active when this configuration was current. This scenario implies a working relationship between open hypermedia systems and configuration management systems that is currently an open research issue. Some work has been done in hypermedia versioning (see, e.g., [5]) and it can be used as a basis for moving forward on this issue.

### 6.3 World Wide Web

The World Wide Web is the largest distributed hypermedia system in existence today. It owes some degree of its success to its reliance on well-designed platform-neutral standards such as HTTP and HTML. The latter defines the primary document format of the Web as well as its rudimentary hypermedia data model. As a result, in order for information to take advantage of the Web’s hypermedia services, it must be specified in HTML. It is this requirement that prevents the Web from being used as a complete solution to the relationship management problem, as described in the Introduction. The Web is incapable of meeting the requirement that legacy document formats be unmodified when providing hypermedia services.

### 7 Conclusions

Our work has produced two contributions for the software engineering community. First, we have characterized the relationship management problem faced by avionics organizations developing large-scale avionics systems. Second, we have demonstrated how open hypermedia techniques can be used to support change-impact assessment tasks over a large-scale software system. In particular, the Chimera open hypermedia system was used to provide rapid navigation services over the pilot study’s artifact set.

**Acknowledgments.** This paper is dedicated to the memory of Gary Brannum, without whom this work would not have been possible. He will be remembered for his ability to foster software engineering research between academia and industry. This effort sponsored by DARPA and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0021 and DARPA Order No. D895. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, Rome Laboratory or the U.S. Government.

### 8 References


A Study of Communication Management in Chinese IT Projects

Hareton Leung, Michael Deng
Department of Computing, The Hong Kong Polytechnic University

Abstract - With the rapid development of China’s economy, Chinese IT market has expanded in recent years. Many IT projects have been initiated to support business development. However, very few IT projects are successful, achieving their goals. Among the factors contributing to the failure of the IT project, poor communication has often been cited as the root cause. In this study, we investigate the communication management in Chinese IT projects, focusing on the state of the practice. The study result shows that the communication management in Chinese IT project is far from satisfactory despite the availability of rich communication resources. The communication model for many IT projects cannot well facilitate the communication among the stakeholders with many problems such as inefficient reporting system, inappropriate communication channel, unavailability of required information, lack of information feedback, very little information clarification, and disorder information management etc. The formal communication is over emphasized while the informal communication is ignored.

Keywords: Communication management, IT project, Chinese project

1 Introduction

Compared to advanced countries such as USA, UK, and Japan, China introduced project management in late 1980s. But in recent years, project management has undergone development at an incredible speed, and has been widely adopted by many organizations and government agencies. Many project management methodologies such as PMBOK, PRINCE2, CMMI, and MSF have also been adopted.

No matter what project management methodology has been adopted, communication is always a critical factor to success [2]. Communication breakdowns are continuously cited as one of the key reasons that projects fail, which is why communication needs to be addressed as a critical activity and skill for project managers.

A survey by the Computing Technology Industry Association (CompTIA) shows that poor communication is a key reason that most IT projects fail. There are 7 primary causes of IT project failure, and inefficient communication was on that list, with the other six causes also closely related to poor communication [7]. For Chinese projects, it is not surprising to run a project without a communication plan because of lack of experience, influence of traditional culture, and government interference. To improve the chance of project success, it is important to understand the state of the practice of project communication in China.

2 Communication Management

Communication is a bi-directional process where individuals share various types of information by using different mechanisms. In any type of business, effective communications are essential. As a general practice, the goal of communication should be to clarify information to the level of depth required by the receiver by minimizing barriers that might inhibit understanding.

There are several models of communication process. Shannon's model is the first general model of the communication process that could be viewed as the common ground for communication research and management [6]. This model breaks the process of communication down into eight discrete components: information source, message, transmitter, signal, carrier or channel, noise, receiver, and destination.

The interactive model elaborates Shannon's model with the concept of feedback [1]. In this model, destinations provide feedback on the messages they receive, and the information sources can adapt their messages in real time. Feedback is transmitted, received, and potentially disruptable by noise sources.

There are many sources of noise or interference that can enter into the communication process. This can occur even when people know each other very well and should understand the sources of error. In a work setting, it is even more common since interactions involve team members who do not know each other well, and communication is often complicated by the complex and conflicting relationships that may exist. The sources of noise may come from: language: defensiveness, misreading of body language, tone and other non-verbal...
forms of communication, noisy transmission (unreliable messages, inconsistency), receiver distortion, selective hearing, ignoring non-verbal cues, power struggles, self-fulfilling assumptions, perceptual biases, and cultural differences [8].

The organization communication can be divided into formal and informal communication [3]. The formal communication refers to the communication through officially designated channels of message flow between stakeholders. Informal communication means the interactions that do not reflect officially designated channels of communication. There are 3 kinds of formal communication: download communication, upload communication and horizontal communication. Individuals may have various preferences for both communicating with others and interpreting the communications from others.

The interaction of communication between sender and receiver can be classified into 3 types: self-action or one-way communication, interaction or two-way communication, and transaction. There are two modes of interaction. In the push model, the information sender sends the receivers detailed information, whether they want it or not. In the pull model, the information sender sends nothing but the most minimal notification, and receivers ask for details explicitly thereafter.

According to PMBOK from PMI, project communication management is regarded as one of the nine management areas of a project [4]. Project communication management employs the processes required to ensure timely and appropriate generation, collection, distribution, storage, retrieval, and ultimate disposition of project information. The key processes are: communication planning, information distribution, performance reporting, and manage stakeholders. These processes provide the critical links among people and information that are necessary for successful communications.

3 Data Collection and Analysis

The data for our research was collected through two ways: questionnaire and interview. Based on literature review and results from 3 case studies, we designed a questionnaire and conducted interviews with selected respondents. The questionnaire was distributed to the potential respondents who have experience in Chinese IT projects. The questionnaire consists of 70 multiple choice questions, which mainly focus on the common practices of project communication.

More data was collected by interviewing some respondents selected from the first group of respondents, achieving a deeper understanding about the communication management in IT projects. An interview guide composes of 16 open questions. Through the face to face communication with the interviewee, richer information can be obtained for our study.

Data collection took one month. For the potential respondents in the local city of Xian, we visited their organizations and asked their directors to complete the questionnaire. For the potential respondents from other cities, we emailed the questionnaires and asked for reply before a due date.

To ensure a high return rate, we contacted the students taking the Master of Science in Information Systems of the Hong Kong Polytechnic University and our friends, especially our former colleague. We also received assistant from the alumni association. Thus, convenience sampling is used for our study.

About 300 questionnaires were distributed, and 157 questionnaires were returned. 20 respondents were selected for interview, taking into consideration their project types, available resource, and cooperation of respondents.

The analysis was more focused on quantitative measurement of choices and preferences. Raw data was input into SPSS 15.0. Then we use its standard features to calculate frequency and means, and generate summary tables and figures.

3.1 Profile of the Respondents

Among the 157 respondents, female accounted for 17.2% and male 82.8%. Their ages ranged from 20 to 50, and most of them (85%) have received higher education (bachelor degree or higher).

The respondents were from 14 industries: IT (including telecommunication), logistics and transportation, finance, manufacturing, hotel and hospitality, government agency, real estate, tourism, education and research, insurance, health and medical, and energy, oil and mining. The top 4 industries were IT, manufacturing, education and research, and energy, oil and mining, accounting for 19.7%, 19.7%, 15.3%, and 12.1% respectively. The respondents came from companies with less than 50 employees to more than 1000 employees, among which 28% had more than 1000 employees.

3.2 Project Management Environment

The communication resource consists of hardware resource and soft resource. We focused on the soft resource because the hardware resource is quite similar among most projects and is readily available. For the investigation of the soft resource of communication, we investigated the following aspects:

- Organization type
- Establishment of PMO (Project Management Office)
- Support from management
- Support from other functional departments
Organization’s experience on project management, especially in communication management.

From the PMBOK, the organization type of an enterprise greatly affects the implementation of the project, and the distribution of the resource on the project. 73% of the respondents confirmed that their companies were functional organization; 20% of respondents were matrix organizations; and only 7% of the respondents were project based organizations.

To facilitate the implementation of the project, many companies, especially the big functional type enterprises, have set up a PMO, coordinating the management across different departments or different projects at a strategic level. Among the respondents, only 16.56% confirmed that they had a PMO.

The commitment from management board is very important for the implementation of a project, especially for IT projects. Our study showed that almost all caring about the implementation of the project, and only 1% of the respondents claimed that their management board showed little interest of their project. But only 4% of the respondents confirmed that their management board positively participated in the implementation of the project and provided full support, and more than half of the respondents (68%) expressed that their management board just gave support without much participation in the project implementation.

Beside the commitment from the high level management team, the cooperation from other functional departments is also important to the success of a project. Our study indicated that other functional departments (87%) of most companies could provide help, but very few departments (11%) positively participated in project implementation.

For the project management experience, most respondents expressed that their experience were far from satisfactory. 52% respondents claimed that their companies had limited experience, and 45% of respondents said they lacked project management experience, especially in communication management.

3.3 Communication Management

Our investigation into the communication management in IT project consists of 4 parts: communication planning, information distribution and information management, project performance reporting, and communication performance. The respondents were asked to answer the questions based on a typical project they have participated.

The sample IT projects given involved the following types: enterprise information infrastructure platform construction, ERP implementation, CRM implementation, SCM implementation, EIP implementation, EAI, website construction, and OA implementation. The top 2 types were enterprise information infrastructure platform construction and website construction projects, which account for 29.3% and 14.7% of all projects respectively.

The study showed that the investment on IT project is increasing. Among the projects reported by respondents, only 9.6% were less than 0.5 million RMB, and about 12.7% were more than 10 million RMB.

<table>
<thead>
<tr>
<th>Table 1. Project Investment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less 0.5 million RMB</td>
</tr>
<tr>
<td>0.5 -1 million RMB</td>
</tr>
<tr>
<td>1 -5 million RMB</td>
</tr>
<tr>
<td>5 -10 million RMB</td>
</tr>
<tr>
<td>More than 10 million RMB</td>
</tr>
</tbody>
</table>

For the development and implementation of the IT project, more and more companies choose outsourcing. Among the reported projects, 65.6% were outsourced, and 22.9% were based on COTS products and implemented with the help from the COTS vendors, and the rest were internally developed systems.

3.4 Communication Planning

For the communication system in the sample projects, 73% of the respondents thought it was incomplete, and 4% of respondents even indicated it was bad; only 2% of respondents thought it was good, and the rest 21% said it was rather complete.

Regarding the components of the communication system, it was shown that the communication system of most IT projects included project performance reporting and information distribution with 90.4% of the respondents confirming they were covered in their projects. But only 17.2% of the respondents agreed that the activity of communication planning was included in the communication system. And 9.6% of the respondents had no idea about their communication system.

68.15% of the respondents said that their communication plan was developed by the project manager alone, and only 15.92% of the respondents confirmed that the communication plan was developed in cooperation with stakeholders, and 7.64% of the
respondents said that the communication plan was just a copy from last projects.

Table 2. Components of Communication System

<table>
<thead>
<tr>
<th></th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication Planning</td>
<td>17.2</td>
</tr>
<tr>
<td>Information Distribution</td>
<td>90.4</td>
</tr>
<tr>
<td>Performance Report</td>
<td>90.4</td>
</tr>
<tr>
<td>No Idea</td>
<td>9.6</td>
</tr>
</tbody>
</table>

The communication plan of most IT projects included communication method, communication channel, and communication policy and protocol, which was indicated by 90.4% of the respondents. But only 17.2% of the respondents confirmed that the stakeholder-information analysis was covered in their communication plan.

Table 3. Content of Communication Plan

<table>
<thead>
<tr>
<th></th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stakeholder-Information Analysis</td>
<td>17.2</td>
</tr>
<tr>
<td>Communication Method</td>
<td>90.4</td>
</tr>
<tr>
<td>Communication Channel</td>
<td>90.4</td>
</tr>
<tr>
<td>Communication Policy and Protocol</td>
<td>90.4</td>
</tr>
<tr>
<td>No Idea</td>
<td>9.6</td>
</tr>
</tbody>
</table>

Regarding the communication protocol, it was found that most IT projects included communication system and policy, information distribution system and information access and security system, which were confirmed by 90.4% of the respondents. The information feedback was not given high attention, with only 9.4% of the respondents confirming that there were some protocols concerning the information feedback. The meeting management was also ignored, with only 3% of respondents mentioned it. The information filing and storage was also not paid enough attention, with 22% of respondents giving positive feedback about it.

Table 4 Content of Communication Protocol

<table>
<thead>
<tr>
<th></th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Information Feedback System</td>
<td>9.6</td>
</tr>
<tr>
<td>Information Filing and Storage System</td>
<td>21.7</td>
</tr>
<tr>
<td>Information Access and Security System</td>
<td>90.4</td>
</tr>
<tr>
<td>Information Distribution System</td>
<td>90.4</td>
</tr>
<tr>
<td>Communication system and policy</td>
<td>90.4</td>
</tr>
<tr>
<td>Meeting Management System</td>
<td>3.0</td>
</tr>
<tr>
<td>No Idea</td>
<td>9.6</td>
</tr>
</tbody>
</table>

The project manager is the heart of the whole process of communication. The PMI cites that 90% of the project manager’s time is invested in communication [5]. For most IT projects, especially the large ones, the communication between the project team and the high level management board was though project manager, and the team leader also acted as the bridge between the lower level team members and project manager as the project manager cannot afford time in communicating with every team member. This was confirmed by 89.8% of the respondents.

Like the communication between the project team and high level management, the communication between the project team and other functional departments was also through the team leader and the project manager. 96.18% respondents confirmed this communication channel.

Formal communication was the main communication mode. Our study indicated that for the communication between project team and third party agencies (such as government agency, standard agency, and consultancy agency), 27.52% of the respondents communicated with the stakeholders using formal communication, and 51.84% of the communication used mainly formal communication. Normally they used one-way communication, and only 17.89% of the respondents tried two-way communication.

For the communication between project team and management, more than half (64.68%) of the respondents used mainly formal communication, and 27.52% of the respondents were using pure formal communication. Only 16.51% of the respondents expressed that their communication was two-way.

For the communication between the project team and supplier or contractor, 60.65% of the respondents confirmed that the formal communication was the main method, and another 27.78% of the respondents also use mainly formal communication. The one-way communication was used by most of the respondents (81.94%).

For the communication between project team and client in the outsourced projects, the formal communication still was the main communication type for most of the respondents and almost all used one-way communication.

Regarding the communication media and tool, it was found that many kinds of tools, traditional and modern, are available. Our study showed that the traditional tools such as report, fax, telephone, meeting, wall board, interview etc. were adopted by all the projects.

More and more companies were using modern IT tools. It was found that all the respondents have used email, and more than half (55.4%) had communication software. Instant chat tools such as MSN, QQ, and Skype were also used in the communication by almost half of the respondents (48.4%); almost half of the respondents can communicate with teams through project website; and more than 1/3 of the respondents have adopted video conference as a communication mean.
Table 5. Communication Means

<table>
<thead>
<tr>
<th></th>
<th>3rd party agency</th>
<th>Management board</th>
<th>Within project team</th>
<th>Suppliers or subcontractors</th>
<th>Client</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure formal</td>
<td>27.5</td>
<td>27.5</td>
<td>27.5</td>
<td>27.8</td>
<td>16.5</td>
</tr>
<tr>
<td>Majority formal</td>
<td>51.8</td>
<td>64.7</td>
<td>45</td>
<td>60.7</td>
<td>42.7</td>
</tr>
<tr>
<td>Majority informal</td>
<td>20.6</td>
<td>7.8</td>
<td>27.5</td>
<td>11.6</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>3rd party agency</th>
<th>Management board</th>
<th>Within project team</th>
<th>Suppliers or subcontractors</th>
<th>Client</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mainly one-way</td>
<td>82.1</td>
<td>83.5</td>
<td>78</td>
<td>81.9</td>
<td>57.3</td>
</tr>
<tr>
<td>Mainly two-way</td>
<td>17.9</td>
<td>16.5</td>
<td>22</td>
<td>18.1</td>
<td>6.9</td>
</tr>
</tbody>
</table>

Regarding the usual place for communication, it was found that offices and meeting rooms were always the main meeting places. Only 15.3% of the respondents confirmed that the dining room or restaurant was also good for communication, as many problems were sorted out there.

Table 6. Communication Means and Channel

<table>
<thead>
<tr>
<th></th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Letter, Report</td>
<td>100.0</td>
</tr>
<tr>
<td>Fax</td>
<td>100.0</td>
</tr>
<tr>
<td>Telephone</td>
<td>100.0</td>
</tr>
<tr>
<td>Meeting</td>
<td>100.0</td>
</tr>
<tr>
<td>Email</td>
<td>100.0</td>
</tr>
<tr>
<td>Video Conference</td>
<td>39.5</td>
</tr>
<tr>
<td>Instant Chat Tool</td>
<td>48.4</td>
</tr>
<tr>
<td>Interview</td>
<td>100.0</td>
</tr>
<tr>
<td>Wall Board</td>
<td>100.0</td>
</tr>
<tr>
<td>Software Package</td>
<td>55.4</td>
</tr>
<tr>
<td>Project Website</td>
<td>44.6</td>
</tr>
</tbody>
</table>

It is very important to organize an effective meeting. It was found that the meeting had not been well organized in many projects. For example, about half (46.5%) of the respondents said not all their meetings could be held on time; only 16.6% of the respondents confirmed almost all the attendants could positively participate in the meeting discussion; and only 20.4% of the respondents thought their meeting could achieve their goal.

Table 7. Meeting Management

<table>
<thead>
<tr>
<th></th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meeting can be held on time</td>
<td>46.5</td>
</tr>
<tr>
<td>All the participants can present</td>
<td>78.3</td>
</tr>
<tr>
<td>All the participants positively participate in the discussion</td>
<td>16.6</td>
</tr>
<tr>
<td>The meeting can achieve its goal</td>
<td>20.4</td>
</tr>
<tr>
<td>The meeting is necessary</td>
<td>84.7</td>
</tr>
</tbody>
</table>

3.5 Information Distribution and Management

Information distribution and management is an important part of the communication management, which is a determinant of effective communication. We investigate the following aspects:
- Stakeholder’s comment on information distribution system
- Content of information distribution system
- Information feedback
- Comment on information management

It was found that 74% of the respondents thought that their information distribution system was incomplete; 2% said it was bad; and only 10% confirmed that it was rather complete.

Regarding the content of information distribution management system, our study showed that information release authority and information release frequency management were included in communication management of most projects (90.4%). But only 7.6% of the respondents confirmed that the information version control and information maintenance system were in place.

Table 8. Content of Information Distribution System

<table>
<thead>
<tr>
<th></th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Information Release Authority</td>
<td>90.4</td>
</tr>
<tr>
<td>Information Release Frequency</td>
<td>90.4</td>
</tr>
<tr>
<td>Information Version Control</td>
<td>7.6</td>
</tr>
<tr>
<td>Information Maintenance System</td>
<td>7.6</td>
</tr>
</tbody>
</table>

The feedback to other team member’s communication is very crucial to the process of communication. Our study showed that the information feedback was negative. 72% of the respondents expressed that they rarely provided feedback, and 15% admitted that they almost provided no feedback on received information. Only 13% said that they could provide feedback on some of the information.

The information management system is a very important part of the communication management, which regulate how the information is managed such as what kind information should be filed, where to store the information,
how to access the information, the privilege of information access, the security of information etc. It was found that the information management system required much improvement. 80% of the respondents believed that the information management system in their projects was incomplete; 7% said it was bad; and only 10% expressed that it was rather complete.

It is very important to have a special team member to manage the information in the IT project such as information collection, information distribution management, information version control etc. It was found that most projects have no special information controller, and only 17.20% of the respondents confirmed that they had this role in their projects.

3.6 Project Performance Reporting

According to PMBOK, The performance reporting process involves the collection of all baseline data and distribution of performance information to stakeholders. To gain a better understanding about project performance reporting, we investigate the following areas:

- Scope of information distribution
- Information customization
- Availability of required information

The scope of information distribution specifies a specific kind of information for each stakeholder, offering them the related information to understand the project. About 15% of the respondents said their scope of information distribution could cover almost all the stakeholders and other related departments; more than 2/3 (73%) of the respondents reported that the scope of the information distribution could involve important stakeholders and other important functional departments, but 12% of the respondents confirmed that just the core stakeholders are covered in the scope of information distribution.

Information customization is very important in the communication process. Proper information customization according to the information type and depth can save time and energy, improving communication efficiency. We found that information customization was far from satisfactory. About 2/3 of the respondents (71%) expressed that most information about the project they received was in standard reports and forms without much customization; 24% of the respondents said the information they received was just simply categorized with little customization; and only 6% confirmed the information were properly customized.

Despite much information many stakeholders still complained they could not get the information they wanted. It was found that 75% of the respondents claimed that they had to spend much time and energy to find the information they desired; and only 13% confirmed that the received information could basically meet their demand.

3.7 Communication Performance

There are many ways to measure the communication performance, such as:

- Stakeholder’s understanding about the communication system
- Use of communication tool
- Information lost in the transmission
- Information sharing among the stakeholders
- Cooperation among the project team
- Information clarification
- Information distribution efficiency
- Convenience of information access

For the stakeholder’s understanding about the communication system, it was found that not all the stakeholders involved in the project understood the communication. More than 2/3 (73%) of the respondents have limited understanding about the communication, and 11% even claimed that they have no understanding of the communication system. Only 16% could basically understand the communication system.

Regarding the application of the communication tool, 73% of the respondents had a limited use of the available communication tools, and 13% badly or improperly use the communication tools. 15% of the respondents confirmed that they could use the available communication tools.

Information lost in the transmission has a great effect on the communication efficiency. It was found that the information lost in the transmission was rather serious. 11% of the respondents claimed that the information lost was common for them, and 76% also confirmed that the information could only keep the basic meaning. Only 13% expressed that there was almost no information lost.

The information sharing among the stakeholders and project team best reflect the communication efficiency in the project. It was found that only 2% of the respondents thought they had high information sharing among the stakeholders and project team; 56% expressed that the information sharing among the stakeholders was moderate, and 42% claimed that they had a low information sharing.

Cooperation among the project team greatly affects the communication efficiency among team members. It was found that more than half of respondents (68%) thought that the cooperation among the team was just moderate, and about 21% of the respondents claimed that their cooperation was low. 11% said their cooperation was high.

Proper clarification helps communicators to understand each other. It was found that clarification was neglected by most communicators. 82.17% of the respondents had no practice to clarify meaning in their communication.
The distribution efficiency is the key to the high efficiency of communication. In our study, it was found that the information distribution efficiency was rather low, and information exchange among the stakeholders was very slow. 83% of the respondents claimed that the information distribution efficiency was slow, and only 6% confirmed it was rather high.

Convenient access to the project information helps to facilitate communication. In the investigation, 69% of the respondents thought the access to information was not very convenient, and 15% claimed that it was even difficult to access the information they wanted.

Involvement of stakeholders in the communication is crucial to effective communication. It was found that 83.44% of the respondents failed to take initiative to communicate with other team members; only 16.6% confirmed that they could initiate communication with other project stakeholders.

4. Summary and Conclusion

Our survey found that:

- The communication resource available to IT projects in China was rather rich.
  1) The organizations are paying more attention on project management. Some organizations have set up project management office, and many organizations have introduced consultancy service from the outside agency to acquire project management knowledge.
  2) Various kinds of modern communication media and tools were available. But these communication resources failed to be fully utilized.
- The experience of project management, especially communication management, was not rich.
- The communication was far from satisfactory despite of having a communication plan.
- There were too much upward and downward communication with little horizontal communication across different departments and different parties; there was also too much formal communication without the use of informal communication; and almost all the communication was oneway with very little feedback to the information sender.
- During the communication, the information customization was not enough. Extra time and energy were required to find the interested information despite much information was distributed regularly.
- There was little clarification by the information sender, and sometimes misunderstandings occurred by the information receiver. Lack of clarification of information is very common; the team member tends to guess other’s meaning and assumes their understanding. The project team often misuse communication tool and the information is not properly managed.
- The information management was not enough. The information was not filed appropriately and could not be conveniently accessed.
- The mutual understanding among the project team members was not good, and cooperation among the stakeholders was not satisfactory. The stakeholder analysis on information requirement is still a problem due to lack of experience.

Our study has several limitations. Firstly, the sample is relatively small and the respondents from each type of project are few; secondly, the data came mainly from questionnaires, which cannot cover all aspects of communication management; thirdly, the distribution of respondents is uneven (mostly from Xian and Suzhou) due to the convenience of sampling. Finally, the respondent’s answer may be unreliable due to personal preference, although we find no indication of this.

More studies from other major Chinese cities such as Beijing, Guangzhou and Shanghai are needed to confirm our findings. We plan to study the root causes of poor communication in Chinese projects, which may include lack of experience, influence of traditional culture, and government interference. We are also planning to develop a communication model for Chinese IT projects, taking into account the local and cultural tendency of the project team.

5 References

[8] The Importance of Effective Communication, College of Business Administration, Northeastern University
Comments as a Sublanguage:  
A Study of Comment Grammar and Purpose

Bradley L. Vinz and Letha H. Etzkorn  
Computer Science Department, University of Alabama In Huntsville, Huntsville, Alabama, USA

Abstract — A sublanguage is a subset of a natural language (such as English), typically occurring in restricted domains. In the past, Etzkorn et al. [1] explored whether comments in computer software could be considered a sublanguage of English. They examined the grammar of comments: sentence-style vs. non-sentence style, tense, mood, and voice. They also examined the subject matter (purpose) of comments, and the degree to which comments exhibit a telegraphic style.

In this paper, we perform a follow-up study to the original Etzkorn, et al. study. Whereas the original study analyzed an arbitrarily-chosen subset of comments, in our study we analyze all comments in each package examined. Our study also investigates a new domain not covered in the previous study. Finally, we compare our results to the Etzkorn et al. results.

The results of this study are important for information retrieval approaches to program comprehension, which examine comments and identifiers in software.

Keywords: Program comprehension, comments, sublanguage, syntax, and grammar.

1 Introduction

A sublanguage is a subset of a natural language. Grishman and Kittredge [2] discussed sublanguages such as “naval telegraphic transmissions” and “car repair manuals,” among others.

Comments in computer software use natural language to describe the software. Over the last several years, researchers working in program comprehension have begun developing tools that automatically analyze the comments and identifiers (variable names and function names) in computer software.

Program comprehension can be defined as any activity that uses dynamic or static methods to reveal the properties of computer software. Back in the 1980s and 1990s, program comprehension focused primarily on analyzing the code itself. This often included comparing flow graphs to a plan library of known constructs. However, formal non-heuristic approaches to program comprehension have been shown to be NP-hard and their success was often illustrated only in toy domains [3]. For this reason, heuristic approaches acquired new importance.

Recently, much research in program comprehension has focused on applying information retrieval techniques to the identifier names (function names and variable names) and comments in computer software [4][5]. For this reason, understanding the characteristics of comments and identifiers is important.

This study examines the characteristics of comments which make them a sublanguage of the English language.

2 Background

Our literature survey includes previous studies of comment grammar and syntax, and previous studies of identifier (variable and function name) syntax. We have also included some brief background information on some program comprehension research areas for which the knowledge of comment formats can be important.

2.1 Previous comment and identifier studies

Recent work in identifier analysis includes that of Lawrie, et al. [6], who examined rules for well formed identifiers previously introduced by Deißenböck and Pizka. Lawrie determined that violations exist which can be identified by a concept mapping. Lawrie further found that programmers tend to use a limited vocabulary. Additionally, Lawrie et al. [7] examined identifier quality, where quality was defined as whether identifiers are constructed out of dictionary words or known abbreviations. Also, Field et al. [8] developed algorithms to separate identifiers into their constituent parts.

Work studying comment grammar and formats includes the Etzkorn et al. [1] work previously mentioned. This work analyzed eleven C++ software packages drawn from the domains of real time software, text analysis software, database software, and mathematical software. In this work, the first 100 comments from each packages’ .h files and the first 100 comments from each packages’ .cpp files were collected in random order for analysis. The Etzkorn et al. work is referred to in more detail later in this paper, where we compare the results of our research to this earlier work.

Other work in comment analysis includes that of Raskin [9], who stated that self-documenting code and
automatic documentation are not enough, that inline comments on the same line as the code are too brief and that several lines should be used. Also, Mason [10] discussed the necessity for teaching students how to write comments.

2.2 Recent program comprehension studies that analyze comments and identifiers in software

Research studies in program comprehension that could potentially make use of the comment analysis provided by our research include the areas of static concept location, dynamic search/software reconnaissance, traceability recovery.

Recent work on static concept location includes that of Marcus et al. [5]. Marcus decomposed source code into a set of documents (file, class, function, and interface) and used informal token understanding to match user queries to documents using latent semantic indexing.

Recent work in software reconnaissance includes that of Poshyvanyk et al., [11], who combined a probabilistic ranking of events observed as a program executes with latent semantic indexing of the source code to identify sections of code relative to a feature of interest.

Recent traceability analysis includes the work of Hayes and Dekhtyar [12], who concluded that Vector Space Information Retrieval (VSIR) and Probabilistic Information Retrieval (PIR) should be considered baseline tracing methodologies. They investigated the use of a thesaurus to enhance VSIR.

3 Study methodology

This study was performed on three independent C++ Artificial Life software packages: Spirit Development Kit [13], xNeoteries [14], and Travis’ 2D Artificial Life Engine [15], and on the Watson C++ graphical user interface (GUI) package [16]. The Watson package was included since we had previously used it in a different, former study; also, in our future research we intend to further analyze whether GUI packages have different comment characteristics than other packages. Some researchers, such as Lorenz and Kidd [17], maintain that C++ GUI code is different from code in other C++ packages; it will be interesting to determine whether that is true of comments. However, that is not the focus of our current study, where we examined the above packages as a group.

Comments in this study were analyzed three different ways: for format/syntax/grammar, for purpose/content, and for type of grouping. The format/syntax/grammar analysis divides comments into sentence form and non-sentence form. When comments are determined to be in sentence form, they are further analyzed for type of grammar. When comments are determined not to be in sentence form, they are further analyzed for different subtypes. Table 1 shows all the categories used for format analysis.

<table>
<thead>
<tr>
<th>Table 1. Common Comment Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sentence Form Comments</strong></td>
</tr>
<tr>
<td>o SF-1 (PT,IDM,AV)  — The comment is in sentence-form, present tense, indicative mood, and active voice.</td>
</tr>
<tr>
<td>o SF-2 (PT,IDM,AV,MS) — The comment is in sentence-form, present tense, indicative mood, active voice, and has a missing subject.</td>
</tr>
<tr>
<td>o SF-3 (PT, IPM, AV) — The comment is in sentence-form, present tense, imperative mood, and active voice.</td>
</tr>
<tr>
<td>o SF-4 (PT, IDM, PV) — The comment is in sentence-form, present tense, indicative mood, and passive voice.</td>
</tr>
<tr>
<td>o SF-5 (PT,IDM,PV,MS) — The comment is in sentence-form, present tense, indicative mood, passive voice, and has a missing subject.</td>
</tr>
<tr>
<td>o SF-6 (XT,IDM,AV,PV,MS) — The comment is in sentence form, past tense, indicative mood, either active or passive voice, and sometimes has a missing subject.</td>
</tr>
<tr>
<td>o SF-7 (FT,IDM,PV,MS) — The comment is in sentence form, future tense, indicative mood, either active or passive voice, and can have a missing subject.</td>
</tr>
<tr>
<td><strong>Non-Sentence Form Comments</strong></td>
</tr>
<tr>
<td>o NSF-1 (simple definition) — A simple definition is included inline close to the software that it describes, or Item- name — definition is included in a header block comment.</td>
</tr>
<tr>
<td>o NSF-2 (prepositional phrase) — An unattached prepositional phrase is included inline close to the software that it describes.</td>
</tr>
<tr>
<td>o NSF-3 (variable values) — Allowed values for a software variable are specified close to the software variable that they describe.</td>
</tr>
<tr>
<td>o NSF-4 (math expression) — Mathematical expressions are included near the software that they describe. Note that this type of construction would most likely be more common in a mathematical package than in a graphical user interface package.</td>
</tr>
<tr>
<td>o NSF-5 (simple tag) — A simple descriptor tag or ItemName is included near the software that describes or defines it.</td>
</tr>
<tr>
<td><strong>Other</strong></td>
</tr>
<tr>
<td>o COMP (Composite Comment) — The comment consists of several comments of different common content that are grouped together. This type of comment needs to be decomposed before further analysis.</td>
</tr>
<tr>
<td>o VC (Voided Code) — This entity refers to a chunk of code that has been commented out, possibly due to testing purposes, and can range from a single line to a large section of code.</td>
</tr>
</tbody>
</table>

The purpose/content analysis divides comments by their intent, or purpose. Some comments contain author in-
formation, some describe how the software works, some define what the software is, etc. The categories used for the purpose/content analysis are listed in Table 2.

The format/syntax/grammar and purpose/content analyses are the same as those employed by Etzkorn et al. [1]. However, the third analysis we performed, the type of grouping, is an additional analysis that was not performed by Etzkorn et al.

**Table 2. Purpose, or Content of Comments**

<table>
<thead>
<tr>
<th>Common Purpose, or Content, of Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>o <strong>Authorship</strong> — The comment contains information about the author, purpose, file name, creation date, revision dates, modification (change) summaries, copyright notice, disclaimer, etc.</td>
</tr>
<tr>
<td>o <strong>Operational Description</strong> — The comment provides an operational description of the software.</td>
</tr>
<tr>
<td>o <strong>Software Definition</strong> — The comment provides a definition of the software. This type of comment occurs in both sentence and non-sentence form.</td>
</tr>
<tr>
<td>o <strong>Software Definition Description</strong> — The comment provides a description of the definition of the software.</td>
</tr>
<tr>
<td>o <strong>Instructed Action</strong> — The comment instructs the reader to perform a certain action.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>o <strong>COMP</strong> — Composite Comment. The comment consists of several comments of different common content that are grouped together. This type of comment needs to be decomposed before analysis can be performed.</td>
</tr>
<tr>
<td>o <strong>VC (Voided Code)</strong> — The entry is voided code.</td>
</tr>
</tbody>
</table>

**Table 3. Type of Comment Grouping**

<table>
<thead>
<tr>
<th>Comment Grouping</th>
</tr>
</thead>
<tbody>
<tr>
<td>o <strong>File Header Block</strong> — The file header block is normally located at the top of the file.</td>
</tr>
<tr>
<td>o <strong>Module Header Block</strong> — The module header block is located above a function, method, or class.</td>
</tr>
<tr>
<td>o <strong>Inline</strong> — This type of comment group is associated with a small amount of code, usually a few lines. It typically contains fewer words than in a block comment, and may span more than one line. It can be located above the source code line, appended to the end of the source code line, or located below the source line code.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>o <strong>VC (Voided Code)</strong> — The entry is voided code.</td>
</tr>
</tbody>
</table>

In Etzkorn et al., they discussed some of the different problems that occur when automatically analyzing block comments, such as file header block or module header block comments compared to analyzing inline comments.

For example, with block comments pronoun analysis must occur across several sentences, whereas with inline comments this pronoun relationship across sentences does not often occur due to the limited number of sentences in an inline comment. However, inline comments sometimes use a pronoun to refer to the code they describe: “This opens the file” would mean that the code statement the comment is attached to will perform a file open.

However, Etzkorn et al. did not analyze block vs. inline comments. In the current study we will report only the quantities of comments that occur in each type of grouping. This will indicate to researchers working on automated comment analysis what type of analysis is most likely to be useful overall. In the future we plan to further analyze the groupings in terms of the kinds of grammar/syntax and purpose that tend to occur for different kinds of groupings. The types of groupings analyzed in this study are described in Table 3 provided above.

An expanded set of descriptions/meanings of the Format/Syntax/Grammar analysis categories are provided below. We did not include further description of the Common Grouping categories, as we think those are self-explanatory.

### 3.1 Format/syntax/grammar analysis

Example comments that illustrate the different sentence form comment types are provided below:

**SF-1 Present tense, indicative mood, active voice:**

This class handles all database accesses.
This class opens and closes files.

**SF-2 Present tense, indicative mood, active voice, missing subject:**

Handles all database accesses.
Opens and closes files.

**SF-3 Present tense, imperative mood, active voice:**

Handle all database accesses.
Open the file.

**SF-4 Present tense, indicative mood, passive voice:**

Database accesses are handled by this class.
Files are opened and closed by this class.

**SF-5 Present tense, indicative mood, passive voice, missing subject:**

Is used for database accesses.
SF-6 Past tense, indicative mood, either active or passive voice, sometimes with a missing subject:

Database access failed.
Failed to open a file.

SF-7 Future tense, indicative mood, either active or passive voice, sometimes with a missing subject:

This will perform database accesses.
This will open and close files.

Example comments that illustrate the common syntactic patterns in non-sentence form comments are given below:

NSF-1 A simple definition comment is included inline close to the software it describes:

Doubly linked binary tree.

NSF-2 An unattached prepositional phrase is included close to the software it describes:

To override the same function in the base class.
For cleaning up the list.

NSF-3 Allowed values for a variable are specified close to the variable:

1= true, 0=false
>99=okay, <=99 = critical

NSF-4 Mathematical expressions are included near the software they describe:

3.14 * R * R = area of the circle

3.2 Purpose/content analysis

Example comments that illustrate the different categories used for the purpose/content analysis are provided below:

Authorship (self-explanatory, hopefully includes useful information for the maintainer of the code)

Operational description:

This class handles all database accesses.

Software definition:

This is a pointer to the student node.

Software definition description:

The function Visit() defines the search procedure for the binary tree.

Instructed action:

Be careful, don’t change this constant unless you really understand the code.
See related code in the Move function.

4 Results of the comment study

The results of our format/syntax/grammar study are provided in Table 4. In our study, we have 62.17% sentence form comments and 30.02% non-sentence comments, whereas in the Etzkorn study, 53% of comments were in sentence form and 47% were in non-sentence form. Thus, our study was weighted more heavily toward sentence form comments. This is a promising result for program comprehension researchers, as sentence form comments would presumably be more easily analyzed.

In Table 5, we compare the grammatical tense of sentence-form comments in this study to the previous results in the Etzkorn et al. study.

Table 4. Format/Syntax/Grammar Study Results

<table>
<thead>
<tr>
<th></th>
<th>Numerical Results (our study)</th>
<th>Results as a Percentage (our study)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SF-1</td>
<td>137</td>
<td>10.42%</td>
</tr>
<tr>
<td>SF-2</td>
<td>40</td>
<td>3.04%</td>
</tr>
<tr>
<td>SF-3</td>
<td>893</td>
<td>67.91%</td>
</tr>
<tr>
<td>SF-4</td>
<td>29</td>
<td>2.21%</td>
</tr>
<tr>
<td>SF-5</td>
<td>26</td>
<td>1.98%</td>
</tr>
<tr>
<td>SF-6</td>
<td>53</td>
<td>4.03%</td>
</tr>
<tr>
<td>SF-7</td>
<td>137</td>
<td>10.42%</td>
</tr>
<tr>
<td>Total (Sentence Form)</td>
<td>1315</td>
<td>62.17%</td>
</tr>
<tr>
<td>NSF-1</td>
<td>516</td>
<td>81.26%</td>
</tr>
<tr>
<td>NSF-2</td>
<td>55</td>
<td>8.66%</td>
</tr>
<tr>
<td>NSF-3</td>
<td>40</td>
<td>6.30%</td>
</tr>
<tr>
<td>NSF-4</td>
<td>3</td>
<td>0.47%</td>
</tr>
<tr>
<td>NSF-5</td>
<td>21</td>
<td>3.31%</td>
</tr>
<tr>
<td>Total (Non-Sentence Form)</td>
<td>635</td>
<td>30.02%</td>
</tr>
<tr>
<td>VC</td>
<td>165</td>
<td>7.80%</td>
</tr>
<tr>
<td>Total (Other)</td>
<td>165</td>
<td>7.80%</td>
</tr>
<tr>
<td>Total (Overall)</td>
<td>2115</td>
<td>100%</td>
</tr>
</tbody>
</table>
Table 5. Comparison of Tense: Our Study vs. Etzkorn et al.

<table>
<thead>
<tr>
<th>Tense</th>
<th>Results (our study)</th>
<th>Results (Etzkorn et al.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Present Tense SF—SF-5</td>
<td>85.56%</td>
<td>77%</td>
</tr>
<tr>
<td>Past Tense SF-6</td>
<td>10.42%</td>
<td>4%</td>
</tr>
<tr>
<td>Future Tense SF-7</td>
<td>4.03%</td>
<td>16%</td>
</tr>
<tr>
<td>Other</td>
<td>3%</td>
<td></td>
</tr>
</tbody>
</table>

The unexpected result from this study is that all sentence form comments could be categorized in the given tenses—there were no entries in the "other" category as in the earlier study. Otherwise, it seems that past tense and future tense traded places between the two studies. The most useful result from this analysis is that very much the preponderance of the comments are in present tense, more so with this study than even with the Etzkorn et al. study. This is a major advantage for researchers trying to develop very simple program comprehension tools, as it means that stemmers (tools that extract the roots of words, particularly of verbs) can largely be dispensed with: the preponderance of the verbs will be in present tense.

Table 6. Comparison of Mood: Our Study vs. Etzkorn et al.

<table>
<thead>
<tr>
<th>Mood and Voice</th>
<th>Results (our study)</th>
<th>Results (Etzkorn et al.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indicative Mood, Active Voice, Subject Present SF1, SF4, SF6, SF7</td>
<td>27.08%</td>
<td>30%</td>
</tr>
<tr>
<td>Indicative Mood, Active Voice, Missing Subject SF2</td>
<td>3.04%</td>
<td>11%</td>
</tr>
<tr>
<td>Imperative Mood SF3</td>
<td>67.91%</td>
<td>58%</td>
</tr>
<tr>
<td>Indicative Mood Passive Voice SF-5</td>
<td>1.98%</td>
<td>1%</td>
</tr>
</tbody>
</table>

In Table 6 above, we compare the mood of sentence-form comments in this study to the previous results of the Etzkorn et al. study. Note that although SF6 and SF7 could include passive voice and/or a missing subject, these occurrences are quite rare (which is why they were not broken into separate categories in our study), so we included those in the first category in Table 6. The primary difference between our study and the Etzkorn et al study here is the reduction in missing subjects. Again, this is a promising result for program comprehension researchers, as it clarifies the agent of the verb action.

In Table 7, we compare the non-sentence form comments in this study versus those in the Etzkorn et al. study. Note that the percentages here have been calculated not including NSF-5, since the Etzkorn et al. results did not include this category. The biggest difference here is NSF-4. This is probably a difference due to the domains analyzed, for example, it makes sense that since the Etzkorn et al. study included mathematical software packages that the mathematical equations category would be larger than in artificial life and graphical user interface packages.

Table 7. Comparison of Non-Sentence Form Comments: Our Study vs. Etzkorn et al.

<table>
<thead>
<tr>
<th>Non-Sentence Form</th>
<th>Results (our study)</th>
<th>Results (Etzkorn et al.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NSF-1</td>
<td>84.04%</td>
<td>85%</td>
</tr>
<tr>
<td>NSF-2</td>
<td>8.96%</td>
<td>4%</td>
</tr>
<tr>
<td>NSF-3</td>
<td>6.51%</td>
<td>2%</td>
</tr>
<tr>
<td>NSF-4</td>
<td>0.49%</td>
<td>9%</td>
</tr>
<tr>
<td>NSF-5</td>
<td>not included in percentage</td>
<td>N/A</td>
</tr>
</tbody>
</table>

In Table 8, we compare the purpose/content of comments in this study to the previous results in the Etzkorn et al. study. The authorship category is included in the numerical results because we did collect that data. However, it is not included in the percentage calculations because the author data was not included in the Etzkorn et al. study.

The primary difference here is most likely the Software Definition versus the Software Definition Description numbers, which have reversed in relative magnitude. It is possible that the Software Definition Description numbers would increase if some individual sentences from the Composite Comment (COMP) category were included: the Etzkorn et al. study did not include this category. Note also that the Etzkorn et al. study ignored voided code.

Another difference is that the operational description category is lower here than in the Etzkorn et al. study, although in both studies the operational description category is the largest category.

In Table 9, we show the results of our comment grouping study. The Etzkorn et al. study did not perform a similar analysis. The primary interesting result here is that the preponderance of comments is inline comments. The relatively small proportion of file header block and module header block comments means that the added difficulty of across sentence pronoun analysis would have a fairly small impact on the overall accuracy of a program comprehension system, and therefore could possibly be ignored.
Table 8. Comparison of Purpose/Content:
Our Study vs. Etzkorn et al.

<table>
<thead>
<tr>
<th></th>
<th>Numerical Results (our study)</th>
<th>Results as a Percentage (our study)</th>
<th>Results (Etzkorn et al.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authorship</td>
<td>87</td>
<td>not included in percentage</td>
<td>N/A</td>
</tr>
<tr>
<td>Operational Description</td>
<td>200</td>
<td>43.11%</td>
<td>60%</td>
</tr>
<tr>
<td>Software Definition</td>
<td>1064</td>
<td>6.24%</td>
<td>36%</td>
</tr>
<tr>
<td>Software Definition</td>
<td>154</td>
<td>17.99%</td>
<td>3%</td>
</tr>
<tr>
<td>Instructed Action</td>
<td>4</td>
<td>8.10%</td>
<td>1%</td>
</tr>
<tr>
<td>COMP</td>
<td>440</td>
<td>17.83%</td>
<td>N/A</td>
</tr>
<tr>
<td>VC</td>
<td>166</td>
<td>6.73%</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 9. Results of Comment Grouping Study References

<table>
<thead>
<tr>
<th></th>
<th>Numerical Results (our study)</th>
<th>Results as a Percentage (our study)</th>
</tr>
</thead>
<tbody>
<tr>
<td>File Header Block</td>
<td>144</td>
<td>6.81%</td>
</tr>
<tr>
<td>Inline</td>
<td>1668</td>
<td>78.87%</td>
</tr>
<tr>
<td>Module Header Block</td>
<td>138</td>
<td>6.52%</td>
</tr>
<tr>
<td>VC</td>
<td>165</td>
<td>7.80%</td>
</tr>
<tr>
<td>Total</td>
<td>2115</td>
<td>100%</td>
</tr>
</tbody>
</table>

5 Conclusions and future work

The results of our study compared fairly well to the earlier Etzkorn et al. study; however, there were some differences. One difference is that in our study even more comments were in present tense than in the earlier study. This means developers of very simple program comprehension tools could largely dispense with stemmers and still expect relatively good results. Another difference is a reduction in missing subject comments, which should also make analysis easier. Our study had a reduced number of mathematical equations, probably due to Etzkorn et al.’s inclusion of mathematical packages in their analysis. Our comment grouping study (Etzkorn et al. did not perform a similar study) indicated a relatively small number of header block comments, which reduces the need for cross-sentence pronoun alignment.

Future research includes determining whether different domains have different characteristics, and further analysis of what kinds of grammar/syntax occur in different kinds of comment groupings.

6 References


Domain Analysis: A Case Study of Engineering RFID Systems in Supply Chain

Leonardo Barreto Campos
Department of Computer Engineering
Federal University of Vale do São Francisco
Juazeiro, BA, Brazil
leonardo.campos@univasf.edu.br

Sérgio Donizetti Zorzo
Computer Department
Federal University of São Carlos
São Carlos, SP, Brazil
zorzo@dc.ufscar.br

Abstract – The information integration have increased in Supply Chain Management after use of the Radio Frequency Identification Systems. The information integration and reuse are possible because the systems presents in Supply Chain use very reusable information through of repeated scenarios. However, there is not a domain analysis process, based on a well defined set of guidelines and steps for the supply chain domain. This article presents a case study of Domain Analysis for Engineering Radio Frequency Indentification Systems in Supply Chain. A feature diagram for Supermarket Supply Chain will showed as result of the case study.

Keywords: Domain Engineering, Software Product Lines, RFID Systems, and Supply Chain Management

1. Introduction

Software reuse is a key factor for improving quality and productivity [1] because to identify common requirements and software architectures in systems. In this context, the Domain Analysis is presented as important activity in development of reusable software.

The Supply Chain Management (SCM) is the “management and control of all materials and information in the logistics process form acquisition of raw materials to delivery to the end user” [2]. In this definition, is shown the necessity of integration between Radio Frequency Idetification and Supply Chain Management.

The System of Systems presents in SCM comprehends large scale distributed systems that are comprised of RFID Systems. The RFID technology is defined as: “Radio Frequency IDentification, or RFID, is a generic term for technologies that use radio waves to automatically identify people or objects” [3].

Thus, the RFID Systems in Supply Chain Management searches to guarantee interoperability providing, for example, accurate and real-time information on inventory of the organizations, product recalls and communications between organizations of the supply chain.

In concordance with [4], our goal is define a domain analysis approach for RFID Systems in Supply Chain Management that: “to find ways to extract, organize, represent, manipulate and understand reusable information, to formalize the domain analysis process, and to develop technologies and tools to support it”[4].

The approach represents a synthesis of several advanced software reuse process, including domain engineering, domain analysis, requirements engineering, and software product lines.

The format for this paper is as follows: Section 2 discusses eight software reuse processes distributed in domain engineering and software product lines. Section 3 presents the RFID Systems standardized for EPCglobal Inc.™. The Section 4 to describe the approach in case study. Finally, the Section 5 presents the concluding remarks and future work
2. Software Reuse Processes

The software reuse has been practiced since programming began [5]. The goal of the software reuse is the use of existing software or software knowledge to construct new software. There is a large number of software reuse processes but two of them deserve special attention [6]: Domain Engineering and, Software Product Lines, as can be seen in the next sections.

2.1. Domain Engineering

The Domain Engineering is defined in [7] as “the activity of collecting, organizing, and storing past experience in building systems or parts of systems in a particular domain in the form of reusable assets, as well as providing an adequate means for reusing these assets when building new systems.”

In this context, we have the first domain engineering approach called Draco. This approach is based on transformation technology and was developed by James Neighbors in his Ph.D. work [8]. The main idea introduced by Draco is to organize software construction knowledge into a number of related domains. Each Draco domain encapsulates the needs and requirements and different implementations of a collection of similar systems.

Draco presented an initial direction to development software using Domain Engineering. However, his approach is very difficult to apply in the industrial environment due the complexity to perform activities such as writing transformations and using the Draco machine.

Described in the 90’s, Feature-Oriented Domain Analysis (FODA) is a domain analysis method developed at the Software Engineering Institute (SEI). The method presented in [9] consists of two phases: Context Analysis and Domain Modeling. The major contribution of FODA method is the feature model. An important part of this model is the feature diagram that defines three types of features: mandatory, alternative or optional features.

Next, the Feature-Oriented Reuse Method (FORM)[9], seen an elaboration of the FODA, to present four layers to classify features: capability, operating environment, domain technology, and implementations technique. Moreover, the processes do not include essential domain analysis techniques such as domain scoping.

Other important domain engineering method is Organization Domain Modeling (ODM) developed by Mark Simos. The ODM process described in [11] consists of three main phases: Plan Domain, Model Domain, and Engineer Asset Base. However, the phases does not present specific details on how to perform many of its activities. According to [7] the ODM provides a “general high-level guidance in tailoring the method for application within a particular project or organization”.

2.2. Software Product Lines

A new trend started to be explored in software reuse process is the Software Product Line area. According to [12], a software product line is: “a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way”.

In this context, [13] to present the Family-Oriented Abstraction, Specification and Translation (FAST) process. FAST focuses on pattern for software production processes that strives to resolve the tension between rapid production and careful engineering. The FAST process consists of three well-defined subprocesses: domain qualification, domain engineering, and application engineering. On the other hand, some activities in the process such as in Domain Engineering are not as simple to perform, for example, the specification of an Application Modeling Language.

Presented in [14], and [15] the Komponentenbasierte Anwendungsentwicklung (KobrA) approach provides a generic assets that can accommodate variants of a product line through framework engineering. The gap in KobrA approach is does not present guidelines to perform systematic tasks such as domain analysis and domain design.

An effort to apply the reuse concepts within the embedded systems domain is described in [16]. The Pervasive Component Systems (PECOS) approach focuses on two issues: how to enable the development of families of PECOS devices? And how pre-fabricated components have to be coupled. Some gaps was identified in PECOS, for example, in component
development, there is no guidance on how the requirements elicitation is performed, and how the components are identified.

Finally, is presented in [17] the Product Line UML-Based Software Engineering (PLUS) approach considered the most current process related to product lines. PLUS extends UML by integrating various product line engineering techniques to support UML-based product line engineering. PLUS define three general steps: Requirements, Analysis, and Design Modeling in order to provides various modeling techniques and notation for product line requirements engineering activity, use case modeling, and feature modeling. However, in Requirements and Analysis, activities related to scoping are not considered.

3. RFID Systems

The RFID Systems showed in this approach is developed and administered by EPCglobal Inc. The RFID Systems are a collection of interrelated standards for hardware, software, and data interfaces, together with core services that are operated by EPCglobal and its delegates, all in service of a common goal of enhancing the Supply Chain Management through the use of Eletronic Product Codes.

This RFID Systems are divided into five parts [18]: (i) Eletronic Product Code, (ii) Identification System, (iii) EPC Middleware, (iv) Discovery Services, and (v) EPC Information Service. Each of these parts is summarized in next sections.

3.1. Eletronic Product Code

The Eletronic Product Code (EPC) is a naming and identification scheme designed to enable the unique identification of all physical and virtual objects, assemblies and grouping of objects, and non-objects such as service [19].

An Eletronic Product Code is comprised of Header and three distinct numbers: Domain Manager number, ObjectClass number, and Serial Number. The header contain the Version Number, it enables the definition of multiples types, or versions, of EPC that differ in total bit length and number of bits in each of the four partitions.

The Domain Manager partition is used to encode the domain manager number of an EPC. The Object Class partition is used to encode the object class number of an EPC, for example: moto, car, bike, etc. The Serial Number partition is used to encode the unique number that identifies a specific object in motion in the supply chain.

3.2. Identification System

The Identification System consists of EPC tags and EPC readers. EPC tags are RFID devices that consists of a microchip and an antenna attached to a substrate. The EPC is stored on this tag, which is applied to cases, pallet or items. The other component is a reader, which creates a radio frequency field that detects radio waves. When a tag passes through a radio frequency field generated by a compatible reader, the tag reflects back to the reader the identifying information about the object to which it is attached, thus identifying that object.

3.3. EPC Middleware

The EPC readers deliver information to local business information systems using EPC Middleware. The EPC Middleware manages real-time read events and information, provides alerts, and manages the basic read information for communication to EPC Information Services and a company’s other existing information systems.

3.4. Discovery Services

The Discovery Services returns locations that have some data related to an EPC [20]. In general, a Discovery Services may contain pointers to entities other than the entity that originally assigned the EPC code. Hence, Discovery Services are not universally authoritative for any data they may have about an EPC. The important service in Discovery Services is the Object Name Service (ONS) that, given an EPC, can return a list of network accessible service endpoints that pertain to the EPC in question.
3.5. EPC Information Service

The EPC Information Service (EPCIS) provides an uniform programmatic interface to allow various clients to capture, secure, and access EPC-related data and the business transactions which that data is associated [21]. Companies that assign EPC numbers can maintain EPC Information Service servers with item information. Using EPC numbers does not require organizations to share EPC data or use other components of the system.

4. Case Study

The approach for domain analysis proposed in [22] consists of four steps: Planning, Requirements, Domain modeling, and Documentation. The next sections presents each step applied in case study of Supermarket Supply Chain.

4.1. Planning

The first activity corresponds to planning phase. This phase is based in three sub-activities (P):

P1. Domain: Encompasses to describe which supply chain will be applied the domain analysis. The supply chain analyzed is the Supermarket Supply Chain. The subdomains identified in Supermarket Supply Chain are: (i) raw material suppliers, (ii) manufactures, (iii) distribution centers, (iv) supermarkets, and (v) customers.

All subdomains in supply chain must be described into four aspects (A): (A1) Activity: that goal of subdomain in supply chain? (A2) Input: from who the subdomain receives informations in supply chain. (A3) Output: to who the subdomains send informations in supply chain. (A4) Technology: where and which goal to use RFID System in subdomain?

The raw material supplier subdomain present the following aspects: (A1) The goal is garnatir/forncecer the available materials for manufacturing, (A2) receives informations of other suppliers, (A3) raw material suppliers send informations for manufactures, and (A4) the RFID Systems apply in receiving, processing, and shipping.

In manufacture subdomain the planning aspects are: (A1) to transform raw materials into finished goods for sale, (A2) receives informations of distribution centers, (A3) send informations for suppliers, and (A4) the RFID Systems are present at various levels, for example: receiving, processing, check-points in line of production, packaging, and shipping.

The distribution center subdomain can be described in followig aspects: (A1) the goal is fulfill the demand created by their respective retailers, (A2) distribution centers receive informations from retailers; in this case the supermarkets, (A3) send informations for manufactures, and (A4) the RFID Systems are present in receiving, storage, location of products, and shipping.

In supermarkets subdomain the aspects are: (A1) monitor the stock and automatically initiates reorders, (A2) supermarket receive informations from customers, (A3) send informations for distribution centers or manufacturers directly, and (A4) the RFID System are present in receiving, smart shelves, cart, and stock.

Finally, the customer subdomain present the followig aspects: (A1) the sole purpose of existence of any supply chain, (A2) customers receive informations from supermarkets, (A3) send informations to supermarket, and in some cases to retailers (A4) the RFID Systems are present in few cases because still lacks structure for RFID Systems in subdomain.

P2. Stakeholder analysis: encompasses to analysis the people or someone who has a defined interest in the result of the project. The stakeholders identified are: (i) RFID specialist, (ii) Domain analyst, (iii) Management, and (iv) Market specialist.

The RFID specialist must be expert in EPclogal Network, RFID readers, RFID tags, your variablities, installation, and uses of hardwares. The Domain analyst is a person who conducts the domain analysis process described in this article. The Management consist in all persons responsible for their subdomains.

Finally, the Market specialist involves the gathering of business intelligence, competitive studies and assessments, market segmentation, customer plans, and the integrations of this information into a cohesive business strategy and plan [12]

P3. Domain scope: the goal this step is identify and discard subdomains described that not related with other subdomains. In this case, every subdomains described early are showed in Figure 1:
The organization which does not have any experience with Domain Analysis should choose a small but important domain. After succeeding with the first domain, the organization should consider adding more and more domains to reach its real domain.

4.2. Requirements

The goal of the Requirements activity is to describe characteristics of the domain, and understand the needs and constraints for the domain users. The requirements identification process includes the stakeholders that were identified early in analysis activity.

In tentative of minimize errors this approach make the requirements elicitation through a features. The feature definition used in this work is concordance with [9]: “a distinguishable characteristic of a concept (e.g. system, component, etc) that is relevant to some stakeholder of the concept.”

The principals features (F) identified are presented in hierarchy of four levels of major concept (C) Supermarket Supply Chain. The first level are: F1. Product, F2. Information Services, and F3. Object Name Service. Associated with Product feature are: F1.1. EPC Number, F1.2 Location, F1.3 Transaction. The next levels are showed in details in Figure 3.

The features were identified through of scenarios. The scenarios are descriptions of how a system is used in practice. Thus, the steps (S) for requirements for elicitation from scenarios are: S1. Initial stage: systems stage at the beginning of the scenario. S2. Events: normal flow of events in the scenario. S3. Alternative events: eventul events out the normal flow that can cause error for systems of systems integration and interoperability. S4. Finish stage: systems stage in conclusion of the scenario. S5. Stakeholders: list of stakeholders that had participated in scenario.

4.3. Domain Modeling

In this activity the domain will be described in function of the features identified in Requirements activity. In this context, has been defined three sub-activities (M):

M1. Commonality analysis: This first activity of the Domain Modeling consist in identify which features or requirements are common to all applications of the domain. There are different ways to identify common requirements. This approach use a based-priority domain-requirements matrix shown in Table 1. The goal is select requirements by priority for all stockholders.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Product</td>
<td>Pr2</td>
<td>Pr1</td>
<td>Pr1</td>
<td>Pr1</td>
<td>Pr1</td>
</tr>
<tr>
<td>EPC-IS</td>
<td>Pr1</td>
<td>Pr1</td>
<td>Pr1</td>
<td>Pr1</td>
<td>X</td>
</tr>
<tr>
<td>ONS</td>
<td>Pr2</td>
<td>Pr1</td>
<td>Pr1</td>
<td>Pr1</td>
<td>X</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shelve</td>
<td>X</td>
<td>Pr3</td>
<td>Pr3</td>
<td>Pr1</td>
<td>X</td>
</tr>
</tbody>
</table>

The priorities (Pr) are classified as follows: Pr1. High: the requirement ‘Pr1’ is mandatory for all subdomains and is thus a candidate to be defined as a common domain requirement. Pr2. Medium: the requirements that assists high-priority requirements to keep the funcionality of the systems. Pr3. Low: low-priority requirements to systems.

After filling of the matrix, the domain analyst must define ideal priority for commons requirements. For example, in Table 1, if the domain analyst had defined the ideal priority with ‘Pr2’, the numbers one, two, and three requirements would be selected. In other hand, the shelve feature will be classified as not-mandatory. The
requirements that not apply the subdomain must be marked with X.

M2. Variability analysis: This activity consist in identify which features are variable to applications of the domain. According with [23] in situation where a lot of effort has been made to preserve variability until very late in the development process, the systems provides greater reusability and flexibility.

In this approach, the variability are identified through of the based-priority domain-requirements matrix. The feature marked with X is examine and defined if that feature will be preserved or eliminated.

M3. Domain modeling: In this step, the commonalities and variabilities are modeled. The model applicable shown Figure 2 is concordance with[7].

4.4. Documentation

In this the requirements, identified in form of features, will be documented. According with [7] the template used for document features is showed in Table 2 for feature Product:

<table>
<thead>
<tr>
<th>F1. Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semantic</td>
</tr>
<tr>
<td>Rationale</td>
</tr>
<tr>
<td>Stakeholders</td>
</tr>
<tr>
<td>Applications</td>
</tr>
<tr>
<td>Constraints</td>
</tr>
<tr>
<td>Attribute</td>
</tr>
<tr>
<td>Priorities</td>
</tr>
</tbody>
</table>

Thus, finished the case study with support necessary for development next tasks in Domain Engineering, for example, Design and Implementatio activities.

Figure 2. Feature Diagram for Supermarket Supply Chain

5. Conclusion and Future Work
The approach proposed in this paper shown the importance of Domain Analysis in Software Reuse Process, and efficient method for building efficient and adaptive of integrated information systems.

The approach can be reusable in other supply chain integration on Systems of Systems. It shown also methods for planning, modeling, documentation, etc.

As future work, we are researching the domain design and domain implementation areas to define a domain engineering process for RFID Systems in supply chain.

Acknowledgements

The authors would like to thanks the members of the Reuse in Software Engineering (RiSE) group¹ for their suggestions during this work and Fundação de Amparo à Pesquisa do Estado da Bahia (FAPESB) for providing financial resources.

References


¹ www.rise.com.br


Automated Defect Prevention with Visual Studio Team System: a Case Study for Software Engineering

Dorota Huizinga¹, David Phung², Tae Ryu¹

¹College of Engineering and Computer Science and Department of Computer Science, California State University, Fullerton, California, USA
²Boeing Corporation, Anaheim, California, USA

Abstract This paper describes a case study of a technology-driven software development, based on principles and practices of “Automated Defect Prevention” (ADP) [1]. ADP is an approach to software engineering and management based on establishing an infrastructure that serves as the foundation of a project. This infrastructure defines people roles, necessary technology and interactions between people and technology. It also provides product visibility, automates repetitive tasks, organizes project activities, tracks project status, and seamlessly collects project data to provide quantifiable measures necessary for decision making.

The project selected for this case study was the design and construction of a voice compression and decompression application, called Vocoder, implemented using Test Driven Development (TDD) [2]. The technology used for guiding the ADP practices was Microsoft Visual Studio Team System and Foundation Server (VSTS).

The case study validated the ADP infrastructure-first approach to collaborative software development. The ability to automatically enforce and validate recommended software engineering practices and seamless visualization, tracking and measuring of project status led to the development of a quality application meeting its deadline.

Keywords: Software engineering, technology infrastructure, collaborative software development

1. Introduction

Like all other engineers, software developers transform ideas into usable products. Unlike other engineers, they create a product that is inherently intangible, transitional and often not completely testable.

Since it was originally coined at a NATO conference in 1968, the term “Software Engineering” has been widely and loosely used both in academia and in industry. At the same time for the past 40 years, we, software researchers, educators and developers have failed to achieve the level of quality and precision that has been achieved in other engineering disciplines. In fact, software defects cost US economy approximately $60 billion annually [3].

According to the IEEE standard glossary [4], “Software Engineering” is a systematic, disciplined and quantifiable approach to software development. However, a closer look at the current state of software engineering practice and education leads to the following observation: its focus is on controlling the activities of software engineers, instead of the product itself. It is, thus, more the “engineering of developers” than the “engineering of software.”

Moreover, as the lists of recommended software engineering practices grow larger, so do the software engineering textbooks, which often are very descriptive, not prescriptive in nature. Consequently, students come out of software engineering courses with an understanding of many different ways of doing software engineering but they do not really know “how to” do it.

Additionally, the aforementioned issues of software intangibility, its transitional nature and the need for a disciplined approach combined with quantifiable measures are not being addressed.

The case study described in this paper tackles the above concerns by applying a subset of practices and policies recommended by Automated Defect Prevention (ADP) [1]. ADP takes a holistic approach to defect prevention by defining six principles of software lifecycle management:

1. Establishment of the infrastructure that integrates people and technology.
2. Application of general best practices, known to avert common problems.
3. Customization of best practices to improve internal processes.
5. Automation.
6. Incremental implementation of ADP practices and policies.
This problem-averting approach to software development can be accomplished because of ADP’s two cornerstone principles: infrastructure (principle 1) and automation (principle 5). While automation is an overarching principle of ADP, it would be not possible without the appropriate technology infrastructure in place and people who interact with this technology through their well-defined roles. Therefore, the infrastructure is a foundation of ADP. This infrastructure defines people roles, necessary technology, and interactions between people and technology. This infrastructure provides product visibility, automates repetitive tasks, organizes project activities, tracks project status, and seamlessly collects project data to provide quantifiable measures necessary for decision making.

The objective of the case study described below was to illustrate the benefits of ADP in seamlessly enforcing software engineering practices throughout the software development lifecycle, thereby putting the “engineering” in software engineering.

While ADP’s implementation relies on using technologies supporting collaborative software development, the methodology itself is vendor-neutral, as it defines the necessary features rather than specific tools. Therefore, the case study required a selection of vendor-specific technologies. We have selected VSTS because it was the most appropriate for the team and the application under the development.

Even though all six ADP principles were applied in the case study, the focus was on principles 1, 2, 4, and 5.

These principles guided development and management of the speech data compression/decompression application by providing continuous product visibility and comprehensive traceability from requirements, through design, construction, and testing to deployment.

2. VOCODER: Project Definition and Objectives

The product developed under this case study was a speech data compression application, called Vocoder. This application consisted of a voice data encoder and decoder, and was to be used to compress voice data for transmission over a network. Input to the encoder was a WAVE [5] file. Output of the encoder was a compressed data file. The decoder was to perform a reversed process. It was to accept a compressed data file as an input and to restore the original WAVE file. The operation concept of the application is shown in Figure 1.

The compression algorithm used to implement Vocoder was Mixed Excitation Linear Prediction (MELP) [6]. The MELP algorithm is based on the traditional Linear Predictive Coding (LPC) but it can mimic more characteristics of natural human speech.

It was planned that the software development will be iterative and it will go through two major releases.

Release 2 was to add data analysis capability to the application by graphing frequency and magnitude of voice data.

3. Development Process Model Selection: Agile Process and Test Driven Development

The software development process model used for this case study was a form of the Agile model called Test Driven Development. Unlike the traditional waterfall model which assumes a predictive and staged approach to software development, agile methods are flexible and adaptive. Their measure of success is based on business value rather than on the level of conformity to a plan. In this paradigm, the software product evolves through many iterations and each iteration adds business value. After each release, customer feedback and requests for changes are evaluated and added to the next release along with new functionalities.

Test Driven Development (TDD) is an agile method in which tests are constructed before the production code is written and therefore they will originally fail due to the lack of the corresponding code. One of the main advantages of using this approach is that it results in
concise code, as tests can be viewed as the most precise technical descriptions of software requirements.

The product growth through TDD iterations is depicted in figure 2.

![Figure 2: TDD Paradigm in Agile Software Development](image)

The Development Iteration, as illustrated in Figure 2, is the smallest phase in which the software product is being designed, implemented and unit-tested. The activities in this cycle are listed below [7]:

1. Understand the assigned work items.
2. “Red” state: Create a test and make it fail.
3. “Green” state: Implement the production code to make the test pass.
4. Refactor: Modify the code to remove duplication.

At first, the developers study the assigned scenarios, quality-of-service (QoS) requirements or work items that need attention during the current iteration. The second step is referred to as a “Red” state. The developer writes a test and then creates just enough production code to make it compile. The test is then run and it is expected to fail due to the lack of the corresponding production code. The goal is to ensure that the test calls the correct method and that no redundant code exists. In the “Green” state, the developer implements the production code and verifies that the test would pass. Once the test passes, all tests created up to this iteration are executed to ensure that the new code does not introduce any regressions to the existing code. The last step in this small cycle is called “Refactor”. Once the production code for the test is implemented, the developer refactors it. The goal of refactoring is to remove duplications caused by the addition of the new code and to improve the overall design. Thus, the design may be altered and the code may be changed again as needed to improve the overall structure of the application. Before ending this step, all tests are rerun to ensure that they pass. Release 1 of Vocoder was to be fully implemented to exercise above practices of TDD.

Described above, general best practices fall under principle 2 of ADP.

4. ADP Infrastructure: People and Technology

4.1 Technology Infrastructure Required for Vocoder

In order to facilitate a disciplined and quantifiable approach to software development ADP defines a minimum technology infrastructure required for a project. In addition to the developers’ IDEs, the minimum infrastructure for Vocoder consisted of requirements management system, defect tracking system, version control system, automated build, and an automated reporting system. The Requirements Management system stores and tracks the status of detected defects such as who is responsible for resolving them and how quickly they are being resolved and closed. The Version Control system organizes source code as it evolves through multiple iterations. The Defect Tracking system traces the status of detected defects such as who is responsible for resolving them and how quickly they are being resolved and closed. The Version Control system organizes source code as it evolves through multiple iterations. The Automated Build compiles and builds the application nightly. The Automated Reporting system seamlessly collects data and produces reports about the project progress and product status throughout the product lifecycle.

4.2 People and their Roles

The people roles and their interactions with technology should be defined as the infrastructure is being built. We have defined the following roles: Project Manager, Architect, Developer, Tester and System Administrator. The Project Manager is responsible for delivering a quality product on time and within the budget. She assigns tasks to team members, reviews the reporting system reports regarding the project status and makes business decisions. Technical decisions are made by the Architect based on evaluating technical reports pertaining to product implementation. The Developer is expected to implement the Architect’s design of the product, follow coding standards, unit testing policies, and submit clean code to the shared version control system. The Tester is responsible for detecting of defects, storing them in the defect tracking system and notifying developers about the problem.

In this case study, the faculty advisors assumed the role of the Project Manager. The students acted as the Architect, Developer and Tester. The IT staff member in the Computer Science Department worked as the System Administrator.
4.3 Technology Selection: Microsoft Visual Studio Team System and Foundation Server (VSTS) - Microsoft

Visual Studio Team System is a team-oriented software development environment which is designed to facilitate tracking and visualization of an application development process. The tool suite is comprised of a set of Visual Studio Clients and a Visual Studio Team Foundation Server. The set of Visual Studio Team Clients includes the Team Edition for Software Architects, for Software Developers and for Software Testers. We decided to use VSTS as the project technology infrastructure because it supports C#, which was our language of choice for Vocoder implementation and because the team members were familiar with Microsoft’s IDE. The VSTS provides an integrated set of features equivalent to the ADP’s minimum required infrastructure. Work items are artifacts describing elements of assigned work and they are used to store and track both requirements and defects. The Version Control system and the Automated Build system are integral part of VSTS and there is a set of predefined reports such as remaining work items and defect density for project managers and architects. Installation and customization of VSTS/FS with multiple views for the project manager and for the architect/developer/tester was completed by the System Administrator.

With VSTS installed and the people roles defined, the principle 1 of ADP was met.

5. VOCODER Project Implementation

5.1 Project Initiation

The project initiation involved definition of the project structure and code check-in policies, as well as the selection of the project solution template together with the establishment of the shared folder for the test data.

Project structure: There are two primary approaches to organizing a VSTS Team Project: by Area or by Iteration. The Vocoder project was structured by Iteration because the degree of parallel development was insignificant.

Check-in Policies: The check-in policies require the team members to confirm to a predefined set of standards before a file or changeset (files containing the code under the development) is checked-in to the version control system. The Vocoder project requires the team members to associate a work item with the changeset and to execute an automated code analysis before checking in the changeset to the version control system.

Project Solution: The initial project solution template was created using the Distributed System Solutions template. This template project allows the team to perform architectural design and to generate the C# development project folder.

Test Data Folder: In order for all members of the team to execute tests on the same set of test data, the project needed to create a centralized test data repository and configure the tests to use the data from this repository.

Figure 3 illustrates the structure of the Vocoder project, organized by iterations (releases).

![Figure 3: Vocoder Project Structure](image)

5.2 Vision Statement, Personas and Scenarios

Once the project infrastructure was set up, the project vision document was used to define personas and scenarios. The requirement definition activity in an Agile process starts with Personas. Personas represent groups of users or customers. Once personas are defined, the application goals are established. These goals represent the basic functionalities that the software product is expected to accomplish. For the Vocoder, these goals, listed in the order of the implementation, were: Read a WAVE File, Extract Voice Data., Compress (Encode) Voice Data, Decompress (Decode) Voice Data, Extract Spectral Properties. A scenario is then established for each goal to represent a single, positive path that a user would take in order to achieve a specific goal. In addition to scenarios describing functionality of the application, the desirable quality attributes were added to the requirement set. All scenarios were entered to VSTS and linked with other relevant product artifacts such as design documents and changesets creating a comprehensive requirement traceability throughout the product lifecycle.
5.3 Design and Construction

The design took place after the scenarios to be implemented in the current iteration were selected. The first task in designing this application was to define its internal structure and its external interfaces. The Vocoder provided the encoding and decoding services for the Transceiver Application. Therefore, the Vocoder must have had a “provide” port for the Transceiver Application to connect to. Internally, the Vocoder was constructed by integrating several smaller components together. This Component-Based Software Development approach reduces overall development time by enabling easier application upgrades as additional features are implemented and new technology emerges. Specifically, the Vocoder was comprised of a user interface component and three algorithmic components as conceptually sketched in Figure 4.

![Figure 4. Conceptual sketch of Vocoder design.](image)

**TDD- Red**

Once the system and component diagrams were drawn, code construction started in the RED state of TDD by creating unit tests which failed in the absence of the corresponding code (Figure 5.)

![Figure 5: Scenario “Extract Voice Data” – Red state](image)

**TDD-Green**

The next step in the TDD cycle was to implement the code so the tests pass (Figure 6).

![Figure 6: Scenario “Extract Voice Data” - Green](image)

**TDD- Refactor**

Refactoring is the final stage in the TDD iteration cycle. In this stage, the working code is modified to remove redundancies and make it more efficient. After removing redundancies, it was observed that the Façade [8] pattern should be applied to the design in order to entirely decouple the user interface logic from the logic of the application. A new class, called WAVUtil, was added to the application design.

As the scenarios were being implemented and defects fixed, the infrastructure provided a comprehensive view of the project status. For example, the complete traceability of the product is summarized in Figure 7.

![Figure 7: Traceability of the Vocoder Project](image)

The Project Development Plan contains all work items. Work items and changesets conversely have links from one to the other. Changesets have a list of source files implemented for the work item and the builds contain information about changesets.

As scenarios are created and implemented they are linked with all the relevant artifacts. Consequently, upon project completion all scenarios can be traced to the corresponding code and the tests verifying their functionality.
6. Quantitative Management and Project Visibility

Throughout the Vocoder project, daily reports, based on seamlessly collected data were generated to provide measures of product quality and project progress, thus fulfilling the fourth principle of ADP. These reports not only provided project visibility but also helped in assessing project progress with respect to the upcoming completion deadlines.

For example, a defect rate report depicted in Figure 8 shows the number of unresolved defects in the current iteration of the product development. This measure should approach zero as the project nears completion.

Another graph, called “The Remaining Work Report” for this project is depicted in Figure 9. It is critical for the success of a project to plan the work as accurately as possible during project initiation. However, the initial upward trend in the number of planned work items, illustrated in this figure, shows that the Project Manager originally underestimated the amount of needed work. After the initial phase, the amount of remaining work for the project stabilized. During this period, the increase in the number of work items largely corresponds to the number of defects found during testing and a few unplanned tasks. It can be observed that the number of work items being closed or defects being resolved also flattens out at the end of the iteration. The temporary halt in the progress is due to the fact that most of the work items for Release 1 had been completed. The remaining work items are to be completed in the subsequent release, and according to the Project Development Plan, the development for the next release has not been started.

Related to the above report is the Unplanned Work graph shown in Figure 10. The Unplanned Work graph illustrates the number of work items being added during the current development iteration, which were not originally planned. This information should be used in future project planning as resources need to be reserved for unplanned activities.

Figure 11 depicts yet another measure of the project progress called the Project Velocity. Project Velocity reflects the rate of work items being closed. This parameter can be used by the project manager to plan the iteration and to estimate the cost of the project. It can be observed on Figure 11 that in the first half of the project, the rate of closing work items was very low. The project velocity was low due to several reasons. First of all, team members needed time to become familiar with VSTS. Similarly, TDD was a new development model for the team and it had a steep learning curve. Secondly, the system administrator had difficulty keeping the Team Foundation Server fully operational in its initial installation phase. As the team members became more familiar with VSTS and TDD, the project velocity increased.
As the product quality and project status reports were reviewed on a daily basis, the decisions regarding adjusting of work assignments and project scope were revised.

The Vocoder was a success despite the fact that approximately one third of the project effort was put into setting up the initial infrastructure. The ability to automate, track and measure project progress facilitated the decision process, which further led to project on-time completion.

7. Conclusion

The goal of the Vocoder case study was to illustrate that a systematic, disciplined and quantifiable approach to software engineering can be accomplished by applying the principles and practices of Automated Defect Prevention. The ADP infrastructure for the project, VSTS, which consisted of a set of integrated features that provide “requirements to tests” traceability, facilitated product visibility and seamlessly collected data to provide measures necessary for decision making, proved crucial to the project success. The upfront time and effort spent on establishing the infrastructure shown to be worthwhile as the infrastructure facilitated enforcement of the best software engineering practices and helped in providing measures necessary for decision making throughout the project lifecycle. While only selected measures and reports were illustrated in this paper, ADP defines a plethora of such measures that can be implemented in VSTS as well as other vendor’s technologies. More work needs to be done in order to refine ADP methodology and to classify ADP practices based on project scope and definition.

8. References


A modal logic for the CARE usability properties for Multimodal User Interfaces

Nadjet KAMEL, Sid Ahmed SELOUANI
University of Moncton, Shippagan Campus
218, Boul. J.-D. Gauthier, Shippagan E8S 1P6, Canada
Email : {nadjet,selouani}@umcs.ca

Habib HAMAM
University of Moncton and Canadian University of Dubai
Moncton, New Brunswick, Canada, E1A 3E9
Email : hamamb@umoncton.ca

Abstract—This paper proposes the use of model-checking technique to validate Multimodal User Interfaces (MUIs). It introduces a modal logic to express the CARE (Complementarity, Assignation, Redundancy and Equivalent) usability properties for MUIs : the LCARE modal logic. The syntax, the semantics and an axiom system for LCARE are defined. LCARE allows for the specification of the CARE usability properties that a MUI must satisfy. The model-checking technique is used to check whether a CARE property, expressed in LCARE, is satisfied by a MUI. For this purpose, the user multimodal interactions are modeled by a transition system and the CARE properties are expressed in LCARE formulae. A satisfiability relation is defined. It checks whether a LCARE formula is satisfied by the transition system modeling the multimodal interactions. LCARE allows for the specification of all the CARE usability properties. An illustration is given for the Matis application (Multimodal Airline Travel Information System) as a case study.

Logics, Model-checking, Multimodal User Interfaces, CARE usability properties

I. INTRODUCTION

Temporal logics are suitable for specifying properties that programs must satisfy. The satisfaction of these properties is usually checked by using a model-checking technique. This technique is algorithm-based and verifies program properties [1] [2]. It is suitable for systems that can be represented by a finite model. The advantage of such formal techniques is that they are based on mathematical models which make specifications non ambiguous. The model-checking technique is fully automaticable which helps designers to debug the specification of the systems under development. This method is successfully used for several systems such communication protocols, computer hardware and software. Recently, this technique was also applied to verify usability properties for interactive systems and particularly for Multimodal User Interfaces (MUI). MUIs allow humans to interact with systems through multiple modalities such as speech, facial expression, gestures and gaze. Properties related to user interfaces are called usability properties. These properties define user needs and show how much the interface is accepted by these users. If these properties are not satisfied by the interface, all the interactive system risk being rejected by users even if the core of the system is robust and reliable. In the case of MUIs, one of the important usability properties that a MUI must satisfy is the possibility for the user to continue interacting with the system in the context where one of the modalities becomes disabled for any reason. For example, in a noisy environment the use the speech is not recommended because there will be a lot of ambiguities. Thus, if the interface allows for the interaction with another modality to perform the same task, then the usability of the system is increased. Four usability properties for MUI were defined in reference [3]. There are the CARE properties (Complementarity, Assignation, redundancy and Equivalence).

The aim of this paper is to introduce a logic to handle the CARE usability properties for MUIs : LCARE. This logic expresses all of the CARE usability properties. We define the LCARE by giving its syntax, semantics and its satisfaction relation. The formulae are evaluated on transition systems model. The satisfaction relation expresses under which conditions a LCARE formula is satisfied by a transition system. A system of axioms for this logic is defined. To illustrate the specification and the verification of LCARE formulae, we use the Matis [4] (Multimodal Airline Travel Information System) application.
This paper is organized as follows: Section II presents the CARE usability properties as defined in [3], Section III presents an overview of related work and our approach, Section IV presents some notations used in this paper. Our formal model, on which LCARE formulae are interpreted, is presented in Section V. The LCARE logic is presented in Section VI. An illustration of a case study is given in Section VII, and finally, section VIII concludes this paper.

II. Usability properties in MUIs

Usability properties measure user performance and preference. They are related to the User Interfaces (UIs) and they are defined by the Human Computer Interaction (HCI) community. Many usability properties were defined for the UIs of interactive systems [5][6].

These usability properties do not cover the specific aspects of new interaction technologies such as multimodal interfaces. To address such aspects, The authors of [3] have identified a set of properties required to state that a multimodal interactive system is usable. The authors have defined Complementarity, Assignment, Redundancy and Equivalence, known as CARE, the main properties that may be satisfied by a multimodal system. The CARE properties characterize four types of relationships between states and modalities. The CARE properties are defined according to a set of modalities and two states of the characterized multimodal system. These states identify the initial and the final state of a user interactive task.

1) Complementarity. A set of modalities is defined to be complementary if they may be used to reach a state \( s' \) starting from a state \( s \). Moreover, a single modality shall not allow to reach a state \( s' \) starting from a state \( s \).

2) Assignment. A modality \( m \) is assigned to reach a state \( s' \) starting from a state \( s \), if \( m \) is the only modality that allows to reach \( s' \) from \( s \) and no other modality allows it.

3) Redundancy. A set of modalities are defined to be redundant if, they are equivalent and are used in parallel to reach a state \( s' \) starting from a state \( s \).

4) Equivalence. A set of modalities are defined as equivalent if each modality allows to reach a state \( s' \) starting from a state \( s \).

The equivalence property expresses the availability of choice between several modalities whereas the assignation expresses the absence of this choice. The redundancy and the complementarity properties express the combination of several modalities. The redundancy property requires that the used modalities must be equivalent.

III. Formal techniques for verifying usability properties

Popular temporal logics such as CTL[7] and ACTL[8] were used to specify usability properties for interactive systems. In [9], usability properties related to Human-Machine dialogue such as Deadlock freedom and task completeness were expressed in CTL temporal logic and checked by the SMV model-checker [10]. [11] and [12] verify also user interfaces with SMV (Symbolic Model Verifier) using CTL (Computational Tree Logic). In these work, interactors specifications are translated into SMV model checker input language and security properties are specified in CTL temporal logic. In [13], the authors use LOTOS to write interactors specifications, and analyses translated finite state machines using ACTL (Action based Temporal Logic). In [14], both the system and the properties are specified as processes expressed in Lotos process algebra [15]. A property is expressed as an undesired behavior. In this case, the satisfaction property verification consists of checking that the behavior of the system process is different from the behavior of the property process. The CADP model-checker [16] was used to implement this approach.

Few studies using formal techniques for the verification of MUIs have been achieved. The authors of [17] used Lotos [18] algebra to model the UI, the ACTL temporal logic to specify the usability properties and the Lite model checker tool to verifying these properties. This work does not consider CARE usability properties. In [19] some of the CARE usability properties are specified in CTL temporal logic and checked by using the SMV model-checker. The same properties are specified in LTL temporal logic and checked by using SPIN model-checker in [20]. Lustre-Lutess [21] was used in [22] and [23] to validate the CARE properties. The authors used the Lutess tool to validate a multimodal interactive system developed using the ICARE framework [24]. In this approach, the CARE properties are specified in Lustre, and then they are checked by using the Lutess tool. In [25], the authors use the B method [26] to verify the CARE usability properties which are described as invariants in the first order logic. This work uses the proof technique to check the satisfiability of the properties.

a) Proposed approach: We propose an approach which models the multimodal interactions by a transition system. For this, we define a formal model inspired from process algebras such Lotos [27]. The semantic of this model is based on transition systems. In this approach, the CARE usability properties are expressed through logic formulae. For this, we define the LCARE logic throught its syntax, its semantics and an axiom
system. The verification of the properties is processed by a procedure that implements the satisfaction relation between a logic formula and the transition system. We define the satisfaction relation \( \models \) which establishes if a LCARE formula is satisfied by a transition system corresponding to the multimodal interaction system. The procedure implementing this relation will be done in a future work.

IV. NOTATIONS, PRELIMINARIES

**Definition 1.** Formally, a labeled transition system is a tuple \( S = (Q, E, T, q_0) \) where
- \( Q \) is a countable set of states;
- \( E \) is a set of labels designing the actions of the system;
- \( T \) a transition relation \( \in Q \times E \times Q \);
- \( q_0 \in Q \) is the initial state of the system.

**Definition 2.** A transition \( t \) is a triplet \( t = (p, a, q) \) \((a \in E, p, q \in Q)\) that we note as \( p \overset{a}{\rightarrow} q \).

A path \( \sigma \) in the transition system \( S = (Q, E, T, q_0) \) is defined by a sequence of transitions \( (q_i, a_i, q'_i) \) for all \( i \). We also note this sequence as \( q_0 \overset{a_1}{\rightarrow} q_1 \overset{a_2}{\rightarrow} \cdots \overset{a_n}{\rightarrow} q_n \).

A behavior is a path in the transition system \( S \) that starts at the state \( q_0 \).

Let \( \sigma_i \) be a path that starts at the state \( q_i \) and \( T_i \) be the set of all the transitions of this path. We rename the initial state \( q_i \) of the path \( \sigma_i \) as \( q_{0,i} \).

**Definition 3.** The cartesian product \( \sigma_1 \otimes \sigma_2 \otimes \cdots \otimes \sigma_n \) is a transition system \( Sprod = (Q, E, T, q_0) \) where
- \( Q = Q_1 \times Q_2 \times \cdots \times Q_n \) where \( Q_i \) is the set of states in the path \( \sigma_i \);
- \( E = \prod_{i=1}^n (E_i \cup \{-\}) \) where \( E_i \) is the set of the labels in the path \( \sigma_i \);
- \( T = \{(q_1, \ldots, q_n)(a_1, \ldots, a_n)(q'_1, \ldots, q'_n)\mid \forall i \in 1..n, (a_i \neq -t \text{ or } q_i = q'_i \text{ or } (a_i \neq -t \text{ and } (q_i, q'_i) \in T_i))\}
- \( q_0 = (q_{0,1}, \ldots, q_{0,n}) \)

V. DESCRIPTION OF THE MULTIMODAL INTERACTION SYSTEM

The chosen representation for the multimodal interactions system is based on the theory of interactive systems and process algebra such as CCS [28].

### A. Syntax

The syntax of the model describing multimodal interactions is given by the following grammar issued from classical process algebra. The rule defining \( S \) generates the multimodal user interface at a higher level by composing interaction events. The rule \( E \) generates the multimodal interaction event using interaction actions of the set \( A \). Let \( a \) be an interactive action in the set \( A \) \((a \in A)\).

\[
S ::= S | S >> S | S || S | S | E
\]

\[
E ::= a; E | a || E | a | E | \delta
\]

The operators \( ; \) and \( >> \) stand for sequence and the operators \( [], || \), and \( | \) stand for choice, interleaving and parallel operators, respectively. The term \( \delta \) denotes the final state. No transition is performed in this state.

#### B. Semantics

For each syntactical term of the previous grammar, a transition system is defined to express its operational semantics. Let \( P \) and \( Q \) be two terms of the previous grammar and \( a, a_1 \) and \( a_2 \) be actions of \( A \), then the transition \( P \overset{a}{\rightarrow} Q \) expresses the state that the term \( Q \) is obtained from \( P \) when the action \( a \) is performed. Using, this notation for transitions, the operational semantics is formally expressed by transition rules expressing the behavior of each operator of the previously described grammar. Each rule of the form \( \text{premise} \rightarrow \text{conclusion} \) expresses that when the premises hold, then the conclusion holds. The formal semantics is given through the following set of rules.

1. **Stop.** \( \delta \) does not perform any transition;
2. **Prefix operator.** The term \( a; P \) performs the action \( a \) and then behaves like the term \( P \);
   \[ a; P \overset{a}{\rightarrow} P \]
3. **Sequence operator \( >> \).** If the term \( P \) performs an action \( a \) and then behaves like the term \( P' \) then, the term \( P >> Q \) performs the same action and then behaves like the term \( P' >> Q \). This is defined by the following rule:
   \[ P \overset{a}{\rightarrow} P' \text{ and } P' \neq \delta \]
   \[ P >> Q \overset{a}{\rightarrow} P' >> Q \]
   If the term \( P \) performs an action \( a \) and finishes then, the term \( P' >> Q \) performs the same action and then behaves like the term \( Q \). This is defined by the following rule:
   \[ P \overset{a}{\rightarrow} P' \text{ and } P' \neq \delta \]
   \[ P >> Q \overset{a}{\rightarrow} Q \]
4. **Choice operator \( [] \).** The first rule asserts that if the term \( P \) performs an action \( a \) and then behaves like the term \( P' \) then, the term \( P []; Q \) performs the same action and then behaves like the term \( P' []; Q \) (the second rule is defined in the reverse way).
   \[ P \overset{a}{\rightarrow} P' \text{ and } P' \neq \delta \]
   \[ P []; Q \overset{a}{\rightarrow} P' []; Q \]
5. **Interleaving operator \( || \).** It allows for the interleaving (asynchronous) of two transition systems (left and right). The first rule, asserts that if the
left system transits from the state identified by the term \( P \) to the state identified by the term \( P' \), by performing an action \( a_1 \), then the composed state \( P \parallel Q \) transits to \( P' \parallel Q \) by performing the same action.

\[
P^a_1 P' \\
P \parallel Q^a_1 P' \parallel Q
\]

In the same way, the second rule expresses, the behavior of the composed system resulting from the behavior of the right term \( Q \).

\[
Q^a_2 Q' \\
P \parallel Q^a_2 P \parallel Q'
\]

6) \textbf{Parallel operator} \( \parallel \). It allows for the running of two transition systems (left and right) in parallel (synchronous). The two first rules express the interleaving between actions.

\[
P^a_1 P' \\
Q^a_2 Q' \\
P \parallel Q^a_2 P \parallel Q'
\]

The third rule states that if the term \( P \) performs an action \( a_1 \) and behaves like the term \( P' \), and the term \( Q \) performs an action \( a_2 \) and behaves like the term \( Q' \) then \( P \parallel Q \) performs the two actions in one step and behaves like \( P' \parallel Q' \). We assume that the system, can not perform more than one action \( a_i \), produced by the same modality, at the same time. For this, we use the function \( \text{mod} \) that assigns, to each interactive action, the modality that it produces.

\[
P^a_1 P' \text{ and } Q^a_2 Q' \text{ and } \text{mod}(a_1) \neq \text{mod}(a_2) \\
P \parallel Q^a_1 P \parallel Q'
\]

The transition system corresponding to any term of the grammar is obtained by using the previous rules inductively.


VI. A MODAL LOGIC FOR CARE: LCARE

In this section, we define LCARE modal logic to specify CARE properties.

A. Syntax of LCARE

Let \( M \) be the set of all the modalities used by the system. \( m_i \) is an element of the set \( M \). We define four modal operators: \( \text{Equi} \), \( \text{Red} \), \( \text{Comp} \) and \( \text{Assig} \) to express, respectively, the equivalence, the redundancy, the complementarity of the modalities of the set \( M \) and the assignment of a modality \( m_i \) of the system. The operators \( \land \) (and), \( \lor \) (or) and \( \neg \) (not) are operators of propositional logic.

\[
\phi ::= \text{Equi}(M) \mid \text{Assig}(m_i) \mid \text{Red}(M) \mid \text{Comp}(M) \mid \\
\phi \land \phi \mid \phi \lor \phi \mid \neg \phi
\]

B. Semantics of LCARE

The LCARE formulae are interpreted on a transition system defined by \( S = (\langle Q, A, \rightarrow, q_0 \rangle) \). For each modality \( m_i \), we define \( A_{m_i} \) as the set of user interactive actions produced by this modality. \( A \) is the set of all the interactive actions. If there are \( n \) modalities in the system, then \( A = \sum_i A_{m_i} \). The function \( \text{Mod} \) assigns to each user interactive action \( a_i \) the modality with which it has been performed. If the user interactive action \( a_i \) is performed by the modality \( m_i \) then \( \text{Mod}(a_i) = m_i \). \( q \) and \( q' \) are two states of the transition system \( S \) and they correspond, respectively, to the initial and final state of a user task \( T \).

\[
S(q, q') \models \varphi \text{ means that the formula } \varphi \text{ holds along paths between the states } q \text{ and } q' \text{ in the transition system } S.
\]

For each syntactical operator, we define the satisfaction relation as follows:

1) \( S(q, q') \models \text{Equi}(M) \iff \forall m_i \in M, \exists q_0 (= q) \rightarrow q_1 \rightarrow a_1 \ldots \rightarrow a_n \rightarrow q' \) such that \( \forall a_j, 0 \leq j \leq n, \text{Mod}(a_j) = m_i \).

It asserts that the modalities of the set \( M \) are equivalent to reach the state \( q' \) starting from the state \( q \) if and only if, for each modality \( m_i \) of the set \( M \), there exists a path starting from the state \( q \) and reaching the state \( q' \), such that all the interactive actions performed on this path are performed by the modality \( m_i \). To note the states of the path \( q_0, q_1, q_2, \ldots \), we rename the state \( q \) by \( q_0 \).

2) \( S(q, q') \models \text{Assig}(m_i) \iff \forall \sigma = q_0 (= q) \rightarrow a_0 \rightarrow q_1 \rightarrow a_1 \ldots \rightarrow a_n \rightarrow q', \forall j, 0 \leq j \leq n, \text{Mod}(a_j) = m_i \).

It asserts that the modality \( m_i \) is assigned to reach the state \( q' \) starting from the state \( q \), if and only if all paths starting from the state \( q \) and reaching the state \( q' \) are performed by interactive actions produced by the modality \( m_i \).

3) \( S(q, q') \models \text{Comp}(M) \iff (\forall \sigma = q_0 (= q) \rightarrow a_0 \rightarrow q_1 \rightarrow a_1 \ldots \rightarrow q', \forall j, 0 \leq j \leq n, \text{Mod}(a_j) \in M \) and \( \forall m_j \in M, \exists j, 0 \leq j \leq n, \text{Mod}(a_j) = m_i \) and not(\( \exists m_i \) \( \in M \) such that \( \forall j, 0 \leq j \leq n \).
It asserts that the modalities of the set $M$ are complementary to reach the state $q'$ starting from the state $q$, if and only if in each path starting from $q$ and reaching $q'$ we have the following conditions:

- a) all actions performed on this path are produced by modalities of the set $M$;
- b) each modality of the set $M$ is used to perform the actions of this path;
- c) there is no modality in the set $M$ that allows reaching the state $q'$ starting from the state $q$.

4) $S(q, q') \models \text{Red}(M) \iff$

- a) $\forall m_i \in M, \exists \sigma_{i} = q_{0,i}(q) \xrightarrow{a_{0}} q_{1,i} \xrightarrow{a_{1}} \ldots \xrightarrow{a_{n}} q'$ such that $\forall j, 0 \leq j \leq n, Mod(a_j) = m_i$
- b) $\forall \sigma \in \sigma_{1} \otimes \sigma_{2} \otimes \ldots \otimes \sigma_n, \sigma = q_{0,i}(q) \rightarrow \ldots \rightarrow q'$
- c) $\otimes$ is the product operator between the paths defined in Section IV.

It asserts that the modalities in the set $M$ are used in a redundant way to reach the state $q'$ starting from the state $q$, if and only if

- a) each modality $m_i$ in the set $M$ allows reaching the state $q'$ starting from the state $q$. This expresses the equivalence of modalities of the set $M$.
- b) there exists a set of paths starting from $q$ and reaching $q'$ obtained by the Cartesian product of all the paths. Each path is performed through a modality $m_i$ in the set $M$.

5) $S(q, q') \models \phi_1 \land \phi_2 \iff$

It asserts that both the formula $\phi_1$ and $\phi_2$ holds on the system $S$ between the states $q$ and $q'$.

6) $S(q, q') \models \phi_1 \lor \phi_2 \iff$

It asserts that at least one of the formulas $\phi_1$ or $\phi_2$ holds on the system $S$ between the states $q$ and $q'$.

7) $S(q, q') \models \text{Not} \phi \iff \text{Not} S(q, q') \models \phi$

It asserts that the formula $\phi_1$ does not hold on the system $S$ between the states $q$ and $q'$.

We say that a formula $\phi$ of LCARE is satisfiable if there exists a model $S(Q, E, R, q_0), q$ and $q' \in Q$ such that $S(q, q') \models \phi$.

The formulae $\phi$ is said to be valid in a model $S(Q, E, R, q_0)$ if $\forall q, q' \in Q$, such that $\exists \sigma = q(= q_1) \rightarrow a_1 q_2 \rightarrow a_2 \ldots \rightarrow q'$ in the transition system $S$, $S(q, q') \models \phi$.

### C. The axiom system

In this section, an axiom system for LCARE language is introduced. It is denoted by $S_lCARE$. This system has the following schema of axioms and inference rules.

1) $\text{Assign}(m_i) \Rightarrow \forall M, \neg \text{Equi}(M)$
2) $\text{Assign}(m_i) \Rightarrow \forall M, \neg \text{Comp}(M)$
3) $\text{Assign}(m_i) \Rightarrow \forall M, \neg \text{Red}(M)$
4) $\text{Assign}(m_i) \Rightarrow \forall m_j, m_j \neq m_i, \neg \text{Assign}(m_j)$
5) $\text{Equi}(M_1) \land \text{Equi}(M_2) \Leftrightarrow \text{Equi}(M_1 \cup M_2)$
6) $\text{Comp}(M) \Rightarrow \neg \text{Equi}(M)$
7) $\text{Red}(M) \Rightarrow \text{Equi}(M)$

Informal reading of specific modal axioms can be illustrated in the following way:

- axioms 1, 2, 3 and 4 means that if a modality $m_i$ is assigned then no other modality is allowed.
- axiom 5 means that if the modalities of the sets $M_1$ are equivalent and the modalities of the set $M_2$ are equivalent, then the modalities of the set $(M_1 \cup M_2)$ are equivalent.
- axiom 6 means that if the modalities of the set $M$ are complementary, then they are not equivalent.
- axiom 7 means that if the modalities of a set $M$ are redundant, then they are equivalent.

The inference rules are here propositional inference rules.

b) Theorem 1.: System $S_lCARE$ is sound.

The soundness of $S_lCARE$ can be obtained by proving the validity of the axioms and taking into account the fact that inference rules are validity-preserving.

### VII. APPLICATION TO A CASE STUDY

To illustrate our proposal, a case study has been chosen. MATIS (Multimodal Airline Travel Information System) [4] is a multimodal application that allows a user to retrieve information about flight schedules using speech, direct manipulation with keyboard and mouse, or a combination of these modalities, supporting individual and synergistic use of multiple modalities. For each request, a window is created by the user order. It contains fields filled with values corresponding to the parameters of the request, such as the departure and arrival city names. The user can fill these fields by pronouncing words that are recognized by the system, clicking on the fields to set the focus, and on the city names list, to select the elements to be put in the selected fields.

MATIS (Multimodal Airline Travel Information System) [29] [4] is a multimodal application that allows a user to request a flight database. The user can use several modalities to interact with this application: speech, mouse or keyboard. To perform his tasks, the
user can use one or more modalities in sequence or in parallel. For each request, a window is created and a set of fields (From, To ...) must be filled in by introducing data needed by the system to give response to the request. The user can start two requests in parallel. Figure 1 shows the interface of Matis.

![Interface of Matis](image)

Fig. 1. Interface of Matis

A. The user multimodal interaction system

To illustrate our approach, we consider a part of the Matis user multimodal system. We consider the system as three tasks performed in sequence:

1) starting a request: This task consists on creating the request window by pronouncing 'Show me flights'.
2) filling the field From: This task consists of filling the text field From by the text Boston using either the speech or the mouse clicks. The user can pronounce 'From' and then 'Boston', or he can click on the text field From using the mouse and then click on the city Boston on an allowed list of cities.
3) filling the field Oslo: This task consists on filling the text field To by the text Oslo using the speech and the mouse clicks. The user can either pronounce 'To' and then click on the city Oslo or click on the text field To and then pronounce 'Oslo'.

The syntax of this system model is given through the following expression:

- ('Show me flights') $\implies$
- ('From' $\implies$ 'Boston') $\land$ (ClickFrom $\land$ ClickBoston)
- ('To' $\implies$ 'Oslo') $\land$ (ClickTo $\land$ ClickOslo)

Figure 2 presents the corresponding transition system $S$. It is obtained by using the semantic rules defined in Section V-B.

![Transition system of the multimodal interaction](image)

Fig. 2. Transition system of the multimodal interaction

Each of the three tasks starts, respectively, at states 1, 2 and 5, and each ends at states 2, 5 and 8.

B. Expressing usability properties in LCARE

In this section we use LCARE to express the CARE usability properties of the Matis system model presented above.

1) $S(1,2) \models Assign(speech)$ is evaluated to be true. Indeed, there exists only one path between the state 1 and the state 2 (1 $\implies$ $\text{show me flights'}$ $\implies$ 2) and $Mod(\text{show me flights'}) = speech$.

2) $S(2,5) \models Equi\{(speech,mouse)\}$ is evaluated to be true. Indeed, for each of the modalities speech and mouse, there exists a path starting from the state 2 to the state 5 performed by the concerned modality. For the modality speech, there exists the path 2 $\implies$ 3 $\implies$ 5 such as $Mod(\text{From'}) = speech$ and $Mod(\text{Boston'}) = speech$.

For the modality mouse, there exists the path 2 $\implies$ 3 $\implies$ 5 $\implies$ 8 such as $Mod(\text{ClickFrom}) = mouse$ and $Mod(\text{ClickBoston}) = mouse$.

3) $S(2,5) \models Comp\{(speech,mouse)\}$ is evaluated to be true. Indeed,

- for each of the two paths ($\sigma_1$ and $\sigma_2$) starting from the state 5 to the state 8:
  - $\sigma_1 = 5 \implies 6 \implies \text{ClickOslo} \implies 8$
  - $\sigma_2 = 5 \implies \text{ClickTo} \implies \text{Oslo'} \implies 8$

  $Mod(\text{To'}), Mod(\text{ClickOslo}), Mod(\text{ClickTo}), Mod(\text{Oslo'}) \in \{\text{speech}, \text{mouse}\}$.

- For each of the modalities speech and mouse, there exists an action performed by this modality: $Mod(\text{To'}) = speech$ and $Mod(\text{ClickTo}) = mouse$.

- not $\exists m_i \in \{\text{speech}, \text{mouse}\}$ such that $\forall j, 0 \leq j \leq n, Mod(a_j) = m_i$

VIII. Conclusion

In this paper, the LCARE modal logic for CARE usability properties has been introduced. The CARE usability properties can be expressed in LCARE modal logic to be checked on the user multimodal interactions system modeled by a transition system. In contrast to LTL, CTL and ACTL temporal logics, in which a part of CARE usability were expressed, LCARE expresses all the CARE properties. In this paper, the expressiveness of LCARE is not formally compared to these previous temporal logic. It will be the aim of our future work. We intend, also, to define a model-checking algorithm to implement the satisfaction relation of this logic.
REFERENCES


Agile Development: Do advantages outweigh short comings?

Olajemi Olagbegi and Hisham M. Haddad
CSIS Department, Kennesaw State University
Kennesaw, Georgia 30144, USA

Abstract: Agile development is increasingly gaining attention in the development community, and as a result, various case studies were conducted to assess its effectiveness and the benefits it promises. The key question being debated is whether the advantages of agile development outweigh its shortcomings. This work examines various case studies and their findings in an effort to construct a bigger picture of the effectiveness of agile development. This work also looks at specific characteristics such as scalability, productivity, training, fast deliveries, test case development, and code quality.

Keywords: Agile development, agile methods, agile case studies, agile development advantages.

1. Introduction

Agile development has become widespread in all spheres of software technology and is being practiced more often than a decade ago. Late 2000, Robert Martin of Object Mentor called a meeting to create an association to endorse lightweight proponent ideas. In 2001, the Agile Alliance was formed with 17 members (proponent of light-weight method supporters), and the agile manifesto was created (www.agilemanifesto.org) [1]. Since then, the agile manifesto has been utilized wholly or partly depending on a few factors including organization size, software project needs, stability of requirements, and project deadlines. Agile development is gaining support as its methodologies have been analyzed through various projects (supported by statistics and documentation) to be efficient and reliant provided the agile manifesto principles have been incorporated. On the other hand, opponents of agile development see agility as an ad hoc collection of principles [2]. This work investigates practical case studies of agile methods and compares such experiences. Based on the results, this work draws conclusions with respect to the experiences of using agile development. The paper is organized as follows: Section 2 highlights agile methods; Section 3 examines selected case studies and highlights their findings; Section 4 is discussion; and Section 5 is the conclusion.

2. Overview of Agile Methods

Agile development is characterized by small development teams, iterative lifecycles, and emphasis more on deliverables and milestones instead of documentation as in traditional development. Most software organizations deal with changing requirements, environments, and schedules therefore software engineers adopt agile methods. Certain factors are identified in most agile methods; changing requirements and incorporating requirement changes into software lifecycle and stressing communication among team members as opposed to any set of software process guidelines. Traditional plan-based approaches are challenged by these factors [2]. Thus, a number of agile methods have been proposed. Although agile methods originated from different continents, their applications and values have similar backgrounds and practitioners were autonomous in deriving the various approaches [3]. Thus, each method has at least one unique feature that distinguishes it from other methods.

1. Extreme Programming (XP):

XP was introduced by Kent Beck in his book, Extreme Programming Explained, in October 1999 with its main aim being reduction of software production cost. XP is the most widely practiced method in academic and industrial settings for both large and small projects. XP is tailored to meet the needs of small teams continually faced with changing requirements [4]. XP is governed by five values and twelve principles, and its key activities include planning, design, coding, and testing.

2. Feature Driven Development (FDD):

FDD was first introduced in the book, Java Modeling in Color with UML, by Peter Coad, Eric Lefebvre and Jeff De Luca in 1999. FDD describes five basic activities: Develop overall model, Build features list, Plan by feature, Design by feature, and Build by feature [11] [12]. Typical of agile methods, FDD focuses on short iterations that deliver fully functional software increments. FDD utilizes a model driven approach; models are developed to help visualize and implement each software increment [5]. FDD has five lifecycle phases; the first three focus on requirements and planning; while the last two focus on design and implementation.

3. Adaptive Software Development (ASD):

ASD evolved from rapid application development work by Jim Highsmith and Sam Bayer. ASD utilizes
three cycles: speculate, collaborate, and learning. Its characteristics include mission focus, feature based, iterative, and time/risk driven and change tolerant. ASD takes the place of the waterfall model as it offers incremental and iterative development, with constant prototyping. It presents solutions to large/complex projects [6, 11].

4. Agile Modeling (AM):

AM focuses on modeling practices and cultural principles. The goal is to minimize modeling and documentation [6]. Agile modeling seeks to develop software in a light weight manner using principles, practices, and methodologies that help improve the development process. The lifecycles includes: initial requirements envisioning, initial architecture envisioning, iteration modeling, model storming, reviews, and implementation. The first two iterations are planning and the last four iterations deal with implementation issues.

5. Crystal Family (CF):

CF is a collection of methods created by Alistair Cockburn and Jim Highsmith to allow developers achieve “maneuverability” if and when the need arises [5]. CF is open to selectiveness such that any method could be used depending on the needs of the project. CF is also flexible as it can be modified or molded if the need arises for project purposes.


DSDM was created by a consortium (www.dsdm.org) of vendors and was publicized in January 1995 [7]. DSDM incorporates a unique perspective as it provides a framework of controls and best practice for rapid application development. DSDM calls for allocating time and resources before identifying the overall functionality of the product. The DSDM consortium defined a process model (DSDM lifecycle) that includes three different iterative cycles preceded by two lifecycle activities namely feasibility and business study. Other three cycles include functional model iteration, design/build iteration, and implementation [5].

7. Scrum:

Scrum, developed by Jeff Sutherland and his team in the early ‘90s, is an experiential approach devised for supervising an unpredictable team environment. Scrum capitalizes on its flexibility to incorporate software development techniques and practices for the implementation process. The standards are utilized for steering development activities within a process that implements these framework activities: requirements, analysis, design, evolution and delivery. The work task(s) that take place in each scrum process framework is referred to as a sprint. The work performed within a sprint is usually identified and often revised by the scrum team. Scrum is also characterized by emphasis on project precedence, categorized work units, communication and user feedback [5, 6].

8. Mobile-D:

Mobile-D, fairly new agile methodology, is developed to address the issues facing mobile software development. Mobile-D has been tested and used by few people to assess its value and effectiveness. Mobile-D is based on XP (for development practices), Crystal methods (for scalability), and Rational Unified Process (for life-cycle coverage). A team of less than ten developers working at the same location aspiring to deliver in short iterations would be optimal for the Mobile-D approach.

9. Internet Speed Development (ISD):

ISD, probably the least known agile method, offers a descriptive, management-oriented framework for dealing with the problem of handling fast product releases. The framework consists of time drivers, quality dependencies, and process adjustments.

3. Case Studies

This section highlights investigated case studies in an effort to present realistic experiences with agile development. These studies are selected for their practical objectives and the methodologies being utilized. Conclusions derived from these experiences are used to address the key question of this work: whether the advantages of agile development outweigh its shortcomings. Brief description of each case study is given below.

1. Accounting Application:

Grewal and Maurer [8] used XP to develop an accounting application. The study utilized Scrum meetings and included more developers as the project progressed. All team members worked together and were not all at the same location for the duration of the project, which lasted for two and a half years. The case study comprised of a total of 32 sprints. The sprints were not delivered after every production partly because it was replacing other systems that have been in place for about twenty years and the sponsors did not want their users using two systems at the same time. The team started out with 8 to 10 developers, adding 2 to 3 developers at a time paired with more experienced ones. The turnover was very low and the team was highly motivated. The project was divided into three main streams: the GUI
team that was used by main teams as needed; the design re-factorimg team, and the clean up team to make the code more intuitive and manageable. Toward the end of the project, a performance team was formed to identify and improve performance bottlenecks. The authors found that agile methodologies work better when there’s limited turn over especially in large teams, motivation affects productivity, extreme programming and scrum can be scaled to forty plus developers, and pair programming (PP) was effective for training new developers and producing high quality code, scrum meetings for the whole team became unmanageable, offshore outsourcing did not work well for the team because of difference in time zones, code quality degenerated when PP halted, customer satisfaction was obtained as a result of high quality, test driven development prevented project chaos, project was challenging and rewarding for developers involved.

2. Pair Programming:

Hulkko and Abrahamsson [9] sought to answer three research questions: what is the current state of knowledge on PP, how is PP used in practical settings, and how does PP affect software quality. To answer these questions, the authors gathered existing empirical body of evidence on PP from 1998 to 2004, reported the context and findings of four case studies and discussed the results and their implications. The existing body of literature on PP showed that pairs are more productive than solo developers, produces better code quality products, reduces development time, produces code with fewer defects, useful for training new persons, more beneficial with code reviews, more enjoyable than solo programming, and lastly the cost-benefit ratio (quality versus effort) is unknown. The authors sought to validate these findings by conducting four different cases studies on their own. The developers were at the same location and worked six hour days, had no prior experience with PP (expect for the project manager for case study one, two and four), trained on PP practices, and means of data collection was held at the beginning of each project. All four case studies were carried out in six iterations. Case one was a product developed for internal use (involved 5th and 6th year Master’s students), cases two, three, and four were commercial products. Case two involved research scientists and cases three and four involved both practitioners and students. The Mobile-D method (based on XP and Scrum) was used in all case studies.

The authors observed the following: (1) PP effort percent was at the highest levels in the beginning of the project and at the system test phase, (2) no regular trends could be detected in the development of the productivity rates between the projects, (3) PP helps developers to learn in the beginning of the project, solve problems and think of ways to resolve complex tasks and find little mistakes from simple code, (4) PP does not increase adherence to coding standards as suggested in existing empirical literature, (5) PP produces more readable code of better quality, and (6) not certain if PP produces less defects than solo programming. In addition, two case studies were examined for defects: one case reflects that pair programmed code would produce less defects while the other reflects that solo programmed code would produce more defects. The authors conclude that PP does not present as extensive quality benefits as suggested in the literature and does not result in consistently productive results when compared to solo programming.

3. Stage-Gate Project Management Model:

Karlstorm and Runeson [10] studied three large software development organizations for the benefits of using agile methods from a stage-gate project management model perspective - A stage-gate model describes a work process from idea to delivered product. The model presents the stages of product development in a generic, abstract project life cycle suitable for high-level management. The goal was to investigate the transition process of the agile teams to the stage-gate model. The companies had these similarities: both were branches of large global companies with thousands of employees and world wide markets, and all used corporate stage-gate project management models for product development. On the other hand, the application domains, products, and markets were different.

XP was implemented for one organization and due to the propaganda surrounding XP, the organization failed to modify its stage-gate models for agile projects. However management provided opportunities for staff training. The XP team did excellently by delivering product releases before time and under budget. XP was also used in the second organization, but the team did receive recognition from management during project execution. However the project was successful; it was found that the product quality was higher in this part of the system than the other parts. Therefore the project management started considering agile methodologies for future projects. In the third organization, the project focused on product management in the stage-gate model. The authors stated that "Management had heard about agile product development from engineering and wanted to see which ideas could apply at a higher level in the company. The company is working on improving its product stage-gate process and is interested in agile methods as one source of inspiration. Despite significant enthusiasm for change from both software engineering and marketing, change is proving difficult to prioritize over day-to-day deadlines. Other issues complicating the Vodafone case include the large number of people involved in a change made at higher organizational levels and the geographic distances involved in distributed
development teams. As a result, the change initiative is moving slower than expected” [10]. The authors conclude that using XP in all three case studies was feasible but worked best if the agile team was ready to interface with the stage-gate model, the integration of agile software development with stage-gate project management helped to achieve cost management, product functionality, and timely delivery. The authors also stated that the problems that were identified were the usual problems encountered when introducing change, in this case a new method.

4. Costs-Benefit Analysis for Conventional and XP Development:

Muller and Padberg [11] performed a comparison of cost and benefit based on the study of two scenarios (one scenario with conventional development and the other with XP) using an economic model. The economic model used in this study is net present value. The net present value of a project is calculated by discounting back the asset (dollar returns of a project) value from the time of project completion (development time) to the time zero and deducing the development cost. To accomplish the purpose of this study, the authors compute the net present value of a hypothetical sample project for various project settings which include XP Speed Factor, XP Defect Factor and discount rate. The net present value of conventional development is compared to the net present value of XP to enable the authors make a concrete analysis.

The authors state their most important findings in using XP: (1) A manager should consider XP when the market pressure is strong and team developers are significantly more efficient (productivity and code quality) when working in pairs compared to when working alone, (2) Individual developers should be added to a project instead of using XP when the market pressure is moderate and the work can be easily handled by additional developers, (3) The stronger the market pressure, the smaller the speed advantage, the defect advantage and the number of pairs which are required for XP to break even with the conventional process, and (4) A manager should consider applying XP if that supports splitting the project into small releases better than a conventional process. In addition, the authors do not fail to mention that other projects will only differ from this sample project in terms of product size, asset value or productivity, the model used does not take into consideration project management issues (like high turn over, increased communication overhead, overtime work and refactoring) and lack of empirical data from industrial XP projects having a realistic size.

5. ASD Methodology:

Begel and Naggapan [12] conducted a study at Microsoft Corporation to understand how ASD methodology is used, the kind of acceptance, failures and successes they have. On the survey, the authors gained a 17% response rate. The survey questions were targeted to understand ASD usage, respondents’ demographics, penetration of ASD practices, and perceptions of ASD. Participants in the survey were fairly experienced and included software developers, test developers and managers which were spread across three continents. The survey was comprised of 46 questions which were divided into three sections; demographics, agile development and PP.

The following benefits of ASD were identified from the survey: quick releases, improved communication and coordination, flexibility of design, more reasonable process, increased quality, better customer focus, better morale and prioritization, increased productivity and testing first. An analysis was conducted on the ranking of the benefits and the authors deduced that using ASD methodologies provided better focus on customer needs, better prioritization of development efforts, improved productivity and more reliance on test driven development. Also the following problems were identified: ASD does not scale to larger projects, too many meetings, management buy-in, unfamiliar with agile, coordination with other teams, lose sight of big picture, culture, bad design, lack of schedule and dev/test integration is difficult. An Analysis was also conducted on the ranking of the problems with these observations: members complained of the inefficiency of these meetings, lack of good design; teams rely on design improvements later on, difficulty coordinating with other agile teams, too much legacy code, too much documentation, insufficient individuals could adapt to ASD. In addition, the authors discussed some threats to the validity of their findings which include: people practicing ASD would have been more likely to participate in the survey, no statistics of individual team sizes and the survey is based on one large organization. The main findings are as follows: around one-third of the respondents used ASD, ASD users had a positive opinion about it, ASD was used by co-located team members (located on the same floor), the scrum methodology was found to be the most popular, teams working on legacy systems used ASD, and ASD does not scale to large projects.

6. Issues and Tactics when Adopting Pair Programming:

Vanhanen, Lassenius and Mantyla [13] reported on the use of PP in an industrial setting over a two-year period. The main purpose of the study was to address the
problems associated with adopting PP and propose some solutions. The study was conducted using four repeated surveys, observing PP usage and discussing PP experiences with the developers in an organization. The first and second survey questions were tailored to get responses on developers’ attitudes toward PP. The results from the second survey indicated that developers were embracing more PP slowly. On the third survey, open questions were added to the questionnaire and the fourth survey included questions about a newly added PP room. Survey participation was voluntary and anonymous submission was employed. The authors concluded the study with the following key findings; the effects of PP were most helpful for learning, compliance of tasks according to schedule, team spirit, forming acquaintance with other developers, different quality aspects, regulation in following work practices, and work enjoyment. Also the use of PP doubled at the end compared to the start and developers reported wanting to use it more. The authors stated that the following tactics helped the adoption of PP such as the use of guidelines for PP, having a team leader for PP, adoption of PP voluntarily, creating a positive atmosphere for PP and dedicating a room for PP.

7. Experiences of Using Extreme Programming in an Agile Project:

Vanhanen and Kopri [14] conducted a case study on different experiences on adopting PP in an industrial project. The paper briefly highlights several case studies that adopted PP and groups the experiences according to adopting PP, pair formation, and PP sessions. The key accomplishments, developer attitudes and reactions to adopting PP, solo programming versus PP preferences experiences were documented. The main case study used for this project was the same one documented by Vanhanen, Lassenius, and Mantyla [13]. This case study’s experiences were also grouped according to pair formation and PP sessions. The authors found quality, knowledge transfer, effort and work enjoyment to be effects of PP. The authors concluded the following: learning PP took place easily and was easy to use for developers, adoption seemed to much easier than proposed in existing documentation, PP was better suited for difficult tasks, development efforts were lower for PP than for solo programming for complex tasks and vice versa for easy tasks, pair rotation may have increased knowledge transfer and productivity in the team, and PP increased discipline in practices like test driven development, coding standard and frequent investigation, PP was a contributing factor to low defect rate, the navigator seldom found defects during the programming meaning that some other aspect of PP, such as designing and test-driven development together, probably helped to avoid injecting defects, team spirit was very high, no developer was against PP and half of the developers preferred it to solo programming.

8. Benefits of XP:

Muller and Tichy [15], report on the experience of graduate students using XP in a university course to illustrate the benefits of XP. The study focuses on experience with PP, iterative planning, testing, refactoring, and scalability during the course. The main purpose of the course was to gather experience with XP from an indifferent point of view. All programming tasks were done in pairs. For the purpose of this study, it should be noted that documentation of features was recorded on index cards apart from source code, no software specification is provided instead unit test cases are developed, the design and testing was done together in small increments, design does not incorporate change scenarios, and lastly there were no formal technical reviews or inspections. The authors made the following conclusions about the study in general; PP was easily adopted, encouraged double loop learning and students enjoyed coding in pairs. The authors also observed that writing test cases before coding can be unrealistic, design in small increments is complicated, XP does not scale properly due to communication overhead and requires coaching before it can be fully adopted. However, the authors report that the experience was a positive one overall.

4. Discussion

The efficiency of agile methods is still somewhat debated in the software community. Many organizations have adopted them based on evidence gathered from existing case studies (empirical data) while others have adopted them as a working phenomenon (without in depth research). Neill [16] shares his thoughts on transitioning from traditional methods to agile methods. He advises management and developers to investigate the needs of their project(s) to determine what methods will work best before joining the pool of agile method users. Neill disproves of two assumptions generally made by developers and or management: late requirement changes to an XP project is cost effective (lacks empirical data) and good code alone (not true in cases with software updates) is sufficient documentation. Another issue of contention is the usefulness of agile methods; are the benefits more than the detriments? If so, is there enough empirical data to support those claims. In order to answer those questions, we analyzed the case studies in the previous section and identified benefits/detriments supported by empirical data in this section. Muller, Tichy, Begel and Naggapan all agree that writing test cases early was beneficial in their projects and XP does not scale to larger projects. Grewal, Maurer, Begel, Naggapan, Hulkko and Abrahamsson experienced
increased productivity using agile methods for their projects and furthermore, Grewal, Maurer, Hulkanko and Abrahamsson observed that PP was useful for training new developers, helping developers learn and producing better product and code quality. Karlstorm, Runeson, Begel and Naggapan all reported that agile methods do bring about quick releases and timely product delivery (time-to-market). Similarities in findings among selected case studies are illustrated in Table-1.

<table>
<thead>
<tr>
<th>Similar Findings</th>
<th>Supported by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalability: XP does not scale to larger projects</td>
<td>Muller &amp; Tichy and Begel &amp; Naggapan</td>
</tr>
<tr>
<td>Productivity: PP increased productivity</td>
<td>Grewal &amp; Maurer, Begel &amp; Naggapan, and Hulkanko &amp; Abrahamsson</td>
</tr>
<tr>
<td>Training: PP useful for training new developers</td>
<td>Grewal &amp; Maurer and Hulkanko &amp; Abrahamsson</td>
</tr>
<tr>
<td>Learning: PP helped developers learn</td>
<td>Grewal &amp; Maurer and Hulkanko &amp; Abrahamsson</td>
</tr>
<tr>
<td>Fast Deliverables: Agile methods bring about quick releases</td>
<td>Karlstorm &amp; Runeson and Begel &amp; Naggapan</td>
</tr>
<tr>
<td>Time-To-Market Release: Agile methods bring about timely product delivery</td>
<td>Karlstorm &amp; Runeson and Begel &amp; Naggapan</td>
</tr>
<tr>
<td>Test Case Development: Writing test cases early was beneficial</td>
<td>Muller &amp; Tichy and Begel &amp; Naggapan</td>
</tr>
</tbody>
</table>

However, differences in findings can be traced to some of the case studies examined for the purpose of this work. Muller & Tichy and Begel & Naggapan vs. Grewal & Maurer’s case studies reflect some differences in case study findings. Muller & Tichy and Begel & Naggapan found that agile methods and XP cannot be scaled to large projects due to communication overhead problems that could be generated in larger teams, difficulty in large team flexibility as regards architectural design and changes, large scrum meetings with inexperienced agile developers could be considered ineffective and apprehensions to products with large release cycles are also mentioned as a possible concern with respect to scaling. On the other hand, Grewal & Maurer found that XP and scrum can be scaled to more than forty developers depending on the team structure and team’s planning strategies. Muller & Tichy found that the XP team meetings will be effective for the next team because emails and other means of communication cannot replace formal team meetings while Begel & Naggapan found that the frequent meetings held by development teams were ineffective. Developers complained about the inefficiency of daily scrum meetings due to inexperienced agile developers, slow meetings leading to time wastage, and managers use meetings to micromanage developers’ work. Grewal & Maurer found that moving developers from one team to the other did not work for teams as learning specialized areas for another was difficult. On the other hand, Vanhanen & Korpi found that team swapping worked in their study. According to that study, team rotation was used as a medium for knowledge transfer. Hulkanko & Abrahamson found that PP did not increase adherence to coding standards because the analysis from their empirical study showed that the density of coding standard deviation was greater in PP than solo programming while Vanhanen & Kopri found that PP increased adherence to coding standards because the navigator was there to oversee the driver’s coding. Differences in findings among these case studies are represented in Table-2.

<table>
<thead>
<tr>
<th>Different Findings</th>
<th>Supported by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalability: ASD methodologies( including XP) cannot be scaled to larger projects</td>
<td>Muller &amp; Tichy and Begel &amp; Naggapan</td>
</tr>
<tr>
<td>Scalability: XP and Scrum can be scaled to larger projects</td>
<td>Grewal &amp; Maurer</td>
</tr>
<tr>
<td>Productivity: Team meetings effective</td>
<td>Muller &amp; Tichy</td>
</tr>
<tr>
<td>Productivity: Inefficiency of team meetings</td>
<td>Begel &amp; Naggapan</td>
</tr>
<tr>
<td>Productivity: Team swapping did not work</td>
<td>Grewal &amp; Maurer</td>
</tr>
<tr>
<td>Productivity: Team rotating may have increased knowledge transfer and productivity in team</td>
<td>Vanhanen &amp; Korpi</td>
</tr>
<tr>
<td>Quality of Code: PP doesn’t increase adherence to coding standards</td>
<td>Hulkanko &amp; Abrahamsson</td>
</tr>
<tr>
<td>Quality of Code: PP increased discipline in practices like test driven development, coding standard and frequent investigation</td>
<td>Vanhanen &amp; Kopri</td>
</tr>
</tbody>
</table>
5. Conclusion

In summary, do the advantages of agile development outweigh its shortcomings? The answer, based on our analysis of these case studies, is Yes and No. We conclude that one should take into consideration what factors create the right environment for effective agile development. The effectiveness of agile development is highly dependent upon the local development environment, available resources, technical skills of team members, and project stability and needs. We believe a combination of favorable characteristics of these factors has higher chance of leading to effective practice of agile development. The attitude and motivation of developers are other significant factors. In addition, we conclude that case studies addressing agile development, as reported in the literature, do not provide consistently productive results when compared to other development methods. In the scientific sense, we found these studies to be inconclusive, and therefore, further studies are needed to help answer this research question and to provide clear guidelines on how organizations adopting agile development can maximize its benefits.

6. References


SESSION

RELIABILITY

Chair(s)

TBA
Combined Code Understanding and Comment Understanding

Bradley L. Vinz and Letha H. Etzkorn
Computer Science Department, University of Alabama in Huntsville, Huntsville, Alabama, USA

Abstract — Program comprehension necessitates the process of extracting properties from a program in order to achieve a better discernment of the software system under study. In this paper, we present a combined code and comment/identifier program comprehension approach that provides twice the input of existing approaches — whereas the latter approaches customarily focus on either code or comments but not both. Our code understanding approach is a heuristic approach. It is based on the same knowledge-based inferencing engine employed by a mature program understanding engine that examines comment and identifiers. A direct consequence of exploiting the same inferencing engine is that our approach allows for a much deeper understanding of the source code than is possible using either code understanding or comment/identifier understanding separately. A subgoal of our approach is to determine whether comments in the software match the associated code. The use of the combined approach is illustrated on computer software.

Keywords: Program comprehension, code understanding, comment/identifier understanding, combined approaches, knowledge-based approaches, synergism.

1 Introduction

Program comprehension is any activity that extracts properties from a program so as to achieve a better understanding of the software. Early work on program comprehension focused primarily on analyzing the code itself. This often included comparing flow graphs to a plan library of known constructs. However, formal non-heuristic approaches to program comprehension [6][12] have been shown to be NP-hard [14]. Even the simpler problem of formally finding instances of a given pattern in source code [4] has been shown to be NP-hard [14]. For this reason, heuristic approaches acquired new importance.

More recently, a substantial amount of research in program comprehension has focused on applying information retrieval techniques to the identifier names (function names and variable names) and comments in computer software [3][9].

However, any approach based on comments alone has the potential problem that comments may not be kept up to date as the code is modified. Similarly, any approach based on code alone is totally ignoring the domain level (as opposed to code level) information that is present in comments [1]. Overall, it should be clear that any approach that analyzes only the code or, alternately, only the comments/identifiers is using only half the available information and is providing an incomplete view of the software.

Finally, it could be useful, where possible, that is, where comments are present, to determine whether the comments match the code. For example, if there is no location anywhere in a software package that the comments and code can be shown to match, one should be suspicious of the documentation quality — possibly there are very few comments or else the comments have not been updated along with the code. Additionally, there is also the flip side of the first argument, if the comments match the code everywhere, one should be suspicious that the abstraction level of the comments is far too low. For example, one might be seeing very uninformative comments like the following:

```
i++; // increment i
```

Instead of a (hopefully) better comment such as:

```
i++; // examine the next customer
```

In our research, we have extended Etzkorn’s PATRicia (Program Analysis Tool for Reuse) system, a mature NL-based program understanding engine which analyzes the semantic aspects of source code contained in comments and identifiers, to handle not only comments and identifiers but also source code constructs. In this paper, we describe how we use the same knowledge-base inferencing engine that was employed in the original PATRicia system to perform code understanding as well as comment understanding. We also discuss how this can be expanded to perform comment-to-code matching.

Specifically, our research focuses on the following goals:

1. A heuristic approach to code understanding.
2. A combined comment and code understanding approach that provides a more complete understanding of software than has been possible heretofore.
3. An automated analysis of whether the given comments actually match the associated code.
2 Program comprehension methods

As mentioned before, program comprehension approaches include code understanding approaches and comment/identifier understanding approaches. Most code understanding approaches date from the 1980s and 1990s. Comment understanding approaches were originally suggested by Biggerstaff et al. [1]. Starting in the late 1990s, comment and identifier understanding came to be treated as an information retrieval task [3][9]. Below we first discuss some formal approaches to code understanding and why several of these were NP-hard. Then we discuss some comment and identifier approaches, particularly the more recent information retrieval approaches.

2.1 Code understanding approaches

Code understanding approaches include Rich and Wills [12], Kozaczynski and Ning [6], and Harandi and Ning [3]. Tjortjis et al. [13] analyzed existing program understanding systems, and divided the types of program understanding systems into formal, rigorous, semi-formal, systematic, and ad hoc. They characterized the work of Rich and Wills, Harandi and Ning, and Kozaczynski and Ning as "rigorous."

2.1.1 Woods and Yang code understanding characterizations and their constraint-based approach

Woods and Yang [14] analyzed the complexity of various program understanding systems. They defined the Simple Program Understanding Problem (SPUP), which consisted of dividing the source code into a series of blocks, each represented as a graph, and comparing each block to a library of program plan templates represented as graphs. They then described how the SPUP is representative of the work of Kozaczynski and Ning [6] and Rich and Wills [12], and proved that SPUP is NP-hard by a reduction from the Subgraph Isomorphism Problem (SIP).

Woods and Yang then described another problem called the Simple Program Template Matching Problem (SPTMP). The SPTMP breaks a program into a collection of program blocks viewed in terms of a graph. Then a program template plan represents relationships between nodes, where the edges between two nodes occur when there is a data flow between the two nodes. This can be described as attempting to recognize typical program plans (clichés) and integrating these instances into a coherent global understanding. They then illustrate how the SPTMP is NP-hard by a reduction from SIP.

Finally, Woods and Yang discussed their own, heuristic, approach based on constraint satisfaction algorithms. In their approach, the legacy software was preprocessed into an intermediate form that contained data-flow and control-flow information as well as Abstract Syntax Tree (AST) information. Next, the source code was partitioned into locally cohesive code blocks. Finally, a Constraint Satisfaction Problem (CSP) was formulated consisting of the blocks and the constraints between the blocks. The constraints between the blocks were derived from the intermediate representation. The constraints were of two types: 1) structural constraints, derived from the legacy code and including scope, called/calling relations, etc., and 2) knowledge constraints, independent of the legacy code, that reside in the plan library. Knowledge constraints include parent/child relationships, generalizations/specialization relationships, and roles. A solution to the CSP problem consists of matching blocks to each program plan component in the plan library, while not violating any structural constraint (from the source code) or knowledge constraint (from the plan library).

Woods and Yang note that the heuristics of some other researchers can also map into the CSP framework. Various aspects of our code understanding approach can be viewed in this category.

2.1.2 Harandi and Ning code understanding approach

Harandi and Ning [3] represented programs as series of program events, organized in a hierarchy. At the source level, events represent language constructs (statements, declarations). Higher level events represent data structures such as stacks, trees, etc. Finally, the highest level represents events such as search and sort algorithms, mathematical computation, etc.

Low-level events are recognized, and then compared to an event plan library. New events are inferred from existing events using plan rules. The reasoning procedure in this method is less formal than some other graph-matching approaches.

Our code understanding approach is very similar to the Harandi and Ning approach. However, our event plan library is less restrictive and more heuristic than that of Harandi and Ning—in the Harandi and Ning approach all necessary low-level events must be present for a higher level event to be recognized. If we understand this process correctly, then we believe this is equivalent to the SPTMP as described by Woods and Yang. Our code understanding approach bypasses this problem by treating the relationship between events more as structural and/or knowledge constraints, similar to the Woods and Yang approach. Additionally, our code understanding approach has the major advantage that the storage format is implemented in the same knowledge-based inferencing engine as is the comment and identifier understanding of the PATRicia system, which is discussed below.
2.2 Comment and identifier approaches

Biggerstaff’s concept recognition prototype, DM-TAO/DESIRE, stored domain knowledge in the form of concepts within a semantic net. Natural Language (NL) tokens, identifiers and keywords from comments, were compared to low-level domain concepts. DM-TAO was partially automated with user selectable interest sets. Note that Tjortjis et al. [13] characterized DM-TAO as “systematic.”

In the mid-90s, Etzkorn et al. [2][3] used fairly extensive natural language, knowledge-based (KB) information extraction techniques in the PATRicia system to understand comments and identifiers. Etzkorn et al. [2][3] discussed comment and identifier based program comprehension in terms of information retrieval. We will discuss the PATRicia system in further detail below, as our research in this paper is an extension to the PATRicia system.

Later, other researchers used different information extraction techniques to analyze comments and identifiers based on Latent Semantic Indexing (LSI) [7][8]. LSI has both disadvantages and advantages compared to a knowledge-based natural language understanding system such as the PATRicia system. LSI is said to be “cheaper but less accurate” than knowledge-based NL approaches. However, LSI does not have the time-consuming overhead of building the knowledge-base.

In more recent work, Marcus et al. [9] decomposed source code into a set of documents (file, class, function, interface) and used comment and identifier understanding using latent semantic indexing to match user queries to documents. Poshvyanyk et al [10] probabilistically ranked events observed during program execution with latent semantic indexing of the source code to identify sections of code relative to a feature of interest. Hayes and Dekhtyar [5] investigated using Vector Space Information Retrieval (VSIR) and Probabilistic Information Retrieval (PIR) in traceability analysis.

2.3 The PATRicia system

The PATRicia system [2][3] employs fairly extensive knowledge-based, natural language understanding techniques to analyze comments and identifiers. In its part of speech processing phase, PATRicia uses a natural language parser to analyze the parts of speech. PATRicia employs heuristics based on prior studies of the grammar of comment sentences and common identifier formats to improve sentence parsing. In its semantic processing phase, PATRicia employs a knowledge-base in the form of a weighted, hierarchical semantic network consisting of conceptual graphs [3]. The knowledge-base inferencing engine is implemented in the CLIPS version 6.0 expert system shell. A form of spreading activation based on link weights is employed within the semantic net. Identified interface layer keywords, using weighted links and threshold values, “fire” concepts in individual conceptual graphs, which then fire other, higher level concepts.

A common conceptual graph example [3] is:

\[ \text{[CAT]} \rightarrow \text{(STAT)} \rightarrow \text{[SIT]} \rightarrow \text{(LOC)} \rightarrow \text{[MAT]} \]

Here, [CAT], [SIT], and [MAT] are concepts, and (STAT) and (LOC) are conceptual relations. This means a cat is in the state of sitting, and the location of sitting is a mat.

3 Research approach and the proof-of-concept system

The new code understander and the original comment/identifier understander employ the same knowledge-based inferencing engine. Only very minor changes to the inferencing engine, to distinguish between a code and a comment event, were necessary. A diagram of part of the knowledge-base required to recognize a bubble sort routine is provided in Figure 1 (the comment processing is simplified for illustration). BubbleSort is commonly used as a feasibility study for code understander systems [4].

Several different kinds of information are represented in this figure: 1) the main comment KB, including concepts and inferencing links, shown as solid rectangles and arrows, 2) the main code KB, including concepts and inferencing links, shown as solid ellipses and arrows, 3) the comment KB interface layer, shown as dashed rectangles and arrows, 4) the code KB interface layer, shown as dashed ellipses and arrows, 5) the conceptual relations between concepts, shown as a heavyweight dashed ellipse, and 6) constraints, shown as heavyweight arrows. Note that the “constraints” are actually implemented within the PATRicia system knowledge base as inference links, the only difference between a constraint and a typical inference link is its purpose. All threshold values are equal one for each concept. In this methodology, the conceptual graphs are not part of the inferencing; rather, they are used to define and describe individual concepts, and are used in the generation of certain types of reports. (They could be made more accurate by line number checking, but we currently have not implemented that feature.)

In this figure, it can be seen that the concept BubbleSort could be inferred two different ways. First, it could be inferred from the comment understander: the keyword bubble recognized as an adjective and the keyword sort recognized as a noun. (Again, this process is simplified for illustration.) Secondly, it could be recognized from the code understander. If a nested if occurs before an else which then occurs before a swap, this would be recognized as a BubbleSort. Compare this to the bubble sort code given in Figure 2. False positives are possible, since not every aspect of the code is examined (for example, \(x[i] > x[i+1]\) is not considered in this example, although it could be added).
/* This version of the bubble sort makes a fixed number of passes. */
for (int j=1; j < n; j++) {
    // count how many times bubble sort
    for (int i=0; i < n-j; i++) {
        if (x[i] > x[i+1]) {
            // exchange elements
            temp = x[i];
            x[i] = x[i+1];
            x[i+1] = temp;
        }
    }
}
in Figure 3. Note that this includes line numbers as well as the class location (BubbleSort) and the method location (DoBubbleSort). The line numbers are used as part of the matching process, as described earlier.

4 Results

The raw output report from the code understander is shown in Figure 4. In this report, a term node is a low-level code event, equivalent to a keyword in the comment understander interface layer; while a node is an internal node that has been “fired” (its value from the inference links was greater than the threshold value of the node).

This is the same kind of raw output report that is generated by the comment understander, except that the part of speech status of the term nodes is ignored. Note that in this run, we actually did not employ the constraint links. When constraint links are employed, this necessitates a full pass through the knowledge-based inferencing engine for each line of code. Although this is polynomial time and thus acceptable, C*n, where C is the time to process inferencing links and n is the number of inferencing links in the KB, the PATRicia system comment understanding engine was originally designed for an inferencing cycle once per class. Thus, to keep the comment understanding engine unchanged (so we can compare it to earlier comment-only results), we decided to inference only once per class. Although there certainly are potentially more false output possibilities, in limited domains these possibilities are still limited. To leave out the constraint links, we simply re-weighted the input links from the term nodes (code events) to have a weight of 1.0.

In Figure 5, the modified conceptual graph report is shown. This report shows every concept that has been fired (has a TRUE match) along with all of its associated conceptual relations. It also contains information (“^nlg_,” for example) about how each concept might best be treated in a separate natural language output report. Note that this report includes both the output of the comment understander and the code understander. There is a separate output report in natural language that is generated using this information; however, in this case that report was not generated, since it is only generated when at least two linked concepts have a TRUE match. For example, if the comment understander had found a match for the increase concept, a natural language description of the bubble sort conceptual graph would have been generated.

![Figure 4. Code Understannder Output Report](image)

![Figure 5. Conceptual Graph Report](image)

The other important output report is the Comment-to-Code Match Report. This report is generated when the same concepts are inferred from both the code understander and the comment understander. The output of the Comment-to-Code Match Report is shown in Figure 6.

![Figure 6. Comment-to-Code Match Report](image)
This is a very simple example. What has occurred is the comment understander has detected the two occurrences of the words bubble and sort, and the related fact that a bubble sort routine is present. Similarly, the code understander has detected the single occurrence of the bubble sort code. Then the comment-to-code matcher has detected that the bubble sort concept has been inferred both from the code understander and the comment understander at the same location, and thus that this comment does indeed match the code.

5 Conclusions and future research

This initial feasibility study indicates that this combined code and comment approach has potential. The primary need is for a study on a larger code example to give some indication of the scalability of this approach. We are currently working on expanding this to a larger package.

In the future, it might be useful to provide more informative comment-to-code match reports that indicate whether there are sufficient comments matching code, too few comments matching code, etc. However, this requires comment-to-code comparison studies to determine what is sufficient.

It would also be interesting to perform concept location, finding where a particular concept of interest is located [9], examining the code understander output as well as the comment understander output.

6 References


A Framework and Associated Models for Determining Change Impact Analysis During Utility Service Provisioning in a Grid Environment

Ekabua, Obeten O. and Adigun, Matthew O.
Centre of Excellence for Mobile e-Services for Development, Department of Computer Science, University of Zululand, South Africa

Abstract
Continuous monitoring of changes to utility services and products in a distributed information system is an interesting issue in software engineering. Evolution of Software moved from functions, to modules, to objects, to components and now to services. There exist a high probability that eventually most software capabilities will be delivered and consumed as services. Change is an inevitable basic operation of software evolution. Changing services makes available useful information on the quality of services provided. But we realise that a change can propagate and this can be resolved by further additional changes. These additional changes create inconsistencies, which require further changes until no propagation is left. We regard a propagating change as a ripple effect on service provisioning. This research paper therefore present a framework together with two impact analysis models to support change automation in a utility service provisioning grid environment. As an inclusion for background information, we discuss change impact analysis, measurement and maintenance that form the basis for this framework.

Keywords: change impact analysis, service provisioning, service maintenance, service oriented architecture, grid services.

1. Introduction
Economy grid architectures are identified as providing fluctuating resources, hence applications that should run in such environments must account for occurable changes. These fluctuating resources range from processing elements, storage and network resulting from interconnection of parallel machines, clusters or even workstations. Amongst other properties of these resources are their changing characteristics during application execution [1]. Constructing individual, isolated and dynamic applications that can operate in a distributed environment where resources must be shared, offers a greater challenge. Design issues of diversity, adaptation, data locality, process locality and control locality must be of paramount interest [2].

Changing services makes available useful information on the quality of services provided. Change Impact Analysis (CIA) shows what impact or effect a change to a service or service provider will have on the system. It determines the scope of the change and provides a measure of the service’s complexity. It can be used during service maintenance to keep the system at a high level of quality, avoiding degradation of the system or during development to ensure that quality of the system is maintained throughout the development process.

Computing systems’ evolution (hardware and software) can be traced to changes in the original requirements, different hardware platform adoption and efficiency improvement. Maintenance management approaches indicate different possible changes during the maintenance process and this is seen as an indication of evolutionary changes. As a result of the complexity involve, error probability becomes high and some of these errors could result into undeterministic consequences such as loss of life, money, time and damage to the environment. This makes system evolution management an important phase in system development and maintenance. Hence, a maintainer faces the challenge of how to respond rapidly, correctly and efficiently to change. This is because the maintainer in most cases is not directly involve in the system development, as responding to change requires system understanding and change
identification before performing the change [3]. The use of formal methods is crucial as it enhances system understanding to the point of unfolding undetected propagating changes [4].

A combination of distributed object computing, component based computing and web-based concepts into what is now known as Service Oriented Architectures has emerged as an approach for developing dynamic and heterogenous service provisioning environments. This technology evolution, combine with this web revolution, poses new challenges in the context of service provisioning. With the massive diffusion of the internet as a distributed environment for service provisioning, people’s interaction with computers has dramatically change. People relate not to their own computer but rather to their point of presence within the service provisioning environment [5, 6, 7].

Service related software applications play important roles in human lives, therefore, products that affect people’s lives must have quality attributes. Good quality software is required and in order to determine the quality of software, we need methods to measure it. A key point to emphasis is that the quality of a service product may change over time and web service related applications are no exception. In the early days of computing, software costs represented a small percentage of the overall cost of a computer-based system. Hence, a sizeable error in estimates of software cost had relatively little impact. But, today software is the most expensive element in many computer-based systems. Therefore steps taken to reduce the cost of software can make the difference between the profit and loss of a company. So by determining the quality attributes of software, more precise, predictable and repeatable control over the software development process and product will be achieved [8].

2. Background Information

Since software capabilities are being delivered and consumed as services, to improve the quality of software during development, we need models of the development process and within the process we need to select and deploy specific methods and approaches and employ proper tools and technologies. We need measures of the characteristics and quality parameters of the software development process and its stages. We need metrics and quality models to help ensure that the development process is under control to meet the quality objective of the product. Data and measurements are the most basic prerequisites for the improvement and maturity of any scientific or engineering discipline. Yet, in the discipline of software engineering, this area is perhaps one that has many critical problems and one that needs concerted effort for improvement.

The use of measurements, metrics and models in software development assumes the availability of good data. In fact, the poor quality of data is a large obstacle in quality improvement. To enhance data accuracy, a good tracking system for the entire development process must be in place and the system must address the data validation issue. Measurements for software projects, therefore, should be well thought out before being used. Metrics that are arbitrarily established could be harmful to the quality improvement effort of a company and there are numerous examples of this in industry. Each metric used should be subjected to an examination of the basic principles of measurement scale, the operational definition, validity and reliability issues should be well thought out [9].

As software industry is rapidly moving towards maturity, resources have shifted from being devoted to developing new software systems to making modifications to evolving software systems: software maintenance [10]. A major problem for developers in a changing environment is that small changes can propagate through software to cause major unintended impacts elsewhere. Therefore, software developers need mechanisms to understand how a change to a software system will impact the rest of the system. This process is called CIA. Making software changes without understanding their effects can obviously leads to unreliable software products. CIA can be used to reduce the amount of maintenance required and thereby increasing the reliability of the software, since fewer errors would have been introduced.

An impact is the effect or impression of one thing on another. Impact can be thought of as the consequence of a change. Impact Analysis (IA) is used to determine the scope of a change request as the basis for accurate resource planning and scheduling and to conform to cost/benefit justification. CIA estimates what would be impacted during service provisioning and related documentation, if proposed service change (sometimes due to a fault or maintenance) is made [11, 12]. CIA information can be used for planning changes, making changes and tracing through the effects of changes. Research into CIA has been concerned mostly with procedural software: function-based programs not serviced-based. SOA is all about services that are found in the real world. CIA is the task through which service providers can assess the extent that a change to a single service will have on the rest of the services. It determines the scope of a change and provides a measure of the service’s complexity [13]. CIA can be achieved by directly finding out the challenges encountered by consumers while receiving services from their clients – service providers.
3. Grid and Agent

The core unifying concept that underlies Grids and Agents systems is that of a service. A service is an entity that provides a capability to a client through a well defined message exchange [14] or a service is a vehicle by which consumer’s need is satisfied according to a negotiated contract, which includes service level agreement and the function offered. In 3rd Generation Grids, all entities are services since service interactions are achieved through web service mechanisms. Although, while every agent is considered a service, not all grid services are necessarily agents. Therefore, the autonomous action notion is a function of how agents and grids interoperate.

The main objective of Grids is that of resource sharing and coordinated problem solving in dynamic, multi-institutional virtual organizations [15]. Grid therefore provides an infrastructure for federated resource sharing across trust domains. Grid primary concern has been the mechanism by which communities form and operate. Thus, grid effort is devoted to how community standards are represented via explicit policy and enforcement and how actions and commitments by community members are specified, monitored and enforced through implementation.

Agents are autonomous identifiable problem solving entities with well-defined boundaries and interfaces. They are situated in a particular environment, designed to fulfil a specific role and capable of exhibiting a flexible problem solving behaviour in pursuit of their design objectives. They need to be both reactive (reacting to changes in their environment on time) and proactive (taking initiatives) [16]. Agent and Grid systems consist of dynamic and stateful services. Since it is possible for new services to be created and destroyed over the system lifetime, the underlying service model for agents and grids is dynamic [17].

4. Change Propagation Framework

Changes are endemic to software artefacts and the services provided by these artefacts. When a change is effected in a particular service connected to grid, it is often difficult to determine the propagation of this service changes. We therefore present a change propagation framework shown in Figure 1 to support change automation in any grid engineering methodology.

![Figure 1. Generic Change Propagation Framework](image)

The Service Detector Engine (SDE) contains all consumer made available set of grid services \( (s_1, s_2, \ldots, s_n) \) under utilization. SDE liaises with Business Service Bus (BSB), a concept developed by Component Based Development and Integration (CBDI) [8] and incorporated into our framework to form Service Architecture (SA) responsible for providing a bridge between the implementation and the consuming application, creating a logical view of a set of services, which are available for use and invoke by a common interface and management architecture. The Activity Checker (AC) is responsible for the specification of the constraints that a well-formed service design should satisfy in order to check whether the application’s design is in conformance to the main host design. Violation of the rules governing the activity checker will trigger a constraint violation event from the Constraint Activator (CA) to be returned to the Change Propagation Mechanism (CPM). This informs the Service Repairer (SR) of a triggered event calling for a way of fixing the violated constraint by performing actions, which change the application’s design and keeps record of the ripple effect. The mechanism Validator (V) is responsible for checking the consistency of the change to the design (through the AC), which can result in further actions.

There are three major architectural perspectives for SOA namely: Application Architecture, Service Architecture and Component Architecture and our framework has these incorporated into it. The architecture has two perspective views: Consumer and Provider. The salient aspect of the architecture is that the consumer of a service should not be interested in implementation detail of a service, but the service provided. This is because the implementation architecture could vary from provider to provider, but still deliver the same service. Additionally, the provider should not be interested in the application that the service is consumed in, because new unforeseen application will reuse the same set of services. The consumer’s main interest is in the
application architecture and the services used, but not in the detail of the component architecture. The interest is in some level of details in the general business objects that are of mutual interest, for example, provider and consumer need to share a view of what is a subscription. But the consumer does not need to know how the service component and database are implemented. Also, the provider is focused on the component architecture and the service architecture, but not on the application architecture. Again, they both need to understand certain information about the basic application inorder to be able to set any sequencing rules including pre and post conditions.

SOA provides the need to be able to manage services as first order deliverables. The communication key between the provider and the consumer is service. There is the need therefore, for a service provisioning architecture in the form of generic framework, that will ensure that services are not reduced to the status of interfaces, but have an identity of their own and can be managed individually and in sets. BSB as shown in our framework is incorporated to meet this requirement by providing a logical view of the available services for any business domain. BSB answers such questions as what services do I need?, what services are available to me?, what alternative services are available?, what services will operate together and what services are connected to me [19]. Our framework is generic because it can be applied to a general service provisioning engineering methodologies to enhance monitoring change propagation. The most important component of the framework is the Change Propagation Mechanism (CPM), which is represented and implemented within the service provisioning architecture and the component architecture. CPM detects any change service due to the triggering effect generated and validated. CPM notifies the SR of the ripple effect for immediate action of fixing the service.

5. Framework Implication

Maintenance has been recognized as the most costly phase in the software life cycle [20]. Since software has been consumed as services, service maintenance effort has been estimated to be frequently more than 50% of the total life cycle cost [21]. This work has the potential to improve service provisioning to customers, thereby cutting cost during service delivery. Using change propagation framework will help to achieve the following:

- Understand the nature of the services needed by a consumer.
- Estimate the effort devoted to a project.
- Determine the quality of service.
- Predict the maintainability of service with respect to the derived benefits.

- Validate best practices for service providers in a frequent changing requirement community.
- Provide optimal maintenance solutions.

By identifying potential impacts before making a change, the risks associated with embarking on a costly change can be reduced, because the cost of unexpected problems generally increases with the lateness of their discovery. The more a particular change causes other changes, the higher the cost. Carrying out CIA will allow an assessment of the cost of the change and help management to choose between alternative changes. It will also allow managers and engineers to evaluate the appropriateness of a proposed modification. If a proposed change has the possibility of impacting large, disjoint sections of a service, the change will need to be re-examined to determine whether a safer change is possible [22].

Maintenance is costly, difficult and is not always clear what the impact of any type of change to service will have across the whole services. CIA shows the maintainer what the effect of any change will be on the system. This proposed generic framework offers the potential to improve the stability and efficiency of SOA and cut the cost of maintenance.

6. Change Impact Analysis Factor Adaptation Model (CIAFAM)

When a change of service is considered in a system, it is worthy to identify system components that may be impacted after such a change. This would enhance the system to keep running perfectly after a change implementation. A system absorbs a change easily if the impacted components is of a small number. One effective method to account for changes in services is to perform CIA and our framework is accessed by the impact model described. Our main concern is pivoted on how the system reacts to changes that lead to propagation.

For a given change K in a service P, we can describe a set of impacted services as a boolean expression. The Impact Analysis Factor (IAF) for such hypothetical change can be given by [20, 22]:

\[ IAF(K, P) = \Lambda^*(\sim \rho) + \Lambda \]

Where \( \Lambda \), \( \Lambda^* \), \( \sim \) denotes the usual boolean operators: conjunction, disjunction and negation respectively. K = a given change, P = a given service, \( \Lambda \) = there is an association between K and P, \( \rho = K \) is derived from the change service, \( \Lambda^* \) = there is an occurrence of aggregation link between K and P and IAF = Impact Analysis Factor.

This expression implies that a service in association (\( \Lambda \)) with K and not derived (\( \sim \rho \)) from the change service K or a service that is in aggregation link (\( \Lambda^* \)) with K is impacted. It is important to state that this
impact model only predicts, which services would be impacted if a change was really made. If a service is really impacted, it means there is the propensity of propagation in which case the IAF becomes 1. We concentrate on changes that have a synthetic impact, therefore, appropriate measures are based on impact that are dependent on the static nature of the provisioning system. This implies that impacts have a likelihood of propagation [20, 21].

7. Fault and Failure Assumption Model.

Depending on the architectural level, time phased and other specific service parameters, SOA failure modes may change. In modern SOA, common failures are due to unavailable infrastructure, client crash, service failure, server crash, session failure and component failure. Therefore, a generic failure \( F_k \) is define as:

\[
F_k = f(al, t_p, ss_p)
\]

Where:
- \( al \) = the architectural level of the faulty components
- \( t_p \) = the time phase during which the fault occur
- \( ss_p \) = the set of specific service parameters identifying the state of the particular service involved in the failure.

If each failure \( F_k \) is identified, the system failure modes can be represented as:

\[
F = F_1 F_2 \ldots F_n = \sum_{k=0}^{k=n} F_k
\]

Meaning that the system fails if at least one of the identified failure occurs. Our failure is recorded as a boolean value (0,1) with respect to \( t_p \).

Increasing redundancy degree may lead to increase in possible sources of failure resulting to a potential decrease in dependability [3]. To understand the impact of redundancy, dependability and interoperability on our framework, the failure model is necessary. Our fault assumption is based on a fail-silent assumption where either a service is actively operating or does not answer at all. This assumption is justified on the basis of our CIAFAM whose IAF is a boolean (0,1). When the value is 1, it indicates a fault (requiring change), but when the value is 0, it is in its active state.

To analyze the error type that a faulty service may induce in a grid environment, we formulate the concept of failure that will enhance change prediction as:

\[
F = f(al, t_p, t_m, i, d)
\]

where \( al \) and \( t_p \) are as previously defined and \( ss_p = t_m, i, d \) where \( t_m = \text{time (in months) when a fault is detected} \)

\[ t_m = x, y \quad \{0 \leq x \leq 6; 6 \leq y \leq 12\} \text{months} \]

\[ i = \text{the particular service item involve in failure} \]

\[ i \in I = \{a, b, c, \ldots, z\} \text{ where I is the set of available services} \]

\[ d = \text{the descriptor of the faulty session} \]

8. Conclusion

Impact analysis needs to be adapted to the type of systems that becomes increasingly common today, such as grid-based applications and publish-subscribed systems. The fact that repositories can be shared amongst several distinct systems introduces interoperability dependencies that impact analysis strategies especially tailored for these technologies must automate the effect of changes in a service provisioning environment. The research paper has therefore, provided a framework and two models that can effectively assist maintainers for a proactive approach to service maintainance during service provisioning.

References


Application of Software Fault Tree Analysis to an Airport Ground Control System

M. Towhidnejad1, L. Shen2, T. Hilburn1
1 Computer and Software Engineering Department, Embry-Riddle Aeronautical University
Daytona Beach, FL USA
{towhid or Hilburn}@erau.edu
2 Avidyne Corp.
Melbourne, FL USA
lshen@avedyne.com

Abstract

With the ever increasing role of software in our day-to-day life, and even more important the criticality of their role in safety and reliability of the applications, it is important to find techniques that help with identifying and balancing risks. One such technique is Fault Tree Analysis, which has proven to be powerful in other domains such as systems and processes. This paper, describes the results of application of the fault tree analysis to software requirement specification for an airport ground control system. In addition, this paper proposes a set of guidelines on how to apply SFTA to software requirement specification.

Keywords

Software Quality Assurance, Software Reliability, Software Requirement Specification, Fault Tree Analysis, Software Fault Tree Analysis

1. Introduction

The complexity of safety (mission) critical software systems has grown over time and given the important role that software presents in our society, which is likely to expand in the future. As these systems grow in complexity, it is necessary to ensure that they are correct, useful, and most importantly, safe and reliable for its operators and users.

Safety critical software systems refer to those whose failure may cause catastrophic consequences, incurring substantial human and material loss. Various techniques are available and used in order to assure the safety and reliability of the system. Some of the techniques include: Hazard and Operability Analysis (HAZOP) [1], used to analyze hazards of a system; Fault Tree Analysis (FTA) [2], used to analyze the causes of a hazard; and Failure Mode and Effect Analysis (FMEA) [3] used to verify the correct functioning of the system. All these techniques permit one to analyze the safety level of components and artifacts generated during the life of a product. Majority of the above mentioned techniques are applied to the system as a whole, and in most cases, the software components of the system are typically overlooked, or are barely analyzed. In this paper, Fault Tree Analysis (FTA) and its application to software, that is the SFTA, will be described and illustrated through application to an actual software system.

During the development of a software product and throughout the software development life cycle, different artifacts are generated, which represents the product under development. Industry data has shown that removing defects from a system is easier and cheaper during earlier phases of the life cycle (e.g. requirements and design) than latter ones. For example, artifacts generated during the phases of requirements and design, such as a Software Requirements Specification (SRS) and a Software Design Specification (SDS), respectively, carry a heavy weight for the correct and safe construction of the software system. Any safety defects present in the SRS will be propagated to the SDS; this in turn, will affect the overall construction and safety level of the system and its components.

Although it is almost impossible to detect and remove all the defects from a complex system, it is feasible to apply the concepts offered by the Software Fault Tree Analysis (SFTA) to find safety related problems in selected artifacts from a specific project. Previously, number of researchers attempted to use SFTA in order to detect defects in requirement document [4] or design document [5]. In most cases, the applications that have been used by these researchers are limited in size and complexity. In this research, we chose a complex system of hardware and software to apply fault tree analysis. One of the contributions of this work is to identify some of the issues associated with the application of the SFT to a system of hardware and
software. Another contribution is to propose a series of steps that will help the analyst in performing SFTA on a complex system. For the purpose of this project, we applied FTA on selected requirements from the SRS documents for the Airport Ground Control System (AGCS).

Normally, the SRS quality is ensured through traditional verification and validation methods. However, the safety level of this document is not normally directly assessed or considered during its creation. In a safety critical system such as the AGCS, it is important to ensure a high safety level, since it is deals with the management and routing of vehicles carrying passengers. If during the creation of the software system unsafe requirements were not removed, they may infect other phases and cause the system to fail. Depending on the severity of the failure, these unsafe requirements may put the system in hazardous states which can lead to catastrophic consequences, with possible material or human loss.

The remainder of this paper discusses the results of the application of SFTA to the AGCS’s requirement specification. In doing so, we will first briefly describe the AGCS and SFTA. This is followed by a discussion of some of the results of this experience, and a recommendation for the future work.

2. Airport Ground Control System (AGCS)

The AGCS was developed to simulate movement of vehicles on an airport surface. The simulation includes various aircraft and ground vehicles interacting with each other on the airport surface in accordance with Federal Aviation Administration separation rules. The AGCS also incorporate a hardware robot, G-Unit, which is represented as a vehicle in the simulation. The hardware robot shall perform in concert with the rest of the simulated vehicle in the system. The simulation can be run in one of two control modes: automatic or manual.

The GCS is divided to three components, hardware, software, and firmware that communicate between the hardware and software. The hardware component consists of a robot called the G-Unit, which simulates an airport aircraft or vehicle. The robot moves on a physically simulated runway, which is also representing the simulated airport, and follows barcode marks on the ground to move from one location to another; the points involved in the movement of the robot from one location to another are called waypoints.

The software component contains a visual display of a simulated airport (Figure 1); it represents the vehicles on the airport surface, in addition to the actual robot position on the physical simulated airport. Under the manual operation mode, the user is capable of controlling different vehicles via the GUI interface. Under the automatic operation mode, the Ground Air Traffic Simulator (GATS) will be responsible to ensure the safety in airport ground operations, by detecting and avoiding ground collision between vehicles at an airport, and enforcing the vehicle separation requirements. The failure to carry out such a task could result in serious human and material loss. The communication between the GATS and the robot emulates the actual communication between the air traffic controller and the pilot and is handled by firmware and the wireless communication modem. Under the hijack mode, the robot is capable of ignoring the communication, and operates independently, using it’s on board control and obstacle avoidance.

Figure 1. AGCS user interface

Based on the initial analysis of the system, we determined the highest safety concerns are in the area of robot’s independent operation, its firmware, and communication between robot and GATS.

Figure 2 presents a graphical representation of the parts of the AGCS that was used for this analysis. The analysis boundary is the dashed line.
3. **Software Fault Tree Analysis**

SFTA is a technique derived from a commonly used system-safety analysis technique that is useful in partially verifying the safety aspects of software [6]. The SFTA technique is an adaptation of the fault tree analysis technique, where the undesired event originates by software failures and faults.

SFTA permits determining the most likely failure events related to a software system, and also what sub-events may have triggered these top failure events. The ordering of failure events is shaped in the form of a tree (thus its name), with a potential software failure as parent root, and failure modes or root causes of those failure events as children. The leaves of the tree are composed by basic software failure events or undeveloped events. The root node of the tree will often represent a system-wide undesired event which potentially may inflict the danger of becoming an accident that leads to a catastrophe. Some of the advantages of applying SFTA are [4]:

- Identify contributing circumstances to an unsafe state
- Demonstrate that a system cannot reach an unsafe state
- Demonstrate that unsafe states are reached with very low probability

SFTA provides a backward analysis from the root node to necessary preconditions for the undesired event to take place. The necessary preconditions include the failures that triggered the hazard, and faults which triggered the failures. The necessary preconditions for the hazard are specified in the lower levels of the tree; each level is then joined to each other with logical gates. Each precondition is similarly expanded until all leaves are events that occur with some calculable probability or cannot be further analyzed. The backward analysis investigates the ways in which the undesired event (root node) might feasibly occur.

A powerful component of FTA is its capability to estimate the likelihood of a hazard. However, probabilistic data which can be assigned to a fault is not readily available for software. As explained by [7] there are promising approaches, currently under research by NASA, which may determine probability for software faults. These approaches include the Probabilistic Risk Assessment (PRA) and the fuzzy logic approach that try to enable quantification of software failure types based on taxonomy of elements that affect the software development effort. As described above, the application of a quantitative approach for software fault trees is not yet mature. Due to the lack of probability data about software faults, a quantitative analysis for the GCS system will not be carried out. Therefore, the scope of the SFTA for AGCS is to find a minimal cut set, which represents the ways a software system may fail.

4. **Observations from the Application of SFTA to AGCS**

As previously mentioned, eliminating defects/hazards from the earlier stages of software development, will increase the overall quality of the software. In addition, based on the industrial data, removing the defects at the earliest possible stage, reduces the overall cost of the development, by eliminating unnecessary rework. To this extend, we decided to apply the SFTA techniques to the software requirement specification of AGCS.

There were number of observations which resulted from our experiment.

**Domain/application Expertise:** It became obvious that people, who have the appropriate domain expertise, are better suited to perform the Software Fault Tree (SFT) analysis. The main reason behind this was that one can look at the development of the SFT as a way of verifying the system requirement. The missing, incomplete, or ambiguous requirements will be easier detected by the domain/application expert, than the one who is not familiar with the application.

**Context Diagram:** By using the context diagram, the analyst gains additional insight to the overall operation of the system. This understanding will help the analyst to identify scenarios with would cause the system to reach a hazardous state. This become especially critical, for the analyst who are not very familiar with the domain/application.
Use Cases: In most cases, the use cases which accompany the system requirements will provide additional insight to the operation of the system. A typical use case, also contain one or more alternative flow, which represents system response under unique (out of norm of operation) situation. It became obvious that by analyzing the use cases (specially the ones with alternative flow) revealed additional hazards that would no be found if we only analyzed the system requirement.

Abstraction: Typically, the system requirement is represented as a series of higher level requirement which are decomposed to a set of lower level requirement. In another word, the lower level requirements are driven from the higher level requirements. It became obvious that one can identify the safety critical component of the system, by first concentrating on the higher level requirements. There by reducing the complexity of the SFT. Once the critical components of the system are identified, then one can expand the SFT, by including lower level requirements just associated with those critical sections. Although using this approach does not identify hazards associated with the lower level requirements that are not considered, it will allow for a short cut during the schedule constraints. The better approach would be to start with the higher level requirement analysis, and then expanding to the lower level requirement one components at a time until all requirements are analyzed. The major advantage of this approach is to allow the analyst to concentrate on a smaller section of the problem.

Undesired Top Event: Identifying the undesired top events is one of the first steps in the process of SFTA. A brainstorming session was used for this step. The idea is borrowed from the FMEA and HAZOP technique, where higher level (hierarchy) requirements are analyzed, and system’s deviations of the system that may be found along with effects, severity, causes, and frequency of their occurrence. Of course this is a very subjective activity, and it could be influence by the bias of the analyst. This is an area that needs to be researched, and we believe the advancement of the research in software reliability could affect this area of SFTA. Another challenge in this stage of analysis is the capability of the analyst distinction between the potential damaging failure, and an actual hazard. Again, this is very subjective, and identifying the top undesired event is very challenging. This is even more challenging, if the analyst does not understand the overall domain and application, in which the software is being used. For example, in our application, collision between two vehicle transferring passengers baggage from the terminal to the airplane, has less severity that the collision which involves an aircraft. This results to a higher severity (or catastrophe), if the collision takes place on the wing of the aircraft due to possibility of aircraft explosion and fire that is associated with the large amount of jet fuel.

Beyond Requirement: Requirements represent the system in its most abstract form. As we move from requirements to design [5] and implementation [8] phase, SFTA can play a major role in identification of additional hazards that may have been introduced during the later phases of the development, or the ones who were not detected in previous phases. An advantage of applying SFTA to requirement is the awareness it generates for the developers of the later phases, to pay special attention to the safety critical components of the system. In another word, by applying the SFTA methodology during the requirements stage, we generate a set of base fault trees that can be explored for hazard identification. These trees may be expanded as the project moves into future phases of its life cycle.

Figure 3 represents a high level fault tree diagram for wireless communication loss. It is obvious that FTA tools are needed to perform these analyses. In our project, we used OpenFTA [9].

5. Conclusion

The current complexity and size of safety (mission) critical software systems make it prone to failures and unexpected actions. Different methodologies nowadays help to ensure and verify the safety level of a system, one of them being the software fault tree analysis.

The fault tree analysis methodology is widely used to understand the causes of possible undesired events which potentially can put the system into hazardous states, prone to accidents. Software fault trees can be applied to different phases of the life cycle such as requirements, design, code or testing. If software fault tree analysis is applied in earlier stages of the life cycle (e.g. requirements), it will help discover and diminish faults present in the system, and prevent them from passing to further phases of the life cycle where the faults will be more expensive and difficult to remove.

This paper proposed a guide for the application of a software fault tree analysis to Software Requirements Specification. The guide was used to assess the safety level of the SRS for an application called the Airport Ground Control System, a
system that simulated the behavior between an automated air traffic controller and the vehicles/aircrafts that had to manage.

The proposed guideline serves as a basis for creation of software fault trees. However, the main contributor to the success of identifying faults in the system is still the ingenuity, creativity and domain knowledge of the analyst who perform this task.

As other researchers have demonstrated, although applying SFTA is a very labor intensive activity, the end result is very rewarding. As the system criticality increases, so does the value of SFTA application.

However, it is almost impossible to apply SFTA to the whole system, therefore, it is best to only apply SFTA to most critical component of the system.

References


Figure 3 - Wireless Communication Loss
Engineering Applications of Software and Hardware Reliability

Chandra S. Putcha\textsuperscript{1}, P. Kalia\textsuperscript{2}, F. Pizzano\textsuperscript{3}, G. Hoskins\textsuperscript{4}, C. Newton\textsuperscript{5} and K. Kamdar\textsuperscript{6}

\textsuperscript{1}Professor, Department of Civil and Environmental Engineering, California State University, Fullerton, CA
\textsuperscript{2}Prince Kalia, NASA Goddard Space Center
\textsuperscript{3}Frank Pizzano, Bastion Technologies Inc., Huntsville, AL
\textsuperscript{4}Gordon Hoskins, Bastion Technologies Inc., Huntsville, AL
\textsuperscript{5}Coy Newton, Bastion Technologies Inc., Huntsville, AL
\textsuperscript{6}Kunal J Kamdar, California State University, Fullerton, CA

Abstract This paper discusses the application of principles to FMEA. A typical Tank pressurant is considered as an illustrative example. The tank pressurant system consists of three sets of solenoid valves (each set consists of a pair of valves), three pressure transducers, a vent valve, a heat exchanger and helium cold pressure bottles. Each set of solenoid valves consist of two solenoid valves connected in series. Using FMEA principles, the hardware and software criticality of various components has been established using the principles of FMEA (Failure Modes Effects Analysis).

Key words: Failure Modes Effects, Reliability, hardware, software, series and parallel system

1 Introduction

Failure Mode and Effect Analysis (FMEA) has been used quite extensively in aerospace industry for several years to establish hardware criticality \cite{2}. Determination of software reliability is equally important. There are several methods for determining software reliability \cite{8,14}. One method for performing software reliability is the software FMEA \cite{11}.

There is no accepted standard definition of software reliability till now as it is relatively new concept \cite{8}. However, the following two definitions are used extensively.

Definition 1 (IEEE)

The probability that software will not cause a system failure for a specified time under specified conditions. \textsuperscript{------}8\textsuperscript{].

Definition 2 (John Musa, Bell Laboratories)

The probability that a given software system operates for some time period without software error, on the machine for which it was designed, given that it is used within design limits.

It is to be noted that all the factors are to be considered in deciding an appropriate tool for measuring the reliability and deciding the adequacy of a given system with respect to a performance objective.

2. Literature Review

An extensive literature review has been done as part of this research project to identify the existing methods for Reliability and Risk Analysis of both hardware and software items. These are \cite{3,9,4,1,5,6,7,2}.

The detailed literature review performed as part of this research project as indicated above showed that the performance reliability of a component or a system (consisting of several components) can be measured modeled through the two following methods:

2.1 RBD (Reliability Block Diagram)
2.2 Fault Tree Analysis
2.3 FMEA (Failure Modes Effects Analysis)

These methods are discussed elsewhere \cite{2}.

In this research study, since only schematic diagram has been provided, FMEA has been used to determine the criticality ranking of each component.
3. Concept of Application of FMEA to a software problem:

As stated earlier, FMEA is used for identifying both the hardware and software criticality in this study. The basic concepts are discussed elsewhere [12,13].

4. Methodology

This is explained using the example of a typical Tank Pressurant (Fig.1).

As, it has been stated earlier, FMEA is chosen as the method for Reliability analysis for both hardware and software items.

Assumptions made in this FMEA analysis are listed below.

1. One set of solenoid valves ($A_1$ and $A_2$) are commanded open at Engine start while the second set of solenoid valves ($B_1$ and $B_2$) are commanded to cycle open/closed to maintain a specific range of pressure. The third set of solenoid valves ($C_1$ and $C_2$) are normally closed and serves as a standby redundant system in case set A fail to open or fail to remain open or set B fail to cycle open/close to maintain the required pressure.

2. Set C will be used to replace set A if set A fails to open or remain open or used to replace set B if set B fails to cycle.

3. Redundancy management will require two out of three sensors to agree on tank pressure to issue and remove opening command to the pressurant control valves.

4. Redundancy management must also be able to detect if solenoid valves ($A_1$ or $A_2$) fails to open at engine start or fails to remain open during engine burn or if solenoid valves $B_1$ or $B_2$ fails to cycle open/close during engine burn in order to switch to the redundant set of solenoid valves set C should set A or set B fail.

5. It is to be assumed that the software will obtain measurement information about tank pressure levels and will provide valve commands to maintain the pressure in the proper range and also check whether at least two out of three pressure sensors are functioning ok.

6. The objective of this paper will be to not only identify the hardware failure modes and Criticality, but also to identify the software failure modes and the corresponding Criticality. The software is to respond to hardware failure modes in such a way to maintain system function if possible. If the software cannot perform its function, then that would be a failure mode. There will be several failure modes of the software that will be analyzed.

7. Items such as Heat Exchanger and Pressure Bottles will have only hard failure modes associated with them.

8. Similarly, items such as Pressure Transducers/sensor will have only hard failure modes with them.

Based on the information provided from NASA, it is concluded that FMEA has to be capable of determining the software failure modes, effects and criticality as well as the hardware failure modes effects and criticality which includes the
pressure transducers, solenoid valves, vent valve, heat exchanger and helium cold pressure bottles.

5. Results and Discussions

The results obtained for the Tank Pressurant example considered are stated in Table 1 below for the case of interaction between hardware and software.

Hitherto, FMEA has been used for identifying the hardware criticality only. It has been shown in this report that the powerful tool of FMEA can easily be used for identifying the software criticality as well as hardware criticality.

Since the software criticality will involve the various functions of the same hardware, it is prudent to discuss the Functional Hardware FMEA and Functional Software FMEA. This will cover the criticality of all the hardware items (in Fig. 1) as well as all the associated failure modes.

Table #1 INTERACTION BETWEEN HARDWARE AND SOFTWARE FMEA

<table>
<thead>
<tr>
<th>Index #</th>
<th>Item</th>
<th>Hardware Failure</th>
<th>Criticality</th>
<th>Associated Software Failure Mode</th>
<th>Criticality</th>
<th>Failure Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.</td>
<td>Solenoid Valves B1 and B2. (Redundancy provided by valves C1 and C2)</td>
<td>Valves B1 and B2 Fails to cycle open. (When tank pressure drops to predetermined value)</td>
<td>IR</td>
<td>Unable to decode proper input signals or generate proper output signals resulting in mismatching of hardware operations.</td>
<td>1</td>
<td>If solenoid valves B1 and B2 fail to cycle “open” to maintain tank pressure followed by failing of C1 and C2 to cycle “open”, tank pressurization will be lost resulting in a criticality 1 effect.</td>
</tr>
<tr>
<td>-----</td>
<td>-------------------------------------------------</td>
<td>-------------------------------------------------</td>
<td>-----</td>
<td>-------------------------------------------------</td>
<td>---</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>Valves B1 and B2 Fails to cycle close. (When tank pressure rises to predetermined value).</td>
<td>IR</td>
<td>Unable to decode proper input signals or generate proper output signals resulting in mismatching of hardware operations.</td>
<td>1</td>
<td>If solenoid valves B1 or B2 fails to cycle “close” to prevent tank overpressurization followed by failing of C1 or C2 to cycle “close”, tank overpressurization could occur resulting in a criticality 1 effect.</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>Solenoid valves C1 and C2. (Provides redundancy backup to valves A1 and A2 should A1 and A2 fails)</td>
<td>Valves C1 and C2 Fails to Open</td>
<td>IR</td>
<td>Error in the decoding of input signal or generating proper output signals resulting in the operation to behave unexpectedly resulting in the criticality 1 effect.</td>
<td>1</td>
<td>If valves A1 and A2 fail to open followed by failure of backup C1 and C2 to open, tank pressurization will be lost resulting in a Criticality 1 effect.</td>
</tr>
<tr>
<td>(Provides redundancy backup to valves A1 and A2 should A1 and A2 fails)</td>
<td>Valves C1 and C2 Fails to Close</td>
<td>Error in the decoding of input signal or generating proper output signals resulting in the operation to behave unexpectedly resulting in the criticality 1 effect.</td>
<td>1</td>
<td>If valves A1 or A2 fails to remain open followed by failure of backup C1 and C2 to open, tank pressurization will be lost resulting in a Criticality 1 effect.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Provides redundancy backup to valves B1 and B2 should B1 and B2 fails)</td>
<td>Valves C1 and C2 Fails to cycle open. (When tank pressure drops to predetermined value)</td>
<td>Error in the decoding of input signal or generating proper output signals resulting in the operation to behave unexpectedly resulting in the criticality 1 effect.</td>
<td>1</td>
<td>If solenoid valves B1 and B2 fail to cycle “open” to maintain tank pressure followed by failing of C1 and C2 to cycle “open”, tank pressurization will be lost resulting in a criticality 1 effect.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Provides redundancy backup to valves B1 and B2 should B1 and B2 fails)</td>
<td>Valves C1 and C2 Fails to cycle close. (When tank pressure rises to predetermined value).</td>
<td>Error in the decoding of input signal or generating proper output signals resulting in the operation to behave unexpectedly resulting in the criticality 1 effect.</td>
<td>1</td>
<td>If solenoid valves B1 or B2 fails to cycle “close” to prevent tank overpressur e followed by failing of C1 or C2 to cycle “close”,</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4. Pressure Sensors

Indicates Low Pressure (two out of three sensors)

1R

Contradiction or false signal results in excess pressure to develop resulting in criticality 1R effect.

Tank over pressurization could occur resulting in a criticality 1 effect.

Indicates High Pressure (two out of three sensors)

1R

Contradiction or false signal results in low tank pressure tank resulting in the criticality of 1R to take place.

Tank under pressurization could occur resulting in a criticality 1 effect.

6. Conclusions

It has been shown in this report that the FMEA (Failure Mode and Effect Analysis) is capable of determining the software failure modes, effects and criticality as well as the hardware failure modes effects and criticality through a specific example of a typical Tank pressurant.

7. References


A Quantitative Method to Detect Design Defects and to Ascertian the Elimination of Design Defects after Refactoring

K. Narender Reddy1, A. Ananda Rao2, M. Gopi Chand3, and Kiran Kumar J4
1 Dept. of CSE, CVR College of Engg., JNTU, Hyderabad, AP, India
2 Dept. of CSE, JNTU College of Engg., JNTU, Anantapur, AP, India
3 Dept. of IT, GNITS, JNTU, Hyderabad, AP, India
4 India Software Lab, IBM, Hyderabad, AP, India

Abstract - One of the ways to make object oriented software systems maintainable is refactoring. Effective refactoring requires a proper method to detect design defects. Recently, some quantitative design defects detection methods which are based on metrics have been developed. However, there is a scope for a design defects detection method which considers design change propagation probabilities between artifacts that are connected through intermediate artifacts. A quantitative method is proposed in this paper considering the above aspect. The main advantage of the proposed method is, it can be used not only for design defects detection, but also to ascertain quantitatively the elimination of design defects after refactoring.

Making use of the proposed method, in example designs, two different design defects are detected and the elimination of these defects after refactoring from the design is known quantitatively. The framework in which this method is used is given.

Keywords: Design defects, Bad smells, Shotgun surgery, Divergent change, Design change propagation probability matrix, Refactoring

1 Introduction

Object oriented systems are subject to frequent modifications during software development (iterative or agile software development) and software evolution. As the software is enhanced, modified, and adapted to new requirements the software becomes more complex and deviates from its original design, in turn lowering the quality of software due to design defects. Design defects cause the system to exhibit high complexity, low reuse, faulty behavior, and low maintainability [1]. It is necessary to detect and correct design defects to make software maintainable. One of the ways to make object oriented software systems maintainable is refactoring. Techniques that reduce object oriented software complexity by incrementally improving the internal software quality come under refactoring [2]. In the context of software under evolution, refactoring is used to improve the software quality. The improvement in the software quality is, in terms of, extensibility, modularity, reusability, efficiency, complexity, and maintainability [3]. Refactorings can be applied to reduce the complexity of software design and in turn number of defects. In other words, eliminate design defects using refactoring. Effective refactoring depends on proper method which is used to detect design defects (which indicate locations for refactoring) and to ascertain the elimination of design defects after refactoring.

Some strategies for design defects detection are proposed by [4]. These strategies are based on bad smells and metrics. There is a scope for developing some more new methods. For example, a design defects detection method which considers design change propagation probabilities between various artifacts that are connected through intermediate artifacts. A quantitative method should be useful in detecting design defects and also in ascertaining quantitatively the elimination of design defects after refactoring. In this paper, a quantitative method is proposed which covers the above aspects. This paper also presents a framework for object oriented software design quality improvement (D’ARTI: Design defects detection and refactoring to improve). One of the advantages of the proposed framework is, having the iterative characteristic of spiral model. The design defects detection and refactoring can be done iteratively until the required quality improvement is attained. In general, a framework is a conceptual or real structure intended to serve as a guide or support for the building of something that expands the structure into something useful [5]. The proposed framework is intended to serve as a support or a guide for developing methods and tools for design defects detection and refactoring. As part of methods development, a method is proposed for detecting two important design defects corresponding to two bad smells: shotgun surgery and divergent change. These defects involve change propagation. The proposed quantitative method for detecting design defects makes use of the concept design change propagation probability (DCPP)
matrix, developed for version management [6]. This type of quantitative method which is used to detect design defects and to ascertain quantitatively the elimination of design defects after refactoring is not addressed in the literature.

The organization of the paper is as follows. Section 2 presents the related work on design defects detection methods. Section 3 presents the proposed framework for design quality improvement. A method for design defects detection which is based on DCPP matrix, explained in Section 4. The analysis results of two example designs are also presented in section 4. The conclusions have been placed in section 5.

2 Related Work

In this section, related work on design defects detection strategies/methods, in particular, methods which cover design change propagation (ripple effect) between artifacts is given. The related work on frameworks related to design defects detection, design quality improvement is also given.

Paper [7] indicates that the CK metrics appear to be useful to predict class fault proneness during the design phase. Even though CK metrics [8] are useful in predicting class fault proneness, consider the metric CBO (Coupling between object classes), which is defined as a count of the number of other classes to which it (a class under consideration) is coupled. This definition of coupling accounts for the interaction of a particular class to which a particular class has some sort of interaction. It does not measure the amount (strength) of coupling between any two classes. Considering the number of discrete messages exchanged between classes, the god classes are identified using link analysis method [9]. The proposed method in this paper [9] is used to detect design defect indicated by the presence of god class bad smell. Our proposed method is aimed to detect design defects indicated by the presence of shotgun surgery and divergent change bad smells.

The paper [10] describes the ripple effect metric. It considers its applicability as a software complexity measure for object oriented software. It is mentioned that this approach has potential to improve the stability and efficiency of object oriented software and cut the cost of software maintenance. List of metric based detection strategies for capturing flaws of object oriented design are defined in paper [4]. Papers [10], [4] have not included how the strength of dependency (coupling) between artifacts (which are connected through intermediate artifacts in more than one path) is calculated.

A novel metric based heuristic framework to detect and locate object oriented design flaws from the source code is proposed in paper [11]. This framework is meant for detecting the design flaws. Paper [12] investigated the construction of probabilistic decision models based on coupling measurement to support impact analysis. It provides an ordering of classes where ripple effects are more likely. A metric for measuring the class weakness for object oriented software is proposed in paper [13]. Inter-class weakness is affected by the interconnection of the class over other classes, and increases if the dependency of the class is more. The ripple effect also contributes to the dependency and this effect has also been considered in this paper. Even though, the papers ([12], [13]) have considered ripple effect, but, they have not correlated the ripple effect results with design defects/bad smells, and it is not given how to rectify the affected classes.

The paper [14] gives software transformation framework which is quality driven in the context of reengineering. The papers ([1], [3], [14]) are the driving force (guiding) to the authors of this paper to come out with a generic framework for object oriented software design quality improvement in the context of software under iterative, agile development, and preventive maintenance. The proposed framework is intended to serve as a support or a guide for developing methods and tools for design defects detection and refactoring. As part of this bigger objective, initially it is aimed to develop a novel quantitative method which is used to detect design defects and to ascertain quantitatively the elimination of design defects after refactoring.

3 A Framework for Object Oriented Software Design Quality Improvement (D³ARTI)

In this paper a framework for object oriented software design quality improvement (D³ARTI: Design defects detection and refactoring to improve) is given. The proposed framework (Fig. 1) has an iterative characteristic similar to spiral model. The design defects detection and refactoring (improvement) goes on in iterations until the design is free of defects or required design quality is achieved. In general, a framework is a conceptual or real structure intended to serve as a guide or support for the building of something that expands the structure into something useful [5].

Something useful which is aimed is better design defects detection methods, methods and tools for refactoring which address refactoring in more generic, consistent, flexible, and scalable way. As part of this bigger aim a quantitative method is proposed to detect design defects and to ascertain quantitatively the elimination of design defects after refactoring.

Steps to be followed in using the framework are given below:

Step 1 (Artifacts in design model):
As part of this step artifacts from the software are extracted. The artifacts in design model map to physical entities in different ways like classes, sets of classes, subsystems etc.

Step 2 (Propose/design defects detection methods):
These methods will be based on metrics, design change propagation probabilities, bad smells, design heuristics etc. A design defects detection tool to be developed will be based on the proposed methods. A design defects detection method is proposed in this method. This
The proposed design defects detection method which is based on DCPP matrix is a quantitative method. The proposed method can be carried out in following steps:
1. Representing different possible values of DCPP matrix, as different conditions.
2. Constructing DCPP matrix for a given design.
3. Checking for the conditions satisfied by a given DCPP matrix and correlating these conditions to design defects.
4. Rebuilding DCPP matrix after refactorings.
5. Checking for the conditions satisfied by the rebuilt DCPP matrix.
6. Steps 4 and 5 are carried out while required design quality is not achieved or design defects are not eliminated.

Proposed design defects detection method is in the context of framework given in Fig. 1. Building DCPP matrix before and after refactoring enables us to compare the design quality improvement after refactoring.

Subsections 4.1 and 4.2 covers the above six steps of the proposed method. Construction procedure for DCPP matrix and formulated conditions are given in subsection 4.1. Detection of design defects and refactoring in two example designs, are given in subsection 4.2.

### 4.1 Construction procedure for DCPP matrix

To construct DCPP matrix, first the design is represented as a unified representation of artifacts (URA) graph [15] and then strength of dependency between artifacts is estimated. In this paper, the strength (amount) of dependency between artifacts is represented by the term cdegree [6]. First the cdegree between adjacent artifacts is estimated and then using proposed equations, combined cdegree between artifacts by considering intermediate artifacts is estimated. Using these cdegree values, N x N DCPP matrix is constructed, where N is number of artifacts in the design. This matrix is called as Design change propagation probability (DCPP) matrix. The entry at row A1, column A3 represents the probability that a design change in artifact A1 requires change in A3 so as to preserve the overall function of the system.

**Definition of cdegree:** The degree of coupling (cdegree) of link is the indicator of the amount of dependency that exists between two related artifacts represented by the URA. The value of cdegree has the range [0,1]. Therefore, a change is propagated to related artifacts based on the cdegree value. Since the artifacts are related one another directly (adjacently) or indirectly (through intermediate artifacts) the change propagation should be calculated for the above two cases.

**Cdegree estimation:** Let A and B are two artifacts that are related adjacent and attributes of an artifact B access attributes of an artifact A. In this paper an attribute is considered as a feature of an artifact. Since various number of links are possible between two artifacts, each link can be given a weightage. Based on these weightages the total strength between these two artifacts can be calculated.

For example, consider a class A has three attributes (a1, a2 and a3) and B has five (b1, b2, b3, b4 and b5) attributes and attribute a2 is called three times by attributes of artifact B. Therefore, the weightage of attribute a2 is 3/7 where total number of calls exist between A and B are 7. Similarly, weightage for other attributes can also be calculated. After calculating the weightages for all the attributes, cdegree between A and B is defined as follows.

\[
cdegree = \text{sum of weightages of each link with respect to a2}
\]
attributes of A from B/Total number of possible links from B to A. The denominator can be taken as the total number of attributes exist in A, since this many maximum links can be made. In the above equation, a link is defined as a call made by class B with respect to a method. It is irrespective of number of calls made to each method. That is, all calls of a method constitute a link even though if it is called more than once by a method of class B. The number of call references are taken into consideration while calculating weightage of each attribute.

The cdegree considers method invocation outside the class and variable reference outside the class. Higher the value of cdegree indicates higher the strength of dependency (coupling) between the artifacts. A design change is propagated to related artifacts based on the cdegree value. For example, a design change is propagated to related artifacts whose cdegree value is more than or equal to the threshold (say 0.5) value.

The above procedure is used to estimate cdegree value between adjacent artifacts. A more general approach would be to estimate the ramifications due to a single change. Therefore, the following method is used to compute combined cdegree value for the artifacts that are connected through intermediate artifacts in more than one path. The combined cdegree value is the probability that the end effect will arise, regardless of the path. This can be calculated using probability lemmas. While calculating the combined cdegree value it is to be noted that the events are not mutually exclusive. Therefore, the following formulas are used to estimate the combined cdegree values [6].

\[
cdeg_{a,b} = 1 - \left[ \left( 1 - cdeg_{a,b} \right) \cdot \left( 1 - cdeg_{a,c} \cdot cdeg_{c,b} \right) \right] = 0.70
\]

Using these cdegree values, N by N matrix is constructed, where N is number artifacts in the design.

### 4.1.1 Formulating conditions representing different values in DCPP matrix

Different possible values of DCPP matrix can be represented as different conditions. These conditions are given below.

- **Condition 1:** Majority of the elements in a row containing larger values (>= threshold value, say 0.5)
- **Condition 2:** Majority of the elements in a column containing larger values (>= threshold value, say 0.5)
- **Condition 3:** Both conditions 1 & 2 exist
- **Condition 4:** All the diagonal element values are one
- **Condition 5:** All the matrix element values are one.

Satisfying different conditions indicate different things in the design. Satisfying the condition 4 indicates the probability that any change in an artifact, affects its design maximum. The unit DCPP matrix (satisfying condition 5) indicates the violation of basic heuristic of developing low coupled system. Condition 5 can be used in validating best practices that are used for software development.

Conditions 1, 2, and 3 can be used in detecting design defects indicated by the presence of bad smells ( shotgun surgery, divergent change) in the design. Detecting design defects and refactoring in two example designs are given in the following section.

### 4.2 Detecting design defects in a given design

Two different design defects indicated by the presence of two bad smells shotgun surgery and divergent change are considered for detection. Detecting design defect indicated by the presence of shotgun surgery bad smell and applying appropriate refactorings, is given in subsection 4.2.1 and detecting design defect indicated by the presence of divergent change bad smell and applying appropriate refactorings, is given in subsection 4.2.2.

#### 4.2.1 Detecting design defect indicated by the presence of shotgun surgery bad smell

When every time a change is made to a class, a lot of little changes to a lot of different classes have to be made [16], indicates the presence of shotgun surgery bad smell in the design. This bad smell indicates that the change is propagated to many other artifacts. Change propagation depends on cdegree value between the artifacts. The cdegree value for artifacts which are not adjacent but...
connected through intermediate artifacts represent combined cdegree which is calculated considering rippling effect. Example design 1 which is shown in Fig. 3 is analysed for detecting the design defects. The URA graph of this example design 1 is shown in Fig. 4 and DCPP matrix in Table I.

Table I

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>0.73</td>
<td>0.67</td>
<td>0.55</td>
<td>0.55</td>
<td>0.14</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>0.75</td>
<td>0.75</td>
<td>0.19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0.25</td>
<td>1</td>
<td>0.19</td>
<td>0.19</td>
<td>0.05</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>0.25</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To eliminate the design defect indicated by the presence of shotgun surgery bad smell, appropriate refactorings have to be applied.

Refactoring: To eliminate the design defect caused by the presence of shotgun surgery bad smell, use “move method” and “move field” refactorings to pull all the changes into a single class. If no current class looks like a good candidate to accommodate moved methods and fields, create a new class [16]. By applying refactoring move method, the G class is created. The classes in example design 1 after refactoring are shown in Fig. 5. The URA graph for the example design 1 after refactoring is shown in Fig. 6 and DCPP matrix in Table II.

Table II

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>0.33</td>
<td>0.33</td>
<td>0.27</td>
<td>0.27</td>
<td>0.09</td>
<td>0.33</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>0.67</td>
<td>0.67</td>
<td>0.22</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0.33</td>
<td>1</td>
<td>0.22</td>
<td>0.22</td>
<td>0.07</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>0.33</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.2.2 Detecting design defect indicated by the presence of divergent change bad smell

Divergent change occurs when one class is commonly changed in different ways for different changes in different classes for different reasons [16]. Example design 2 which is shown in Fig. 7 is analysed for detecting the design
defects. The URA graph of this design is shown in Fig. 8 and DCPP matrix in Table III.

If a column (in DCPP matrix) contains high values with respect to a particular artifact (satisfying condition 2), then it can be inferred that this artifact is likely to undergo frequent changes during evolution, and indicates the presence of bad smell divergent change in the design. Fourth column in Table III satisfies the condition 2. The artifact which is likely to undergo frequent changes is D. This condition indicates the design defect caused by the presence of divergent change bad smell in the design. To eliminate this design defect, appropriate refactorings have to be performed.

The artifact which is likely to undergo frequent changes is D. This condition indicates the design defect caused by the presence of divergent change bad smell in the design. To eliminate this design defect, appropriate refactorings have to be performed.

### TABLE III

**DCPP MATRIX FOR EXAMPLE DESIGN 2**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>0.50</td>
<td>0.67</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0.33</td>
<td>1</td>
<td>0.17</td>
<td>0.22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0.18</td>
<td>0.4</td>
<td>1</td>
<td>0.7</td>
<td>0.09</td>
<td>0.4</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td>0.75</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td>0.5</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Refactoring: Identify everything that changes in the class for a particular cause and use extract class refactoring to put them all together. This may lead to creation of a few new classes [16]. In the example design 2 the class/artifact D (Fig. 7) is refactored using the extract class refactoring. Applying extract class refactorings on class D resulted in new classes G, H, and I. The classes in example design 2 after refactoring are shown in Fig. 9. The URA graph for the example design 2 after refactoring is shown in Fig. 10 and DCPP matrix in Table IV.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>0.67</td>
<td>0.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>0.33</td>
<td>1</td>
<td>0.22</td>
<td>0.17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>0.15</td>
<td>0.4</td>
<td>1</td>
<td>0.1</td>
<td>0.4</td>
<td>0.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td>0.75</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td>1</td>
<td>0.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

### TABLE IV

**DCPP MATRIX FOR EXAMPLE DESIGN 2 AFTER REFACTORING**

5 Conclusions

A quantitative method to detect design defects which considers design change propagation probabilities is proposed in this paper. The main advantage of this method is, it can be used not only for design defects detection, but also to ascertain quantitatively the
elimination of design defects after refactoring. Two different example designs are considered for analysis in this paper. These example designs are represented by URA graphs. cdegree values for adjacent artifacts and combined cedge values for artifacts that are connected through intermediate artifacts are calculated. Using these cdegree values, DCPP matrices are constructed. The conditions to be satisfied by DCPP matrix to indicate the presence of defects are specified. The DCPP matrices representing two example designs are analysed. In one example design, defect caused by the presence of shotgun surgery bad smell and in the second example design, defect caused by the presence of divergent change bad smell are detected. Based on the detected design defects appropriate refactorings are applied on the original design and it is known quantitatively with the proposed method that the defects are eliminated after refactorings. The framework in which the proposed method is used is given in this paper.

The proposed quantitative method has to be evaluated empirically using a case study. While applying refactorings, the inclusion of design patterns into the overall design should be considered. Design patterns improve the maintainability of software [17].

6 References


Acknowledgments

Authors of this paper would like to acknowledge and express their thanks to CVR college of Engineering, Ibrahimpatnam, Hyderabad, and JNTU College of Engg., Anantapur of Jawaharlal Technological University, for providing research facilities and financial support.
A Software Maintenance Process Model
With Feature-based Tool and Reliability Metrics

Mr. Abdallah Qaisi, Dr. Omar Albasheer, and Dr. Ahmad Sharieh
Computer Science Department, Amman Arab University, Amman, Jordan
Computer Science Department, University of Jordan, Amman, Jordan

Abstract—Software maintenance cost can be significantly higher than the initial development cost. High maintenance cost is the result of several inefficiency factors. This proposal identifies these cost factors and introduces a process model that minimizes their impact. The new process model targets the entire development team helping them improve their skills in various software maintenance areas, such as: program understanding, change impact analysis, regression testing, documentation, quality assessment, and code complexity metrics. Several new metrics are introduced to measure the reliability of the product, individual features, and functions. A feature-centric tool is developed and used alongside this research to demonstrate the model, metrics, and other ideas introduced in this proposal.

Keywords—Software maintenance process models, maintenance tools, feature-based code analysis, program understanding, change impact, regression testing, metrics.

1 Introduction

Software maintenance is identified as a major cost factor. It has received much less research than software development, despite costing a lot more. Recently, there has been a growing interest in this area of software engineering, especially in the areas of processes and tools. However, the recent studies and maintenance tools are not comprehensive enough in terms of coverage and intended users. This research proposal introduces a new maintenance process model along with a new maintenance tool that is intended to assist the development team in adopting the new model. The process and tool are comprehensive enough to support all major phases of the software maintenance process, and are easy to adopt and use by the entire development team: developers, testers, documentation writers, and project managers.

The tool parses the project code base, and builds visual representations of the product in terms of features and underlying functions. It operates at four levels of details (graphical views) selectable by users based on their level of programming expertise. Through its feature-based graphs, the tool helps each class of users attain and maintain better understanding of the project, so that code changes are safer, debugging more productive, testing more focused, documentation more accurate, and project management more effective.

Good software process models and tools are intended to improve software reliability, and reliability, in turn, reduces maintenance cost further. Most software reliability models and metrics ignore the maintenance process and focus on results, i.e. the number of observed defects. In this proposal, five new in-process metrics are introduced to support the new process model and measure reliability. These metrics, which are implemented inside the tool, measure the Complexity, Maturity, and Reliability of functions, as well as the Reliability of features, and the Reliability of the product - thus the tool name “CMR3”. The aim with these metrics is to help developers reduce complexity and risk, tell testers how much and where additional testing is required, and give managers a good indication of the readiness of the features and the product for delivery to the next phase of software maintenance.

The new metrics are based on the premise that a software product is made up of one or many features, and each feature is implemented by one or many functions. Therefore, the reliability of a product should be computed from the reliability of its features, whereas the reliability of a feature is to be computed from the reliability of its functions. The reliability of a function is based on two new measurements introduced in this research: Feature-Based Function Complexity (FBFC), and Function Maturity (FM). This proposal claims that the code complexity of a function increases as more features use that function. The higher the feature-based function complexity (FBFC), the more likely the function will have defects, and the more maintenance is required to detect and fix these defects. Another claim is that function maturity (age and number of releases the function has been in) matters when measuring reliability, as equally as the function complexity metric.

In the next section, the research problems and assumptions are briefly outlined. Section 3 contains definitions and related work. Section 4 discusses the methodology used in this research and highlights its major contributions. Section 5 contains two case studies, while section 6 concludes this proposal.
2 Problems and Assumptions

The major problem this research intends to address is the ever-increasing cost of software maintenance. “Among the most challenging problems of software maintenance are: program comprehension, impact analysis, and regression testing” [3]. Other cost factors include project mismanagement, and inadequate development environments. The following is a list of these sub-problems and the challenges of software maintenance:

1) Program understanding is becoming more difficult.
2) Change impact analysis is incomplete and inaccurate.
3) Regression testing is often incomplete and unfocused.
4) Project management lacking accurate and timely data to make good decisions.
5) Development environments and tools offering limited support for true maintenance and user coverage.

In addressing these problems, this research builds its solutions on a few hypothesis and assumptions, which directly and indirectly cause the above problems, and increase the overall maintenance cost. Here are the assumptions:

1) Most maintenance tasks or defects are related to a single feature in the product. When it comes to program comprehension of a given product feature, a graphical representation of the feature is truly worth a million lines of its code. Keeping the code path of the feature graphically represented, and isolated from other code complexities, insures easier program comprehension and change impact analysis.
2) Keeping testers and documentation writers away from the source code delays the detection of defects, reduces the accuracy of defect reports and documentation, and increases the time needed to remove defects. Delayed error detection increases debugging time, and leads to further deterioration of the source code.
3) Almost all maintenance tools available in the market today are intended for developers only. Such tools operate at the code level dealing with functions, libraries, and modules. Testers, writers, and managers prefer to work at the product-feature level, so the existing tools are not very helpful to them.
4) Hands-on observation of the project status and reliability metrics by the project manager results in better and more-timely project decisions.

3 Definitions and Related Work

3.1 Software Maintenance: The IEEE definition of Software Maintenance is: “The process of modifying a software system or component after delivery to correct faults, improve performances or other attributes, or adapt to a changed environment.” [4] Software maintenance is more than correcting errors. It includes all the improvements made to the system after it has been delivered to the customer at least once.

3.2 IEEE Software Maintenance Process Model: The IEEE process model for software maintenance has been specified and is shown in Fig 1. There are other variations of this model in the market today [3], but they are not working effectively, as indicated by the ever-increasing high cost of software maintenance throughout the industry.

3.3 Feature-Based Code Analysis Studies and Tools: Recently, a few studies are beginning to cover the problem of locating features in source code [2][20], and a few tools have been developed [8][12][21] to assist in that direction. All these studies and tools are intended for use by developers and offer little to no support to non-developers.

3.4 Program Understanding: “Program understanding is the ill-defined deductive process of acquiring knowledge about a software project through analysis, abstraction, and generalization” [15]. This acquired knowledge aids in performing all three types of maintenance work, outlined above. Key to understanding legacy systems is organizing the knowledge about the subject project and presenting its architecture and design in a graphically intuitive way [18][7].

3.5 Change Impact Analysis: Software change impact analysis estimates the potential effects of changes on the rest of the software. A major problem for developers is that “seemingly small changes can ripple throughout the system to cause major unintended impacts elsewhere” [13]. Developers are able to evaluate the change before actually committing to making the change, or after making the change. This helps developers reduce risks associated with releasing changed software. [17]

3.6 Regression Testing: Regression testing is an expensive testing process used to validate new versions of the software and detect if new faults have been added. “It has been estimated that regression testing may account for almost one-half of the cost of software maintenance” [9]. Effective regression testing requires focus on only the areas that are most likely to contain introduced faults [16]. Better fault exposure capability leads to more efficient defect removal.

3.7 Software Complexity Metrics: McCabe’s Cyclomatic Complexity metric is the most widely used complexity metric [10], and is independent of the language format [11]. Another is Maintainability Index [14][19], which is gaining popularity lately. Cyclomatic complexity measures the number of independent paths through a function, and is
calculated from its connected graph (number of nodes N, number of edges E, and number of connected components P):

\[ \text{Function Cyclomatic Complexity (FCC)} = E - N + P \]

3.8 Software Reliability Metrics: The formal definition of reliability: “the probability of failure-free operation of the software for a specified period of time in a specified environment” [5]. The original reliability metric is shown in the following equation:

\[ \text{Reliability (R)} = \exp(-\lambda t) \]

Where \( \lambda \) is the number of failures per hour, and \( t \) is the time period for which the reliability is to be calculated. The range of values for \( R \) is 0.0 to 1.0, with 0 indicating no reliability, and 1 indicating maximum.

But, like almost all software metrics and models, this metric has its unrealistic assumptions and limitations, such as specifying the time and environment. In general, software reliability measurement cannot be performed easily and directly. Instead, other related software attributes that lead to reliability are measured [6], such as LOC, code complexity, and faults.

4 Methodology and Contributions

To prove its hypotheses, this research introduces a new maintenance process model, five new reliability metrics, and CMR\(^3\) – a tool to assist the development team in adopting the new model and metrics. As part of the research, the CMR\(^3\) tool will be used on a real commercial software project during a given maintenance release cycle that purposely follows the proposed maintenance process model. Performance measurement (such as productivity, quality, and duration) will then be taken and compared with previous release cycles. Further refinements in the tool and the built-in metrics will be made as needed, until the metrics are accurate, and the process is complete. Reduction in maintenance cost should be noted as a result of using this methodology, and should be seen in two important aspects of software production: faster time to market (i.e. higher productivity and efficiency), and better release quality (i.e. less defects escaping to field). The next few sub-sections explain these three contributions.

4.1 Proposed Process Model: The software maintenance process model proposed in this research is shown in Fig. 2. The main maintenance activities: program comprehension, change impact analysis, regression testing, documentation update, and maintenance management are shown as distinct phases, and shaded to indicate the use of CMR\(^3\) tool during each activity. The process model is a continuous loop of carrying out maintenance tasks until the manager determines that the product is feature-complete and ready for delivery. The right-hand side of the diagram is for developers only. After understanding the program and designing and implementing the solution, the third activity (change impact analysis) may find problems suggesting going back to get better understanding of the program, or to the solution’s design and implementation. If the phase indicates no negative impact, the task moves on to testers to revalidate the change and perform regression testing, then to documentation writers to update any related documents.

4.2 Proposed Metrics. There are many known metrics that can be used during the various phases of software maintenance to measure reliability; such as product size (e.g. LOC, function points), cyclomatic complexity, maintenance index, estimated vs. actual durations, number of defects found/ fixed, fix backlog, fix response time, and fix quality. In practice, a small percentage of these metrics are actually being used. “Most technologies developed by the software community have not been transferred into industrial use, and the number of papers on the software process modeling and technology presented at conferences and published in journals is decreasing” [1].

Regression testing in this model is feature-based, rather than module or function based. It’s also more focused on the exact features that have been affected by a particular change, thus limiting their testing to only those areas. Documentation updates are performed on time and with better accuracy. More efficient management is made possible by various reliability measures and trends that are obtained directly via the tool. The full benefits of the model as illustrated by the tool to be discussed in more details in Section 4.3.

4.2.1 Feature-based Function Complexity (FBFC): The higher the Function’s Cyclomatic Complexity (FCC), the more errors it’s likely to cause and the more maintenance it will require [10]. A very high FCC value indicates a potential need to rewrite the function rather than making small changes to it. Rewriting may involve reducing the function nodes and paths, or breaking it into smaller manageable pieces (sub-functions). This proposal claims that a function shared by multiple features is more complex by design and will likely introduce more errors than a single-feature function with the same FCC measurement. The FBFC metric, shown in Equation (1), is a better indicator at measuring the function complexity and estimating maintenance cost.
$$FBFC = FCC * \log (N+9) \quad (1)$$

Where:
- **FBFC**: Feature-Based Function Complexity
- **FCC**: Function Cyclomatic Complexity
- **N**: Number of features that use the function

For unused functions, no computation of FBFC will be triggered, as only accessible functions get assigned FBFC values. For single-feature functions, FBFC equals FCC, since the multiplier is 1 (log 10). For multiple-feature functions, the metric above essentially multiplies the FCC value by a logarithmic value of the actual number of features using the function, plus 9. The range of FBFC is between 0 and 1. If the final result exceeds 1.0, it is truncated to 1.0, which indicates a very high complexity value and a high potential for: errors, change impact, regression test, and maintenance cost. The aim is to reduce the FBFC values in order to improve reliability of the functions, as well as the features and product that use them. Generally, shared functions tend to be relatively small in size (i.e. low FCC value). One should avoid sharing long functions as that increases complexity and maintenance cost. During debugging, identifying shared functions is the first place to look for root causes of multiple feature failures.

### 4.2.2 Function Maturity (FM):

It’s well known that the higher the complexity of a function, the higher the potential for having defects inside the function. While this metric is true, it ignores a very important aspect of software development, and that is: function maturity. A “mature” function that has undergone a lot of testing and been included in several releases will likely have a lower number of defects (i.e. higher reliability) than a brand new function of equal complexity. The new metric “function maturity”, shown in Equation (2), takes into account the maturity of the function with respect to the product’s maturity as a whole. Maturity, as used in this context, indicates the number of times the function/product has been released to customers, the duration of a release cycle, and age since creation date.

$$FM = \frac{(D \times R_f + A_f)}{(D \times R_p + A_p)} \quad (2)$$

Where:
- **FM**: Function Maturity
- **D**: Average duration, in weeks, of the product’s release cycle
- **R_f**: Number of times the function has been part of a product release. It can be zero if created after the last release
- **A_f**: Function’s age in weeks since creation date
- **R_p**: Number of times the product has been released. **R_p** is always a positive integer. If the product has not been released, it has not entered maintenance mode, and this metric, and research, won’t apply. **R_p** is always $$\geq R_f$$
- **A_p**: Product’s Age in weeks since its creation date. **A_p** is always $$\geq A_f$$

A typical product release goes through a lot of testing (unit, integration, system, beta customer testing, etc.) to validate the product features before customer delivery. Every function that goes through a release gets its “age” increased by the duration of the release, thus increasing its maturity. The same applies to the product as a whole.

The computed FM value will always fall between 0 and 1, with 0 indicating minimum maturity and 1 indicating maximum. Obviously, the higher the FM value, the more reliable the function is, and the less effort and cost needed to maintain it. When a function is ported into the project, rewritten, or receives major changes, its creation date should be reset as if it were a brand new function, resulting in a new and lower FM value.

### 4.2.3 Function Reliability (FR):

Feature-based Function Complexity (FBFC) and Function Maturity (FM) metrics are combined into a new metric “Function Reliability” that realistically computes the function reliability. See Eq. (3):

$$FR = \text{average} (1 – FBFC, FM) \quad (3)$$

Where:
- **FR**: Function Reliability
- **FBFC**: Feature-Based Function Complexity
- **FM**: Function Maturity

Typically, the reliability of a function is the complement of the function’s complexity, thus the term (1-FBFC) in the above metric. But, in software maintenance, FM impacts the reliability of the function just as effectively, thus the average of the two in the metric.

Reliability is usually measured as a function of time. As a function grows in complexity and maturity, its reliability is computed over time, typically after each build where the function changed. When these computed FR values are plotted over time, they follow a well-known exponential model - a special case of the Weibull distribution family that is used widely for reliability studies in many fields. The starting point of the curve depends on whether the function is brand new or has been through one or many releases. The reliability of a brand new function follows a curve starting at zero and sloping up as the function reliability increases to a maximum of 1 without reaching 1. See the solid line curve in Fig. 3. The reliability of an existing function follows a similar pattern but the curve starts out higher than zero. See the dashed line curve in the figure.

![Fig. 3. Function Reliability Model](image)

Most functions have a constant complexity value but gradually increase in maturity, which results in better reliability over time. As a function is changed to fix a defect or add a new enhancement, its complexity and reliability change as well, hopefully for the better! Any fluctuation results in a curve that may not be as nice and smooth as the ones shown. In making changes to a function, the developer must keep an eye on the FR values over time, and try to keep them from degrading as much as possible.
4.2.4 Feature Reliability: A software feature is implemented as a sequence of functions and/or methods. The reliability of a feature is therefore dependent on the reliability of its underlying functions. A new reliability metric “feature reliability” is introduced, and shown in Equation (4) below, which is calculated by taking the minimum of all the reliability measures of the underlying functions. This metric is intended for software managers and decision makers. Note: This particular metric was found to be very aggressive in some case studies, especially for very large features. The metric will fine tuned taking into account the total number of functions and lines of code of the overall feature.

\[ R_{\text{feature}} = \min (R_{f1}, R_{f2}, \ldots) \]  
\[ \text{Where} \]  
\[ R_{\text{feature}} \quad \text{Feature Reliability} \]  
\[ F_{fi} \quad \text{Function i Reliability} \]

Fig. 4 shows the reliability model of a feature (X) which consists of three functions (A, B, and C). Notice that, at any point in time, the feature is as reliable as its least reliable function.

\[ R_{\text{feature}} = \text{average} (R_{\text{feature}1}, R_{\text{feature}2}, \ldots) \]  
\[ \text{Where} \]  
\[ R_{\text{product}} \quad \text{Product Reliability} \]  
\[ R_{\text{featurei}} \quad \text{Feature i Reliability} \]

An example product made up of two features is shown in Fig 5. Notice that at any point in time that the product reliability is equal to the average reliability of its two features.

4.3 The CMR³ Tool. The CMR³ tool creates and uses a database of features and functions in the software project, and tracks these relations on build-by-build basis. It reads the source code to count tokens (branches, operations, operators, LOC, etc.) and stores that data inside the function and feature relations. Architecturally, the tool is essentially made up of two components: a parser and a viewer. A simple illustration of the two major components, and their interactions with the project and its database, is shown in Fig. 6. More details on each component will follow.

In terms of user interface, the tool is designed to be extremely easy to use by all its users, novice and experts alike. The tool’s menu bar and menu commands (shown in Fig. 7) are explained next.
Add Feature dialog in Fig. 8), perform the feature inside the running project, and when the feature is executed to completion, the user would click Add Feature in CMR 3. This prompts the tool to take the entire trace log of visited function names and relate them to the new feature in the database for later viewing.

Fig. 8. CMR 3 Add Feature Window

This process is repeated until all the features are entered. Nodes and edges of the feature can be edited for better layout (via Edit Mode commands). Additional nodes can be added manually (via Add Function Node) or removed (via Delete Function Node). After each build and whenever major changes are made to a certain feature, the database representation of the feature can be updated to reflect new code paths and/or new complexity metrics. Multiple metric measurements, taken at each build, help CMR 3 build its reliability curves for each function and feature (via Metrics menu commands).

4.3.1 CMR 3 Parser: To generate the database relations (features and functions), a source code parser is used as part of the tool. The parser assumes C, C++, and Java function/method naming conventions and comments. The parsing should be performed after each build in order to keep the information in the database up to date. The parser automatically inserts a debug-print statement at the start of every function/method. It also extracts function comments and writes it into the function relation. Comments are used by the Viewer to generate Comment Views to help testers in detecting and verifying errors, minimizing interactions with developers and/or the actual code. Comments written in English are easier to understand than high-level language. Developers will have to be aware of this and are advised to thoroughly document their code and use descriptive names for functions/methods. An example view is shown in Fig. 10.

Fig. 10. Comment View of transactions()  

4.3.2 CMR 3 Viewer: Graphical views are generated based on the above relations allowing users to view each project feature at four different levels of details: Tree, Comment, Code, and Metric views. The user can switch between the four views via Show Function View commands. The following is a brief description of each view.

4.3.2.1 Tree View: The default view, and as shown in Fig. 9, is intended for the entire team and as a starting point for many tasks, such as debugging, testing, and documentation. This view shows a tree graph of a particular product feature with its underlying functions in the code path. The tree is read from left to right and in depth-first order. A red function node indicates the function had changed in the current build. A green circle in the upper corner indicates the function is shared by a multiple number of features, and the actual number is shown inside the circle. Shared functions require extra attention during regression testing and error removal.

Fig. 9. Tree View of a Product Feature

4.3.2.2 Comment View. This view provides more information about each feature and function (such as function header, the code comments, etc.). The information provided in this view is mostly intended to writers and testers who need an idea of how a feature works without actually seeing the underlying code or talking to the developers. Comments written in English are easier to understand than high-level language. Developers will have to be aware of this and are advised to thoroughly document their code and use descriptive names for functions/methods. An example view is shown in Fig. 10.

Fig. 10. Comment View of transactions()  

4.3.2.3 Code View. This view is intended for developers and technical testers and managers. This view shows the actual source code of the selected function in a special window (see Fig 11 below). The user also has the choice to open the original file where the function is implemented inside the user’s favorite text editor. Through this view, the user is able to obtain full information on how to analyze, modify, and test each function, with full risk analysis in mind.

Fig. 11. Code View of transactions()
4.3.2.4 Metric View. This view is intended for managers, technical testers, and developers. This view shows several known complexity metrics, new metrics introduced in this research, and the factors used to compute these metrics, such as: number of features using the function, releases, LOC, comments, and aging information. See an example Metric view in Fig. 12.

5 Case Studies
In validating the models, metrics, and ideas presented above, the CMR3 tool is being used on a small open-source Windows Java project, as well as a commercial Mac OS X C/C++ project whose identity cannot be exposed at this time. The tool is used during a maintenance release cycle that follows the maintenance process model proposed here. Performance measurement (such as productivity, quality, and duration) are being taken and compared with previous release cycles. A reduction in the overall maintenance cost should be noted at the end of the maintenance process. This reduction is the result of higher productivity by all team members (i.e. less mistakes are made), and better release quality (i.e. less defects escaping to field).

6 Conclusions
Together, the new process model, the CMR3 tool, and the built-in metrics promise to help software organizations maintain their software projects more effectively and with minimal cost. It’s important to note that this is still work in progress with many features and enhancements deferred to future versions. Among the planned enhancements is error tracking per feature within each project build and across multiple builds, information related to engineers, their specialties and assigned tasks, and more automation of certain aspects of code trace generation, parsing, and analysis.

8 References
SESSION

SOFTWARE QUALITY

Chair(s)

TBA
Modeling Software Systems with Decidable Semantics: Implications on Software Quality Assurance

Kenneth M. Anderson¹ and Steven Bucuvalas²

¹Department of Computer Science, University of Colorado, Boulder, CO, USA
²ioSemantics LLC, Buffalo Creek, CO, USA

Abstract - Programs with decidable semantics have long been considered unworthy of study due to a perception that they apply only to toy problems. Real-world problems, it is argued, require more powerful models of computation with undecidable semantics. This perception is in error, however, since there are application domains (e.g., financial services) that contain programs whose behaviors can be specified using restricted computational models that retain decidable semantics. ioRules is a system for modeling large-scale software systems such that finite, complete-and-correct models of their behavior can be generated and analyzed. The benefit of modeling systems this way is that several SQA activities can be automated, greatly reducing costs. The key to enabling this savings is producing models compatible with subsumption reasoning.

Keywords: constraint logic programming, subsumption reasoning, software quality assurance, ioRules

1 Introduction

Understanding the behavior of software systems is a difficult task. In the context of introducing his book on program analysis [8], Nielson simply states that all approaches to software analysis are approximate and incomplete. And, indeed, this is true for all software systems whose semantics depend on the computational power of a Turing machine. It can be shown that certain classes of less powerful automata can be used to model the semantics of programs whose behaviors can be analyzed completely and correctly, i.e., these programs have decidable semantics. There is, however, a stigma attached to these less powerful classes of automata that suggests that they can only be used to model “toy problems” and, therefore, are not worth the effort of developing languages and tools based on these less powerful computational models.

We assert that this stigma is unfounded and that for certain application domains there are large-scale software systems whose behavior can be completely captured by computational models with decidable semantics. Consider the example of account processing in the financial services industry. These systems are the “bean-counting” automata that credit and debit accounts as financial transactions occur. Banks demand that such systems have decidable behavior. Undecidable semantics in this context would mean “given this input, we’re not sure what the outputs will be!” We always want financial transactions to be accounted for with programs that have decidable semantics. Financial software systems comprise 50% to 75% of the software engineering activities in the banking, insurance, and mutual fund industries. For the U.S. Economy this represents approximately 50 billion dollars spent annually on development and maintenance of such systems. Yet these software systems require only a simple computational model: non-recursive procedures, bounded looping, and linear numeric expressions.

We have developed an approach to modeling the semantics of software systems whose computational needs match those described above and such that our techniques can produce a complete and correct model of the run-time behavior of such systems. That is, for any legal input to the modeled system, our model predicts accurately the output that will be produced. Furthermore, the models produced by our techniques are compatible with a form of automated reasoning called subsumption reasoning [3] that allow us to determine how program A’s behavior relates to program B’s behavior. We make use of this compatibility with subsumption reasoning to automate portions of the software quality assurance (SQA) process associated with our modeled software systems.

In this paper, we start by describing an example of generating a semantic model of behavior for a program written in a language with a simple computational model. The steps in that process serve as a foundation for understanding the more powerful language (that nevertheless retains decidable semantics) that is embedded in a system developed by the second author called the ioRules development platform (ioRules). IoRules supports the modeling of large-scale systems drawn from financial services or similar domains.

We provide details about ioRules and then discuss its logic programming language. This language, known as iorLP, allows users to specify the behavior of large-scale systems, such as mortgage loan processing systems, while retaining decidable semantics. An iorLP program produces a semantic model containing a finite number of patterns, each of which represents a unique execution path through the program with all the attendant conditional constraints and calculations encountered in that path. Because iorLP programs have decidable semantics, the generated semantic model is both complete and correct, covering all possible paths through the program and correctly capturing the constraints associated with the program’s input and output variables. These models can be processed by a subsumption reasoning algorithm, as mentioned above, to enable the automation of activities traditionally associated with SQA.
IoRules provides two means for writing iorLP specifications: (1) a text interface in which programs are written using a Prolog-like syntax and (2) a graphical spreadsheet-like interface. We provide insight into the algorithm that converts an iorLP specification into a complete and correct model of that specification’s behavior. As financial systems process both linear and non-linear equations, developing a complete algorithm for the data flow of our specifications required borrowing from other disciplines, e.g., constraint logic programming [7] and nonlinear numeric methods [5].

After presenting the details of our semantic model, we describe our subsumption-based reasoning algorithm and show how it can help to automate activities associated with SQA. In particular, since the semantics of the modeled system are decidable and compatible with subsumption reasoning, we can eliminate the traditional need for test cases and instead present users with information concerning whether a change in the specification of a software system has changed its behavior. For instance, a change might alter the behavior of a system such that it no longer meets quality criteria specified for a previous version.

We conclude by presenting information on the types and scale of systems that can be modeled in ioRules and by outlining a future research agenda that will require techniques from many different disciplines to address adequately. The main theoretical issue facing this research is the tradeoff between the expressivity of a specification language and its ability to retain decidable semantics. Other important issues include the computational complexity of semantic model generation, incremental model generation, how best to present the information contained in the model, and so on.

2 Simple Example

In this section, we present an example of generating a semantic model for a programming language with a simple computational model. This language has two data types, CODE and FLOAT—with their associated literals and expressions—support for variables and procedure calls (excluding recursion) and simple statements, such as assignment, if-then-else, return, etc. but with no looping constructs. Fig. 1 shows a simple program written in this language.

The program takes a price and a product type and determines the final price of the product after applying discounts and sales tax. Generating a semantic model for programs written in this language involves three steps: logic representation, control/data flow processing, and semantic model generation. The source code is first translated into a logic representation that transforms imperative program statements into a series of logical assertions. The model logic for the program displayed in Fig. 1 is shown in Fig. 2.

```
System PriceCalculator {

Entry Point: CalcPrice;

Numeric DiscountAmount(Code ProductType, Numeric BasePrice) {

Numeric Discount;

if (ProductType == "SaleItem") {

Discount := 0.1 * BasePrice;

} else {

if (ProductType == "SuperSaleItem") {

Discount := 0.4 * BasePrice;

} else {

Discount := 0.0;

}

return Discount;

}

Numeric CalcPrice(Code ProductType, Numeric BasePrice) {

Numeric Discount;

Numeric FinalPrice;

Discount :=

DiscountAmount(ProductType, BasePrice);

FinalPrice := BasePrice + -1.0 * Discount;

FinalPrice := 1.08 * FinalPrice;

return FinalPrice;

}

Figure 1. Simple Program with Decidable Semantics
```

As part of the translation process, all procedure invocations get assigned unique ids and all assignments to variables cause a new data element to be created. For instance, in the DISCOUNTAMOUNT() method, three statements assign a value to the DISCOUNT variable. As a result, three data elements are created: DISCOUNT.1, DISCOUNT.2 and DISCOUNT.3. During control flow processing, the algorithm will know which data element was actually encountered by an execution path.

The data flow and control flow constructs of the logic representation are then processed by a tableaux algorithm [1] that computes all possible execution paths and calculates all possible output values for the source program. While generating all of the possible paths through the program, it keeps track of all the constraints and calculations being applied to the input variables as they are transformed into the output of the program. At the end of this algorithm, an “assertion box” has been created for each path of the program that contains all of the constraints for that particular path. The program in Fig. 1 has three possible paths depending on the value of the input variable PRODUCTTYPE. Fig. 3 shows the assertion box that was generated for the path that is taken when this variable has the value “SUPERSALEITEM.” As can be seen, the assertion box maps data elements into instances and these instances participate in the constraints that are being accumulated for each input and output variable. In Fig. 3, it

---

1 Due to space constraints, we are unable to provide rigorous definitions for the structures that appear in Figs. 2–4.
AND
  AND
  CalcPrice.0.BasePrice.0  ==  ANY
  CalcPrice.0.ProductType.0  ==  ANY
)  ProcedureLink()
  CalcPrice.0
AND
  CalcPrice.0.Discount.1  ==  DiscountAmount.0.RESULT
AND
  DiscountAmount.0.BasePrice.0  ==  CalcPrice.0.BasePrice
  DiscountAmount.0.ProductType.0  ==  CalcPrice.0.ProductType
)  ProcedureLink()
  DiscountAmount.0
AND
  OR
  AND
    DiscountAmount.0.ProductType  ==  "SaleItem"
    DiscountAmount.0.Discount.1  ==  0.1  *  DiscountAmount.0.BasePrice
  )
  AND
    DiscountAmount.0.ProductType  !=  "SaleItem"
    DiscountAmount.0.Discount.2  ==  0.4  *  DiscountAmount.0.BasePrice
  )
  AND
    DiscountAmount.0.ProductType  !=  "SaleItem"
    DiscountAmount.0.Discount.3  ==  0.0
  )
  )
  DiscountAmount.0.RESULT  ==  DiscountAmount.0.Discount
)  )
  CalcPrice.0.FinalPrice.1  ==  CalcPrice.0.BasePrice  +  1.0  *  CalcPrice.0.Discount
  CalcPrice.0.FinalPrice.2  ==  1.08  *  CalcPrice.0.FinalPrice
  CalcPrice.0.RESULT  ==  CalcPrice.0.FinalPrice
)  

Figure 2. Logic Representation of Simple Program

can be seen that when PRODUCTTYPE equals “SUPERSALEITEM” that seven numeric constraints and three code constraints are encountered along the execution path. We apply Fourier elimination to the constraints of each assertion box to produce the final set of constraints on the input and output variables of the simple program. This produces three patterns, one for each possible execution path, which defines the semantic model for the entire program. The semantic model for the simple program appears in Fig. 4.

The language used in this section is, of course, too simple to be used to model large-scale systems. However it serves to demonstrate the basic ideas behind our work and illustrates the major steps of generating a semantic model for programs with decidable semantics. In ioRules, iorLP (ioRules logic programming language) has sufficient expressivity to model large-scale systems while retaining decidable semantics. We present ioRules and iorLP in more detail next.

3 IoRules Development Platform

IoRules uses model-based techniques to solve two classes of SQA problems: policy satisfaction and regression testing. We will discuss the support that ioRules provides for SQA in Sec. 6. As mentioned above, SQA activities are supported using subsumption-based reasoning on semantic models of program behavior automatically generated from specifications written in iorLP. The system attains decidability of subsumption reasoning by restricting programming constructs while retaining the expressiveness needed for large-scale applications. IoRules also allows the end user to browse the generated semantic model of a program, viewing the behavior of the program flexibly from multiple perspectives.

The user interface for ioRules is designed by analogy to other common business tools: spreadsheets and database query tools. Its core development concept is a table, which is used to specify a rules-based system. The ioRules specification interface is illustrated in Fig. 5. A table allows a user to express the input description of a system, as well as its output and calculation behavior. As we will show in the next section, the constructs of ioRules’ spreadsheet interface map to familiar concepts in logic programming. For readers familiar with Prolog, the “input” portion of an ioRules table corresponds to the “head” of a Prolog predicate, and the “output” portion corresponds to the “tail” of a Prolog predicate. Each column corresponds to a term in an individual predicate, and rows correspond to repetition of predicates for different potential solutions.

In Fig. 5, a specification for a mortgage pricing system is shown. A “main” table has defined the following input columns: loan amount, property value, income, debt, and FICO. These comprise the data needed to determine product eligibility and pricing for consumers. The output data needed by a consumer to make a mortgage selection are the product
type, the rate being paid, and the monthly payment. The output area shows the structure of the output response of the system and the possible solutions. Fig. 5 shows that this system is offering five different potential product choices: conforming 15/30-year mortgages (c15/c30), non-conforming 15/30-year mortgages (nc15/nc30) and a conforming 1-year adjustable rate mortgage (arm1). This specification also demonstrates IoRules support for modularity as the mortgage pricing domain was decomposed into several tables, each representing highly cohesive conceptual chunks. IoRules allows these tables to be linked together such that queries and potential solutions can flow back and forth across the links. In this example, three additional tables were created: a PRODUCT table, an ELIGIBILITY table and a RATEADJ table.

In addition to being able to create specifications and view potential solutions, IoRules offers tools that allow a user to specify policies that indicate properties that must always be true of a specification, to capture portions of the semantic model of an existing specification to be used in the regression testing of future versions of that specification, to perform traditional testing by supplying inputs to a specification and viewing the possible outputs, and to browse the semantic model in various ways.

4 IoRules Logic Programming Language

IorLP is a logic programming language similar to Prolog in many ways but with key differences. Programs in IorLP are a collection of clauses that consist of either rules or facts. Each rule defines an n-ary predicate as its head and a set of clauses as its tail that define the conditions that must be true to consider the head as true. A fact is simply a predicate in which constant values have been assigned for each argument of the predicate. Thus:

```
FATHER(X, Y) :- PERSON(X), PERSON(Y), MALE(X), PARENT(X, Y).
PERSON(KEN).
PERSON(MAX).
MALE(KEN).
PARENT(KEN, MAX).
```

defines one rule and four facts. Once a logic program has been loaded into its interpreter, queries can be made by entering predicates in which some or all of the arguments are passed in as variables. Thus, in the example above, the query `FATHER(KEN,Y)` finds a solution by assigning MAX to the variable Y. As mentioned in Sec. 3, each table in the IoRules GUI editor can be mapped to an IorLP program. Thus, Fig. 5 shows graphically a set of definitions for the predicate MAIN that textually take the form:

```
MAIN(LOANAMOUNT, PROPERTYVALUE, INCOME, DEBT, FICA, PRODUCT, RATE, PAYMENT) :-
    LOANAMOUNT > 0, PROPERTYVALUE > 0, INCOME >= 0, DEBT >= 0, FICA > 0,
    PRODUCT = PRODUCT(...),
    RATE = BASERATE(PRODUCT) + SUM(ADJUSTMENT(...)),
    PAYMENT = PAYMENT(...).
```

As with Prolog, IorLP programs define a database of facts/rules; queries are resolved using unification and depth-first search over the database. Here the similarity to Prolog ends. We now discuss the main differences between IorLP and Prolog. Recall that our goal is to create a language similar to Prolog but with decidable semantics.

Find All Solutions: The search algorithm of IorLP is implemented to always find all possible solutions to a query. This is different from Prolog, which generally finds one solution at a time.

Semantic Model Generation: The purpose of IorLP is to generate a semantic model that contains information on all execution paths of a modeled software system and shows what output is produced by each path. (See Fig. 4.) This differs from Prolog in which the output of a logic program depends on the queries that are submitted to it. IoRules, on the other hand, generates all possible solutions that are valid based on the possible values its input variables can take on. The result of an IorLP program is a set of patterns that specify all possible outputs the program can produce.

Recursion and Iteration: Logic programming and Prolog support both general recursion and iteration. However, these control-flow features are known to have undecidable semantics. IorLP allows for limited recursion and iteration predicates, but does so in a fashion that retains decidable semantics. IorLP recursion is limited to predicate calls that in their transitive closure produce a call-invocation tree that is a DAG. If a predicate were to reference itself, directly or indirectly, the DAG structure would break down and the resulting program would have undecidable semantics. In Prolog, iteration is implemented either by using general
recursion or by using repeat-fail loops. Since neither general recursion nor looping are allowed in iorLP, they are replaced with decidable “aggregation” predicates. iorLP builds conceptually upon the Prolog built-in predicate FINDALL and converts iteration into: (1) decidably producing lists, and (2) applying decidable aggregation predicates to these lists. In the initial iorLP implementation, there are four built-in predicates that implement business-level semantics for which iteration is used: SUM(), COUNT(), MAX(), and MIN.

Support for Numeric Constraints containing Linear and Non-Linear Functions: As iorLP generates a semantic model for a program, it has to resolve systems of numeric constraints involving the program’s input and output variables. (See Fig. 3.) iorLP can resolve these constraints even if they include references to linear and non-linear functions (for a certain restricted class of non-linear functions). Our constraint solver makes use of Fourier elimination and two numeric algorithms (fixed-point iteration and repeated bisection) to ensure that constraints involving non-linear functions can be resolved in a decidable way. The specific restrictions on non-linear functions deal with monotonicity, the presence of inverse functions, and the connectedness between variables processed by non-linear functions and subsequent control flow constraints [4].

5 IoRules Reasoning Algorithm

The semantic models generated from iorLP specifications are compatible with a form of automated reasoning known as subsumption reasoning. The basic goal of subsumption reasoning is to determine if a set $A$ is a subset of some other set $B$. This type of reasoning is used in description logics [2] to determine subset relationships between concepts defined in a description logic specification. For instance, given reasonable definitions of the terms STUDENT and PERSON, subsumption reasoning can determine that the former is a subset of the latter. Indeed, subsumption reasoning can determine subset relationships that involve making inferences over an arbitrary number of rules and facts.

IoRules transforms this form of ontological set reasoning to set reasoning applied to program behaviors. Each predicate in an iorLP program corresponds roughly to a function that can produce a variety of output depending on the input it receives. A pattern in the semantic model captures all of the predicates “touched” in a particular execution path of the program and records them as a series of constraints applied to input, output, and intermediate program variables. These constraints specify the range of values that these variables can take for this particular path, thus acting the same way a concept does in description logics. Once the constraints of an iorLP program have been partitioned into the patterns of a semantic model, they can be used to compare sets of behaviors across multiple iorLP specifications.
For example, imagine that we have specified a mortgage loan pricing system in iorLP and that the resulting semantic model has tens of thousands patterns. Each pattern specifies a constraint on the input variable LOANAMOUNT of the form:

\[
200,000 < \text{LOANAMOUNT} \leq 417,000 \\
150,000 < \text{LOANAMOUNT} \leq 175,000 \\
0 < \text{LOANAMOUNT} \leq 200,000 \\
\]

Further, imagine that none of the constraints set an upper bound higher than the first listed constraint or set a lower bound lower than the third listed constraint. Given this situation, we know that no matter what execution path the loan system takes, the legal values for LOANAMOUNT is captured by this constraint:

\[
0 < \text{LOANAMOUNT} \leq 417,000 \\
\]

Given this information, we can use subsumption reasoning to check this knowledge of the variable LOANAMOUNT with assertions made against its iorLP specification or with knowledge of the variable LOANAMOUNT contained in some other semantic model. Imagine that an assertion has been made that states that LOANAMOUNT must be less than 418,000. Subsumption reasoning would allow us to state with confidence that this assertion would always be true of the loan pricing system since “0 < LOANAMOUNT ≤ 417,000” is a subset of “LOANAMOUNT < 418,000”. Furthermore, if the latter constraint was a member of a semantic model of a different version of the loan pricing system, then we could say that the behavior of the new system subsumes the behavior of the original system for this variable.

Indeed, the subsumption reasoning algorithm implemented in ioRules enables these types of comparisons for all of the input and output variables of two separate semantic models at once. In particular, it produces a lattice that captures all of the various constraints made on a variable and shows how the values of the other variables are constrained for a given partition of the original variable’s legal values. The lattice is used to allow the semantic model of an iorLP program to be viewed in a number of different ways.

### 6 Software Quality Assurance

The key benefit to applying automated reasoning on programs with decidable semantics is that their behavior becomes query-able. This database-like ability to answer queries about the program’s behavior enables automation of certain SQA tasks. Traditional SQA is labor intensive and error prone; the primary challenge is the difficulty of creating test sets that cover all of a program’s possible behaviors. As these test sets grow in size, it becomes difficult to run the entire set of tests after a change. This problem has led to an active area of research in the software engineering community related to regression test selection [6]. The problem with the manual creation of test cases is that it is impossible to “test everything;” programs with undecidable semantics possess an infinite number of states and execution paths and its impossible to create a test set that can cover all of them.

This problem is greatly reduced and/or eliminated when dealing with programs that have decidable semantics. Such programs have a finite number of execution paths and it becomes possible to achieve complete coverage of a program’s possible behaviors. However, although finite, the number of paths can still be quite large. IoRules has been used to model large financial systems that, when implemented using traditional techniques, are thousands to tens of thousands of lines of code and, in one instance, contains approximately 80K distinct execution paths. Regardless, the nature of SQA changes to one in which the semantic model enables monitoring of the entire set of program behaviors.

In addition, given the representation of the semantic model, it is possible for a single pattern to cover an infinite number of test cases. This is easy to see if you consider that a pattern might declare that it applies when an input variable is “greater than 0.” As a result, a finite set of patterns (i.e. a semantic model) can verify completely a software system that has ostensibly an infinite number of states.

IoRules provides support for two primary SQA activities: policy satisfaction and regression testing. In a policy problem, the user wants to know whether a program being evaluated satisfies a constraint, i.e. a policy, that is critical in the application domain. In a regression problem, the user wants to know whether a modified version of a program implements the same functionality as some original baseline.

**Policy Support:** To support policy definition and validation, the ioRules policy editor allows an analyst to specify assertions that must always be true about their iorLP specification. The user can specify multiple assertions on any of the input and output variables defined in the program’s iorLP specification and can then click a single button to apply subsumption reasoning on the specification to ensure that the program’s behaviors are all subsets of the assertions created in the policy editor. When an assertion has failed, the semantic model can be used to pinpoint the failure.

**Regression Test Support:** Regression testing is supported in a unique fashion within ioRules. A regression test set is simply a portion of a previously generated semantic model. Drawing on the mortgage example, after an iorLP specification has been created that models the behavior of a mortgage processing system, an analyst may want to ensure that future changes to the specification do not affect the behavior of the system with respect to particular mortgage products. In the ioRules quality browser, the semantic model can be filtered to show patterns matching a certain criteria. A regression test can then be created by making a copy of the selected patterns and inserting them into a new semantic model. This new model is given a name and is henceforth available in the ioRules regression validator to be applied to
new versions of the semantic model for the mortgage processing system.

**Other SQA Activities:** IoRules supports other SQA activities as well. The ioRules quality browser allows an analyst to browse the generated semantic model of their iorLP specification. Traditional testing is supported via the ioRules “Pop and Test” tool which allows a user to enter values for each input variable and be shown the generated output.

The importance of automating SQA activities cannot be overstated. SQA is traditionally a labor intensive and error-prone process due to the fact that traditional systems have undecidable semantics. Many of the traditional problems of SQA can be removed if the program under test is one that uses a model of computation with decidable semantics and has a semantic model that is compatible with automated reasoning techniques such as subsumption.

### 7 Metrics

IoRules has been used to model large mortgage processing systems consisting of thousands of lines of code. One such system generates a semantic model consisting of 80K patterns that takes fifteen minutes to produce on a Dell Precision workstation using a 3.0Ghz processor and 1.5 GB of RAM. The original system consisted of 4,500 lines of Prolog and 8,000 lines of Java and accessed a database consisting of 3500 rows of data. In ioRules, the need for the Java code was eliminated. That code dealt with the system’s user interface. The logic within the Prolog code could be represented more compactly in ioRules due to its increased support for numeric calculations/constraints and the database was migrated into the ioRules specification directly. We estimate that the size of the original Prolog code was reduced by 25% during its translation into iorLP and that in our experience we see similar reductions when modeling other financial systems.

IoRules is currently being evaluated by a large North American bank for its potential to model one of their production systems and to apply the ioRules approach to automating SQA activities to their development process.

As a result of these metrics and the existence of outside interest in our tools and techniques, we have confidence that there are practical as well as theoretical benefits to exploring the boundary between the expressivity of a software specification language and the decidability of the behaviors that can be generated from such a specification. We believe our techniques can scale to model even larger systems as our techniques depend more on models of computation with decidable semantics, than the number of instructions contained in the systems to be modeled.

### 8 Conclusions

Our work on ioRules is motivated by our belief that it is time to revisit the concepts, techniques, and tools that can be provided to software engineers working on programs with decidable semantics. This class of systems, while necessarily including “toy programs,” also includes large, production systems that organizations depend on. In addition, there are domains that demand decidable behavior of their systems, such as financial services, in which undecidable behavior may lead to legally unacceptable inaccuracy in financial accounting. Indeed, one might ask the rhetorical question: “When would businesses ever want undecidable behavior in commercial systems?” Developers in these domains would benefit from having languages that allow the construction of these systems while retaining decidable semantics. Finally, as we have shown, programs with decidable semantics can be analyzed by automated reasoning algorithms that, in turn, can automate significant portions of SQA activities.

Our future work on ioRules will touch on important issues such as balancing iorLP’s expressivity with its decidability, characterizing the types of non-linear functions that iorLP can use, developing techniques for incrementally updating large models, visualizing the contents of models, and looking for additional domains where these techniques can be applied. We believe the software engineering community can make significant contributions to the design and implementation of techniques and tools that allow developers to create and evaluate software systems with decidable semantics.

### 9 References


Software Quality Assessment and Project Risk Management Based on Bayesian Belief Networks

Kanaan Abed Faisal
Information and Computer Science Department
King Fahd University of Petroleum and Minerals
Dhahran 31261, Saudi Arabia

Abstract - The ability to reliably predict the risk for a software project presents a significant advantage for a development team. It provides an opportunity to address high risk components earlier in the development life cycle, when their impact is minimized. This article proposes a Bayesian Belief Networks (BBNs) model that captures the evolution of the quality of a software product and provides reliable forecasts of the end quality of the software being developed in light of factors that affect project risk. Insights into risk management, development team skill, software process maturity, and software problem complexity are hypothesized as driving factors of software product quality. The cause-effect relationships between these factors and the elements of software quality and risk management techniques are modeled using BBNs. The BBN software project risk assessment model is utilized to quantify the factors that influence and represent software quality.

Keywords: Bayesian Belief Networks, Reliability, Software Project Risk Assessment.

1 Introduction

Recent surveys suggest that 61% of software projects are delivered over budget, late or with fewer features originally specified in the scope of the project. There is no doubt that the field of software engineering is fraught with software projects that have been unsuccessful or mismanaged. Inherent uncertainty in software projects has been previously tackled through complex statistical tools and isolated software metrics. Numerous researchers have proposed the use of Bayesian belief networks to cope with this uncertainty [1-12].

The collection of software metrics in isolation is not a reliable method to predict software attributes [1]. Bayesian belief networks are effective primarily because of their nature to allow for the inclusion of all evidence that affects software attributes. We have yet to see research that applies BBNs to software project management holistically.

Software quality is immensely affected by the process with which it is developed. Hence [1-4] propose Bayesian belief networks that aim to improve the software development process. [2,6,7,8,9,10] focus on particular areas of software development and propose models that specifically aim to raise the level of software reliability by predicting possible residual defects. The authors in [5, 11] consider factors that affect project risk but again with little consideration of other software quality attributes such as software architecture, defect detection and software quality.

In this paper we aim to propose a holistic model that aims to combine the best of the techniques surveyed into a single model that can assist project managers in making a software project successful in all aspects of development; budget, schedule, quality, residual defects, etc.

2 Bayesian Belief Networks

A Bayesian network is a directed acyclic graph associated with a set of probability tables. The graph is made up of nodes representing variables, arcs representing probabilistic dependency relations among the variables and local probability distributions for each variable given values of its parents. Nodes in a BBN represent random variables, whose states are usually expressed in discrete numbers whose states or ranges. Arcs represent the causal relationships between the variables. A Conditional Probability Table (CPT) is associated with each node to denote such causal influence. The node representing a variable A with parent nodes B1, B2, ..., Bn is assigned P(A|B1, B2, ..., Bn) as its CPT. CPTs are filled using a mixture of empirical data and expert judgment, sometimes using the maximum likelihood approach. When root nodes are unknown, current tools usually assign evenly distributed probabilities to these roots. Once new evidence is obtained, evidence can be plugged in the graph followed by re-calculation and updating of nodes to child nodes and vice versa.

A BBN graph can be expanded into an influence diagram by adding decision nodes and utility (cost, or profit) nodes, represented by rectangles and diamonds, respectively. Figure 1 - Sample BBN Topology is a simple example, where the number of defects detected is influenced by problem complexity, the size of the design and the testing efforts applied to the project. BBN’s are criticized for the subjectivity in constructing its influence diagrams and CPT’s. In general, a BBN models the constructor’s belief. Based on this belief, it provides mathematical calculation and prediction. BBNs can be used to support visible and repeatable decision-making. Ziv and Richardson (1997) have used BBN in software testing and maintenance.


Figure 1 - Sample BBN Topology

3 Software Project Risk Assessment

Risk is defined as “The possibility of suffering harm or loss; danger.” Unlike the hazards of daily living, the dangers in the young and emerging field of software engineering must often be learned without the benefit of lifelong exposure [1]. A more deliberate approach is required. Such an approach involves studying the experiences of successful project managers as well as keeping up with the leading writers and thinkers in the field. One such writer in the area of risk is Dr. Barry W. Boehm. In his article “Software Risk Management: Principles and Practices” [2] he lists the following top ten software risk items:

- Personnel Shortfalls
- Unrealistic schedules and budgets
- Developing the wrong functions and properties
- Developing the wrong user interface
- Gold-plating
- Continuing stream of requirements changes
- Shortfalls in externally furnished components
- Shortfalls in externally performed tasks
- Real-time performance shortfalls
- Straining computer-science capabilities

Software Development Life Cycle (SDLC) constitutes the following five steps; initiation, development or accusation, implementation, operation or maintenance and disposal [12]. The above mentioned risks are prevalent through all the phases of SDLC. Project management is the controlling process that governs the execution of the previous mentioned phases. Risk management can be iteratively applied to each phase in the lifecycle to reduce negative impacts on a project.

4 Literature Review

We have conducted an exhaustive survey of research work in the area of BBN assisted software engineering. Various researchers have utilized Belief Networks to assist in reducing costs, risks, time to market and increasing the product quality. The following section summarizes the bulk of the most current research done in this area.

4.1 BBN for Assessing Software Architecture

Software development is done in phases that are usually performed sequentially. Changes or deviations required in later stages compel that the earlier phases be revisited and this impacts the cost, quality and maintainability of the entire system. Therefore this has been an area where a great deal of effort has been done to improve the quality of software, there by reducing the cost and improving the quality at the same time. In the inception phase of a software project only partial and fragmented data is available, in the form of functional and non-functional requirements. This information is used by the designers to create an initial system architecture blue print. Software architecture represents the initial design plan based on which the entire solution or application is developed. Architecture provides the framework and the technical boundary, using which other design and implementation decisions are made.

<table>
<thead>
<tr>
<th>Category</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>Architecture Style, Class Inheritance depth, Component Granularity, Component Interdependencies, Context Switches, Coupling, Documentation, Dynamic Binding, Exception Handling, Implementation Language, Interface Granularity, Multiple Inheritance, Number of threads</td>
</tr>
<tr>
<td>Attributes</td>
<td></td>
</tr>
<tr>
<td>Quality Criteria</td>
<td>Complexity, Fault Tolerance, Horizontal Complexity, Memory Usage, Responsiveness, Security, Testability, Throughput, Understandability, Vertical Complexity</td>
</tr>
<tr>
<td>Quality Factors</td>
<td>Configurability, Correctness, Flexibility, Maintainability, Modifiability, Performance, Reliability, Reusability, Safety, Scalability, Usability</td>
</tr>
</tbody>
</table>

The quality of software architectures is assessed using qualitative techniques as opposed to quantitative. This research work utilizes Bayesian Belief Networks (BBN) to assess the quality of a system’s architecture, using a combination of qualitative and quantitative data. For the purpose of providing a meaningful support information to make design decisions an initial BBN has been developed called the SAABNet (Software Architecture Assessment Belief Network). Variables within this network have been divided into the following three categories.

The values for the above mentioned variables are two state of yes/no, true/false, high/low etc. apart form Architecture style which has been deemed have four possible values. The BBN based on these variables was

Table 1 - Software Architecture Assessment Belief-Network Variables

<table>
<thead>
<tr>
<th>Category</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>Architecture Style, Class Inheritance depth, Component Granularity, Component Interdependencies, Context Switches, Coupling, Documentation, Dynamic Binding, Exception Handling, Implementation Language, Interface Granularity, Multiple Inheritance, Number of threads</td>
</tr>
<tr>
<td>Attributes</td>
<td></td>
</tr>
<tr>
<td>Quality Criteria</td>
<td>Complexity, Fault Tolerance, Horizontal Complexity, Memory Usage, Responsiveness, Security, Testability, Throughput, Understandability, Vertical Complexity</td>
</tr>
<tr>
<td>Quality Factors</td>
<td>Configurability, Correctness, Flexibility, Maintainability, Modifiability, Performance, Reliability, Reusability, Safety, Scalability, Usability</td>
</tr>
</tbody>
</table>
created using Hugin software. This software allows the variables and their relationships to be mapped visually as a BBN. It also allows for probability of variables to be calculated based on the provided evidence. The SABBNet model was used to assess the quality of case studies. Based on the entered evidence regarding architectural attributes SABBNet provided Output for quality attributes given in Table 1 - Software Architecture Assessment Belief-Network Variables. Based on the evaluation of this relatively simplistic network it can be concluded that, it is possible to make qualitative assessment about an architecture that can rival with expert assessments. Meaningful assessment can be made about the quality of software architecture even with limited information available, using Bayesian Belief Networks.

4.2 A Critique Software Defect Prediction

Predicting software quality has always been an important goal for development firms and substantial work has been done on statistical and metric models in this field. Software quality can be predicted based on the presence of defects in a system. These are classified as follows:

1. Predicting the number of defects in the system
2. Estimating reliability of system in terms of time to failure
3. Understanding the impact of design and testing processes in defect counts and failure densities.

Defects can be predicted by use of complexity or size metrics but these provide a distorted view. Therefore a different techniques is required that addresses the limitations of statistical and metric based prediction models. Although this paper discusses other techniques of software defect detection, the emphasis in this text is to detail the use of Bayesian Belief Networks, in predicting software defects. The rational behind using BBN for defect prediction is its capability to represent probabilistic relationships among variables. BBNs combine reasoning under uncertainty and the advantages of intuitive visual representation with sound mathematical basis in Bayesians probability. As all defects are not the results of project complexity and are influenced by other factors. Some of these are listed below that explain the presence of defects in a program:

- Difficulty of the problem
- Complexity of designed solution
- Programmer/analyst skill
- Design methods and procedures used.

A sample BBN has been constructed with variables that represent major influences in injecting software defects. These variables includes; problem complexity, design effort, design size (KLOC), defects introduced, testing effort, defects detected, defects density at testing, residual defect count and residual defect density. These variables were modeled using the Hugin Explorer tool and provided an evidence data set. As a result the system displays predictions as a histogram based on the entered “facts”. The model propagates these facts and predicts the introduced defects, detected defects and the defect density statistics.

Figure 2 – Defect Prediction Model

4.3 BBN for Software Suitability Estimation

This paper presents the software quality model which is based on Bayesian Belief Network (BBN) to predict the software quality in the early stages which is essential for the project development team. Predicting the high risk components earlier in the development cycles is useful as their impact can be minimized. This model captures various software quality factors like development team skills, software process maturity, and software problem complexity and provides reliable forecast of the end quality of the software being developed in terms of product suitability. The relationship between these factors and the elements of the software suitability is presented using BBN. Modeling of the software quality involves the current estimation of the development effort and prediction of the quality of the delivered software product. This Software quality model basically provides the projection of the quality of the software product using the software development life cycle.

The software quality model uses the BBN which is a directed acyclic graph consisting of nodes and edges where node represent the random variables and edges represent the relationship between the nodes. Figure 3 shows the high level structure of software quality model using BBN to predict the software quality. Software quality model consist of three inputs and one out. The inputs to the models are rating of development team’s capability, maturity of the project’s development process and the complexity of the software problem. The output of the model is the final software product quality.

Development Team Capability is determined in two ways. First way is to identify the individual team member skills and experience and then assigned some rank (1, 2, 3 or 4). Second way is to provide a method for consolidating the various individual capabilities into a team capability.
Software Process Maturity is measure using ISO/IEC 15504 which is comprised four different process categories that address four distinct areas within a software development project which are project management process, engineering processes, support processes and customer supplier processes. Software Problem Complexity is captured in requirement, design, implementation and testing phases differently. Software Product Quality uses ISO/IEC 3126 which is international standard for a software product quality which represent the quality of a delivered software product in terms of functionality, efficiency, reliability, usability, maintainability and portability.

Software Quality Model in figure 3, itself relates the inputs and outputs through a set of intermediate nodes which represents the correctness and completeness at each phase of the development life cycle. The internal structure of Software Quality Model is depicted in figure 4. This is basically the Bayesian Network for modeling a quality metric. Each quality metric confirms the correctness and completeness of all the activities associate with each of the four software development life cycle. The correctness and completeness will be determines on the based software quality drivers which are development team capability, process maturity and problem complexity. Figure 5 shows the generic model structure that will be applicable for all these four software development life cycle phases. This model represents that how the correctness and completeness can be determined for each software quality indicators which are depicted in figure 4.

4.4 Software Reliability: BBN and Fault Trees

For the complex systems like mission critical or safety critical systems depends on software to provide added functionality. Different design techniques and development methodologies have reduced the complexity of a software design and increase the analysis. Software reliability is important metric to determine overall system reliability. This paper uses Bayesian Belief Network (BBN) and fault trees to estimate the software reliability and presents the dependences between the software engineering processes and software reliability and proves that BBN and fault trees and can improve the software reliability analysis in the complex systems.

BBN provide a robust probabilistic framework to evaluate the impact of evidence on uncertain outcomes. BBN network consist of number of nodes and graphs where nodes represent the random variable with some probability distribution values whereas edges represents the connection relationships between the nodes. Fault trees describe the combination of events which causes the system failure. In the hierarchal combination of BBN and fault trees, fault trees are used to compute the probabilities of root node for
BBN and BBN are used to compute the probabilities of the basic even in fault trees. Figure 6 shows the Bayesian Belief Network relating processes of software engineering life cycles with the reliability of the software.

The basic idea is to integrate the software standards and development procedures into a BBN to meet the criteria of all the processes in the software engineering cycle. Figure shows the three models which presents the connection of software engineering processes with the initial computed reliability of the software and in the result it generates the new estimated net reliability. If there are not dependencies between the software processes and the software reliability and the overall reliability is equal to initial reliability value and software engineering process is excellent so in that case the reliability will not be more then this and this will be the upper limit but if there is dependencies then the reliability may reduce for any level of the software engineering process.

5 Discussion and Analysis

We have decomposed all aspects of software development into four broad areas and have applied BBN-based techniques in each area with all eventually contributing to the quality of the software product. The following sections are in somewhat of a hierarchy with project risk management encompassing the allocation of resources followed by the lifecycle that focuses on the development process, software reliability on the architecture and design of the system and finally defect detection, that is software quality.

6 Software Project Risk Management

6.1 Initialization

This area pertains with the use of BBNs to minimize software project risk and improve resource allocation. BBNs should continuously assist the decision-making process during the project using feedback so that problematic areas can be dynamically detected and adjusted. The cause-effect relationships in BBNs allow for sources of risks to be identified easily. Also, the ability of BBNs to allow for the introduction of new evidence can improve any iteration to produce estimates of greater reliability. Adapting the risk-management tasks recommended by the IEEE Standard 1540 (2001) and [11], we arrive at the following procedure for the entire software development process:

1. Initialization
2. Maintaining a Project Risk Profile
3. Performing Risk Analysis and Monitoring
4. Performing Risk Treatment

Also, we categorize all the factors that can affect software quality and project risk into two broad groups:

1. Organization-related factors: Factors that involve an organizations risk culture and practice, management experience and capability as well as process maturity.
2. Project-related factors: Factors that involve the nature of the project, complexity, design, architecture, etc.
6.2 Organization Related Factors

The BBN in [11] can be modified to include factors mentioned in [5] for greater breadth. Additionally, we need to modify our definition for Design Quality. Design Quality with respect to our goals of generalizing project-related factors will pertain to not only the design of the components, but the overall design of the software system. That is, it shall include the software process lifecycle, the architecture of the different modules as well as the defect related attributes collected as a result of testing.

6.3 Software Project Quality

Most of the research on Bayesian belief networks for software development deals with predicting the residual defects. Residual defects are the bugs that go unnoticed and are discovered after a product is in production. The later these defects are caught, the higher the cost of fixing them. Hence, it is essential for the success of any software project, that defects be caught as early as possible in the software life-cycle, with appropriately allocated resources for testing efforts.

The BBNs in [9,10] in Figure 2, shown the Defect Prediction Model, accurately predicts defect densities and combines evidence from testing data to provide an effective method to predict the probability of defects within the software. The authors in [2] propose the detection of defects in a cyclic fashion, with defects carrying over to each phase of the software project which is inherent in any kind of software development process. This process can be seen in Figure 9 – A Dynamic Bayesian Belief Networks Life Cycle. The BBN and software metrics in [10] combined with the methodology proposed in [2] can result in a defect detection system that is of greater consistency and accuracy.

7 Conclusion

This paper proposed a holistic scheme of developing software with the help of Bayesian belief networks. A comprehensive literature survey in this area was done with a wide-ranging approach to measuring software and project attributes. A BBN-based procedure was adapted to continuously monitor project progress, project risk and assist managers to dynamically adjust resource allocation. The merits of BBNs are they provide visible and repeatable decision-making support under uncertainties. Other BBNs that predicate attributes of software process, reliability and quality were incorporated into our scheme as well. BBNs are advantageous over software metrics due to their ability to consider all diverse factors affecting the problem. Using static analysis, we have shown that the direction of incorporating BBNs in overall software project management and development is promising. Further application of our approach to realistic on-going projects using dynamic experimentation is desired.

Acknowledgement

The Author would like acknowledge the support of King Fahd university of Petroleum and Minerals.

8 Reference


Continuous Quality Assurance for Performance Degradation in Evolving Software Systems and MYSQL QOS

Mr.J.Prabhu 1, Dr.N.Malmurugan 2, Prof.G.Gunasekaran 3
1. Research Scholar, CSE Dept Sathyabama University, Chennai, India, jprabhu@rediffmail.com.
2. Senior Consultant, Satyam Computer Services Limited, Bangalore, India, n_malmurugan@yahoo.com
3. Professor, Dept. Of IT, Panimalar Engg College, Chennai, India, gunaguru@yahoo.com

Abstract
The developers in software system are often use highly configurable performance-intensive of software. The “regression testing” that ensure their modifications do not adversely affect their software’s performance across its large configuration space followed by unreliable extrapolation from these results to the entire configuration space. As a result, many performance bottlenecks escape detection until systems are fielded. It executes formally designed experiments to identify an appropriate subset of configurations on which to base the performance-oriented regression testing.
1. It allows developers to economically verify key assumptions during process execution.
2. It integrates several model-driven engineering tools and software architecture to make process configuration and execution much easier and less error prone.
3. We evaluate this process via several feasibility studies of three large, widely used performance-intensive software frameworks. Our results indicate that this screening can detect performance degradation in large-scale software systems. This presents a new quality assurance process called “Flexible Effects Screening”.

This paper provides several contributions to the study of Distributed continuous QA (DCQA). First, it shows the structure and functionality of Skoll, which is an environment that defines a generic around-the-clock QA process and several sophisticated tools that support this process. Second, it describes several MYSQL processes built using the Skoll environment. Third, it presents two studies using Skoll: one involving user testing of the MYSQL and another involving continuous build, integration, and testing of the ACE+TAO communication software package.

KeyWords: Quality assurance, performance-oriented regression testing, software architecture design, Quality Of Service,

Introduction
The quality of service (QoS)[1][2][5] of many performance-intensive systems, such as scientific computing systems and distributed real-time and embedded (DRE) systems[1]. The developers on platforms using developer-generated input workloads, depends heavily on various environmental factors. QA teams have extensive knowledge and unrestricted access to the software.

For example, dependencies include the specific hardware and operating system on which systems run, installed versions of middleware and system library implementations, available language processing tools, specific software features that large (and often dynamically changing) space.

The shortcomings of QA efforts are well-known it includes

(1) Increased QA cost and schedule
(2) Misleading results.

When increasingly larger QA task spaces are coupled with shrinking software development resources, it becomes infeasible to handle all QA in-house. For instance, developers may not have access to all the hardware, OS, and compiler platforms on which their software will run. In this environment, developers are forced to release software with configurations that have not been subjected to extensive QA. Moreover, the combination of an enormous QA task space and tight development constraints means that developers must make design and optimization decisions without precise knowledge of the consequences in fielded systems. To address the challenges described above, we have developed a collaborative research environment called Skoll [9] whose ultimate goal is to support continuous, feedback-driven processes and automated tools to perform QA around-the-world. Skoll[5][9] QA processes are logically divided into multiple tasks that are distributed intelligently to client machines around the world and then executed by them. The results from these distributed tasks are returned to central collection sites where they are merged and analyzed to complete the overall QA process.

In prior work, we created a prototype DCQA[5] support Environment called Skoll that helps developers create, execute, and analyze their own DCQA processes. To facilitate DCQA process development and validation and verification, we also developed model-driven engineering tools for use with Skoll. We then used Skoll to design and execute an initial DCQA[5] process, called flexible effects screening. whose goal was to estimate performance efficiently across all system configurations called configuration space. Flexible effects screening borrows ideas from statistical quality improvement techniques that have been applied widely in engineering and manufacturing, such as Exploratory Data Analysis, Robust Parameter Design, and Statistical Quality Control.A central activity of these techniques is to identify aspects of a system or process that contribute substantially to outcome variation. We use similar ideas to identify important configuration options whose settings the distribution of performance across all configurations by causing the majority of performance variation. Evaluating all combinations of these important options for randomizing. Thus provides an inexpensive but reliable estimate of performance across the entire configuration space. Although our initial work
on flexible effects screening presented in showed promise it also had several limitations. For example, the definition and execution of the process had many manual steps. To improve this we have extended and better integrated several model-driven engineering (MDE) tools[3], including the Options Configuration Modeling Language (OCML), which models configuration options and interoption constraints, and the Benchmark Generation Modeling Language (BGML), which models the QA tasks that observe and measure QoS behavior under different configurations and workloads. These MDE tools[3] precisely capture common and variable parts of DCQA processes and the software systems to which they are applied. They also help reduce development and QA effort by generating configuration files and many other supporting code artifacts needed to manage and control process execution across heterogeneous computing resources. Another limitation with our initial main effects screening process was its dependence on strong and untested assumptions regarding the absence of interactions among certain groups of options. If these assumptions do not hold in practice, our results could be wildly incorrect QA task space. Performance-intensive systems, such as the ACE+TAO+CIAO[2][1] QoS-enabled middleware, provide a range of configuration options that can be used to tune its behavior. To be effective, DCQA processes must keep track of these options, in addition to other environmental information, such as OS platform, build tools used, and desired version numbers. This information is used to parameterize generic QA tasks and aids in planning the global QA process, e.g., by adapting the process dynamically and helping interpret the results.

**PERFORMANCE-ORIENTED REGRESSION TESTING**

As software systems change, developers often run regression tests to detect unintended functional side effects. Developers of performance-intensive systems must also be wary of unintended side effects on their end-to-end QoS. To detect such performance problems, developers often run benchmarking regression tests periodically. However, in-house QA efforts can be confounded by the enormous configuration space of highly configurable performance intensive systems, where time and resource constraints severely limit the number of configurations that can be examined. Our earlier experience with applying Skoll to the ACE+TAO[9] middleware found that only a small number of default configurations are benchmarked routinely by the core ACE+TAO development team, who thus get a very limited view of their middleware's QoS. Problems not readily seen in these default configurations therefore often escape detection until systems based on ACE+TAO are fielded by end-users.

This section describes how we address this problem by using the model-based Skoll environment to develop and implement a new hybrid DCQA process called Flexible effects screening. We also describe the formal foundations of our approach, which is based on design of experiments theory, and give an example that illustrates key aspects of our approach.

The Flexible effect of option a, PE(a), is

$$PE(a) = z(a^-) - z(a^+)$$

where $z(a^-)$ and $z(a^+)$ are the mean values of the observed data over all runs where option a is (-) and where option a is (+), respectively.

If appropriate, 2nd-order Flexible can be calculated in a similar way. The interaction Flexible of option a and b,

$$INT(A, B) = 1/2 \{PE(b|a^+) - PE(b|a^-)\} - (2)$$

$$= 1/2 \{PE(alb^+) - PE(alb^-)\} - (3)$$

Here PE(b|a+) is called the conditional Flexible effect of B at the + level of a. The effect of one factor (e.g., b) therefore depends on the level of the other factor (e.g.). Similar equations exist for higher order effects. Once the effects are computed developers will want to determine which of these effects are important and which are
not. There are several ways to determine this, including using standard hypothesis testing approaches.

For this paper we opted not to use formal hypothesis tests primarily because they require strong assumptions about the standard deviation of the experimental samples.

**Chi-Square Goodness-of-Fit Test**

An attractive feature of the chi-square goodness-of-fit test is that it can be applied to any univariate distribution for which you can calculate the cumulative distribution function. The chi-square goodness-of-fit test is applied to binned data (i.e., data put into classes). This is actually not a restriction since for non-binned data we can simply calculate a histogram or frequency table before generating the chi-square test. However, the value of the chi-square test statistic are dependent on how the data is binned.

The chi-square test is defined for the hypothesis:

H₀ : The data follow a specified distribution
Hₐ : The data do not follow the specified distribution

Test Statistic: For the chi-square goodness-of-fit computation, the data are divided into $k$ bins and the test statistic is defined as

$$\chi^2 = \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i}$$

where $O_i$ is the observed frequency for bin $i$ and $E_i$ is the expected frequency for bin $i$. The expected frequency is calculated by

$$E_i = N(F(Y_{i+1}) - F(Y_i))$$

In both commands above, $Y$ is the frequency variable. If one X variable is given, Dataplot assumes that it is the bin mid point and that bins have equal width. If two X variables are given, Dataplot assumes that these are the bin end points and that the bin widths are not necessarily of equal width. Unequal bin widths are typically used to combine classes with small frequencies since the chi-square approximation for the test may not be accurate if there are frequency classes with less than five observations.

**Related Distributions**

Probability distributions are typically defined in terms of the probability density function. However, there are a number of probability functions used in applications. *Probability Density Function* For a continuous function, the probability density function is the probability that the variate has the value $x$. Since for continuous distributions the probability at a single point is zero, this is often expressed in terms of an integral between two points.

$$\int_a^b f(x)dx = P_r[a \leq X \leq b]$$

For a discrete distribution, the pdf is the probability that the variate takes the value $x$.

$$f(x) = P_r[X = x]$$

The following is the plot of the normal probability density function.

The cumulative distribution function is the probability that the variable takes a value less than or equal to $x$. That is

$$F(x) = P_r[X \leq x] = \alpha$$

For a continuous distribution, this can be expressed mathematically as

$$F(x) = \int_{-\infty}^{x} f(\mu)\,d\mu$$

For a discrete distribution, the cdf can be expressed as

$$F(x) = \sum_{i=0}^{x} f(i)$$

The following is the plot of the normal cumulative distribution function.
SURVIVAL FUNCTIONS

Survival functions are most often used in reliability and related fields. The survival function is the probability that the variate takes a value greater than x.

\[ S(x) = Pr[X > x] = 1 - F(x) \]

The following is the plot of the normal distribution survival function.

For a survival function, the y value on the graph starts at 1 and monotonically decreases to zero. The survival function should be compared to the cumulative distribution function.

INVERSE SURVIVAL FUNCTION

Just as the percent point function is the inverse of the cumulative distribution function, the survival function also has an inverse function. The inverse survival function can be defined in terms of the percent point function.

\[ Z(\alpha) = G(1 - \alpha) \]

The following is the plot of the normal distribution inverse survival function.

As with the percent point function, the horizontal axis is a probability. Therefore the horizontal axis goes from 0 to 1 regardless of the particular distribution. The appearance is similar to the percent point function. However, instead of going from the smallest to the largest value on the vertical axis, it goes from the largest to the smallest value.

MySQL Testing with Skoll

MySQL offers one of the most robust transactional database engines on the market.

One of MySQL’s distinguishing features is that it offers a number of different “storage engines” that are specifically designed to handle a particular application need. One of the most popular engines is InnoDB, which supports transactional database requirements. The InnoDB engine has been GA (production) with MySQL.

Distributed continuous quality assurance test for transactional data is so important, MySQL offers many features that ensure high levels of data concurrency. Unlimited row level locking, multi-version consistent (or “snapshot” as in Oracle) lock-free reads, isolation level control, and instant deadlock detection are all present in MySQL.

There are many other rich features present in MySQL’s transactional storage engine like clustered and standard indexes, distributed transaction control, server-enforced referential integrity, and much more.

MySQL Testing: around the clock, around the world

A test-based distributed process, to test MySQL across a configuration space of about 48 million different
configurations. Skoll test clients connect to our server and receive instructions to build MySQL in a specific configuration. The client then compiles MySQL and runs a set of standard MySQL installation tests. Finally, the client sends a summary of the test results back to our server. There is show_test_results, which gives an account of what we have tested, listing the result of each test, a summary of passed and failed ones, and a further list of failed test for easy review.

The basic principle of test case evaluation is that output resulting from running a test case is compared to the expected result. This is just a difference comparison between the output and an expected-result file that the test writer provides. This simplistic method of comparison does not by itself provide any way to handle variation in the output that may occur when a test is run at different times. However, the test language provides commands for post processing result output before the comparison occurs.

Modeling the QA subtask MYSQL:

We formally model aspects of the QA subtasks and underlying software that will be varied under control of the distributed process. This includes not only process and software configuration parameters, but also constraints among them. To do this, the overall QA process. With the model, Skoll server can intelligently assign test configurations to clients in order to cover the configuration space we developed a general representation with configuration options. The option settings and inter-option constraints. We also developed the notion of temporary interoption constraints to help us reduce configuration space size artificially in certain situations.

Exploring the MYSQL:

The configuration space of a QA process for a performance-intensive infrastructure system can be quite large. Even with a large pool of user-supplied resources, brute-force approaches may be infeasible or simply undesirable. Consequently, we developed techniques to explore/search the configuration space. We developed a general search strategy based on uniform sampling of the configuration space and supplemented it with customized adaptation strategies to allow goal-driven process adaptation.

To ease the task of testing, there are a few routines. The most important one, and perhaps the only one that you may need in setting a test, is log test which expects four parameters

**Difference Between Precision And Accuracy.**

The board on the upper right shows darts that are precise but not accurate. They are closely grouped, so the thrower has precision, but it is not very accurate because the darts didn’t even hit the board. The board on the lower left is an example of accuracy but poor precision. The darts are very close to the center, so the thrower is getting close to what its aiming at, but because they aren’t closely positioned, the precision is off. The board in the lower right is a perfect match of precision and accuracy. The darts are closely grouped and on target.

The software and MYSQL we test needs to be precise or accurate depends much on what the product is and ultimately what the development team is aiming at Distributed Continuous Quality Assurance.

Resource availability:

Since QA subtasks are assigned to remote machines, volunteered by end users. Increased QA cost, schedule and misleading results when the test-cases are input workload to the software version and platform at the developer’s site differ from those in the field.

These problems are magnified in performance-intensive software, such as that found in high-performance computing systems, distributed real-time and embedded systems and the accompanying systems software (e.g., operating systems, middleware, and language processing tools).

Product quality:

The Metric used for product quality (QUAL)[8] is defects which were measured as the total number of defects that escaped to the customer and were detected during the first three month of production use of the software. A period of three months is used, as it is typically the warranty period of newly developed software, and the defect date for the first-three months is generally tracked by software development organizations. A low value of QUAL indicates higher quality software.

Although defect density or its reciprocal, which normalize defects for size, have been used as a measure of quality[6],[7]
We use total number of defects instead due to high correlation between defect density and size. When we do log transforms of variables in our analysis,

\[ \ln(\text{defect-density}) = \ln(\text{defect/size}) = \ln(\text{defects}) - \ln(\text{size}) \]

We find that \( \ln(\text{defect-density}) \) has a correlation, since this correlation is very high and highly significant, using quality and size together as independent variables in the same model causes multicollinearity problems. Our approach of using total number of defects as a measure of quality is in line with MYSQL.

**FAULT VISIBILITY**

To detect an implementation error, a test case must cause an internal fault to propagate to a visible output. Consider the following fragment of a state machine description:

\[
\begin{align*}
\text{next}(t) & := \text{case} \ldots f(i) : v; \ldots \text{esac; } \\
\text{next}(o) & := \text{case} \ldots g(t) : w; \ldots \text{esac; }
\end{align*}
\]

In the example, \( i \) is an input variable, \( t \) is an intermediate variable, \( o \) is an output variable. Suppose that mutation replaced the formula \( f(i) \) with \( f'(i) \). In the case of direct reflection, the corresponding mutant formula is

\[ AG (f'(i) \rightarrow t = v) \]

Often, the model checker will find a counterexample that will show inconsistency in the intermediate variable \( t \) but not in the output variable \( o \). Such a test is of little value.

We proposed two methods [6] to guarantee that tests cause detectable output failures. The first method, in-line expansion, uses only the reflections of the transition relation involving output variables. In these temporal logic formulas, any internal variable is replaced in-line with a copy of its transition relation. This substitution is repeated until the formulas are comprised

**Solution approach: for MYSQL using DCQA.**

To address these challenges, we are conducting a collaborative research project called Skoll. Skoll is a general, continuous, feedback-driven process, supported by automated tools to carry out around-the-world, around-the-clock QA. Our approach divides QA processes into multiple subtasks that are intelligently distributed to client machines around the world, executed by them, and their results returned to central collection sites where they are fused together to complete the overall QA process. We built a configuration space model with MySQL compile- and run-time configuration variables. With the model, Skoll server can intelligently assign test configurations to clients in order to cover the configuration space.

**REFERENCES:**

Practically Relevant Quality Criteria for Requirements Documents

T. Simon, J. Streit, and M. Pizka
itestra GmbH, Ludwigstr. 35, 86916 Kaufering, Germany

Abstract. This paper presents common weaknesses of requirements documents from commercial software projects that frequently cause problems in practice. Many documents contain extensive, unstructured or even superfluous descriptions of details. At the same time, they lack a thorough description of the aim of the software system or one of its features. Such documents are difficult to read and to work with and they unnecessarily restrict possible solutions.

We argue that two quality criteria must be considered in addition to common criteria. First, requirements documents should be based on a sound refinement of the main goals to concrete requirements. Secondly, they should follow the principle of minimality, i.e., no more details than necessary.

In order to gain improvements in projects in practice, quality criteria must be communicated to authors. For this purpose, we propose two visualizations, which describe the information structure of the document in a technology-free and domain-independent manner.

Keywords: Software Requirements, Requirements Documentation, Quality Assurance, Refinement, Minimality.

1 Lost in details

1.1 Quality assurance for requirements documents

It is widely recognized that requirements engineering plays a crucial role in a software project and that it is—at the same time—a very difficult task [1, 2]. The quality of requirements documents and their adequacy for further use in the project must therefore be assured and—if necessary—be increased.

Characteristics of a good requirements specification according to IEEE [3] are: correct, unambiguous, complete, consistent, ranked for importance and/or stability, verifiable, modifiable and traceable.

Much effort has been spent in the scientific community on the aspects correctness, consistency and unambiguous-
Table 1. Use Case describing trivialities

<table>
<thead>
<tr>
<th>Use Case: Editing a data item</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1.</strong> The actor chooses one data item from the list.</td>
</tr>
<tr>
<td><strong>2.</strong> The actor chooses “Edit item”.</td>
</tr>
<tr>
<td><strong>2a.</strong> The actor enters values and confirms.</td>
</tr>
<tr>
<td><strong>2b.</strong> The actor enters values but does not confirm.</td>
</tr>
<tr>
<td><strong>2ba.</strong> The actor confirms and thus approves storage of the changes.</td>
</tr>
<tr>
<td><strong>2bb.</strong> The actor negates and thus disapproves storage of the changes.</td>
</tr>
</tbody>
</table>

Finally we introduce two visualizations of the information structure in a requirements document that make it easier to explain the quality criteria and possible weaknesses to authors.

2 Typical weaknesses of requirements documents

In the following we present weaknesses of requirements documents we have frequently encountered in commercial projects. The examples are taken from an in-depth analysis of information system requirements documents from six projects, involving four companies from three different domains, and altogether comprising approximately 1,500 pages. For confidentiality reasons the examples were slightly modified; their core however remains the same. Note that some of the described weaknesses are related, so a sharp distinction is not in all cases possible.

Trivialities. Almost all documents contain lengthy descriptions of commonly known issues.

In table 1, a Use Case scenario description taken from one of the analyzed documents can be seen that mainly describes how editing a data item in the system can be aborted. This is an example of a common feature which in most cases does not have to be specified further in detail. Therefore the lengthy description could be easily summarized with one sentence “The scenario can be aborted by the user at any time. The system asks for confirmation”. Trivialities may appear just as a waste of effort but not really as harmful. However, they artificially inflate the size of requirements documents. Lengthy documents are not only harder to read and to maintain, but are also susceptible to inconsistencies and errors, as it is more difficult to retain the overview. It thus becomes more likely that information added to the document conflicts with already existing information.

Information out of scope. We repeatedly identified sections of requirements documents containing information that does not add any value to the description of the system to be build.

In one case, the document contained the following statement:

Department X is responsible for entering the data into the system.

This information is superfluous. The business process had already been described based on roles. The assignment of responsibilities to departments should not be part of a software requirements document. Even worse, the sentence triggered a discussion between the departments about the responsibilities, as they feared that such a statement in an approved document could be used to shift responsibilities from one department to the other.

Information out of scope also unnecessarily increases the size of the document, and we have seen in the previous section that this is to be considered problematic. Additionally, such information may lead to irritations and unnecessary struggles.

Thinking in solutions. Another frequent defect is the description of solutions or products where a description of the problem to be solved would be appropriate.

The following excerpt represents this flaw:

The user shall be notified when another user attempts to edit the same data. The server must inform the client software using a push-mechanism.

The second sentence clearly describes an implementation decision. What the author actually wanted to express is that no user interaction shall be needed for getting the notification. Using a push-mechanism in the client-server communication is one but not the only solution, and the choice should normally be made during software design (unless there is another reason to prefer one architecture, for example an existing software landscape).

Describing a solution obscures the actual problem to be solved to the reader. This is problematic if the solution turns out to be difficult, costly or impossible to implement, or if there are better solutions the author had not thought of. Provided with a description of the problem, a software engineer who reads the requirements document can propose alternatives. Without such a description, he or she can only guess
what the intentions of the author were. Needless complex, clumsy and expensive software is the result of this weakness.

**Pinpointing details.** Often, authors pinpoint details that unnecessarily complicate the implementation, i.e., an equally well or better suited solution would be possible at equal or lower cost. This weakness is related to “Thinking in solutions”, however, its root is not the missing problem description but just a needless detailing.

An example that represents this common flaw is hidden in the following description of a data field taken from one of the analyzed documents: 

*There are 5 different values to choose from.*

A closer investigation revealed that only two values were currently known and in use. The other three were planned as placeholders for later extensions. The restriction to five values is however completely unnecessary, as this field would probably be implemented as an enumeration of $n$ values anyway, and adding three new values entails the same cost as adding ten of them. Furthermore, additional code is needed to limit the number of entries to five. Instead, it would have been better to state that two values were currently known and that an extension should be possible. This way, a solution that increases the extensibility of the system and decreases its development cost at the same time could have been implemented.

**Lacking rationale.** Many requirements documents fail to describe what shall be achieved with the software or one of its particular features. Note that in contrast to the “Thinking in solutions” defect, the concrete requirements in these documents may be described at an appropriate level of detail.

The following example extracted from one of the analyzed requirements documents represents this frequent weakness:

*The content of the database table SHARESMOUNT has to be exported to a Microsoft Excel spreadsheet.*

The context and the rationale explaining why spreadsheet sheets at all or why spreadsheet sheets of this particular vendor should be used was lacking. The developers later faced the problem that Microsoft Excel spreadsheet sheets can only contain up to 65536 rows and the number of rows in the database table exceeded this number. Because the document also did not mention what the resulting spread sheet is used for, the developers were not able to implement an appropriate solution based on the available information. Due to a scheduled meeting with the stakeholders it turned out that the spread sheet was determined only to be printed and handed to the analysts as a report. If this information had been contained in the requirements document, they would have been able to develop an alternative solution, such as a PDF export, providing the same or an even better result.

Lacking rationale in the requirements documentation is problematic for several reasons. First, it usually makes the document more difficult to understand for persons not familiar with the subject. Secondly, it makes it much harder for a software engineer to find alternative solutions when a particular requirement turns out to be costly, difficult or impossible to implement.

### 3 Practically relevant quality criteria for requirements documents

The weaknesses presented in the last section are not captured sufficiently by common quality criteria for requirements documents and they are usually not in the authors’ focus, consequently causing severe problems in later project phases.

We therefore argue that additional criteria have to be considered for judging and improving the quality of requirements documents, namely adherence to a refinement relation between requirements and minimality.

#### 3.1 Root-based Refinement

The “Lacking rationale”, “Thinking in solutions”, and also the “Pinpointing details” weaknesses have in common that the requirements document fails to explain why a certain requirement is demanded. This *why*—the rationale—helps readers to get an understanding on what the software is meant for, as explained for the weakness “Lacking rationale”, and to maintain the overview more easily. Also, it allows software engineers to propose alternative solutions in line with the main goals when obstacles appear during further analysis and implementation.

We therefore demand to include the rationale for all requirements in the document. The most abstract rationale is the goal for which the software system is being developed. Any concrete requirement to the software system must in the end serve this overall goal, and hence should be an iterative refinement of it, i.e., adhere to a refinement relation of requirements. Refinements restrict the set of possible implementations: for example, while the requirement “The system shall allow users of company X to compose emails.” is fulfilled by any email software, the refinement “Users want to compose emails while traveling.” rules out a web-based solution.

The refinement relation should be included entirely (i.e., with the main goal as root) in the document, and it should be traceable, e.g., by cross references, links or an adequate document structure. Besides improving the understandability of the document, the goal description and its refinements also help the authors to stay focused: sections that describe solutions instead of problems can be identified, as their refinement relation will not be sound from the abstract goal to
a concrete requirement.

3.2 Minimality

A second issue that several of the above weaknesses have in common is that the requirements document contains too much information. Requirements documents need to specify what is required for developing the software as requested by the stakeholders\(^1\). In particular, they should not leave for implementations that do not fulfill the stakeholders’ demands. However, any additional information possibly impedes the development of the software, as it makes the document longer and may rule out possible (and potentially more efficient) implementations, as explained for the “Trivialities”, “Information out of scope” and “Pinpointing details” weaknesses.

We therefore affirm that the minimality of a requirements document—in particular the avoidance of superfluous details—is an important quality criterion, which is, unfortunately, often neglected.

Demanding minimality and the inclusion of the requirements rationale simultaneously may seem contradictory. However, there are good reasons for including the rationale in the document, as we have explained above. Therefore this information is not to be considered superfluous, while needlessly detailed descriptions and information out of scope are. Furthermore, the refinement relation described above can be helpful for obtaining minimality: if the information on a certain level of detail describes an aspect accurately enough to allow no wrong systems to fulfill the specification, it is unnecessary to refine it further. In other words, if the question “Does the goal description already express the needs of the stakeholders sufficiently?” can be positively answered, there is no need for a more detailed description.

3.3 Relation to other quality criteria

The proposed quality criteria do not only lead to a requirements document that can be efficiently used in the development process but also have a positive impact on the characteristics of good requirements specification according to IEEE [3].

With the overall goal and its refined goals included in the document, it is more likely that incorrect requirements will be detected in an early phase of the project, as they might be in contradiction with one of the goals and thereby visible to an attentive reader. The analysis of the rationale using “why”-questions also helps to identify requirements that were missed out and thus to increase completeness, as explained in [10].

\(^1\)In this paper, we do not distinguish different groups of stakeholders with potentially differing requirements.

Furthermore, the modifiability of each requirement is improved, as the rationale leading to the requirement is contained in the same document and can be used to determine the impact of the change.

4 Information structure visualizations

The authors of requirements documents are usually unaware of the shortcomings described above and their consequences. It is thus crucial to not only define quality criteria for requirements documents, but to also develop ways of communicating them to authors in practice.

In the following, we introduce a graphical representation for this purpose. The criteria will be illustrated through visualizations of requirements documents and the systems they specify, the Requirements Information Pyramid (RIP) and the Space of Solutions (SoS).

4.1 RIP and SoS

The Requirements Information Pyramid shown in figure 1 visualizes the refinement structure of the information in a requirements document. Every piece of information in the document is represented by a gray dot. Hence, the gray area is a representation of the information that is actually contained in a document. If the dot is located between the black boundaries the information specifies a characteristic of the system that is demanded by the stakeholders. A gray area outside the pyramid represents unnecessary information that does not influence the implementation of the system and that therefore does not add any value to the description.

The pyramid shape results from the hierarchical refinement of the information in the requirements document. The peak of the pyramid represents the overall goal of the system to be built (level of detail 1). Every further gray area is a more refined and detailed description of the information above it: the middle region states general intentions
and requirements to the system, the lower regions stand for more detailed information on the same issues. We consider the implemented source code (together with additional information about the deployment and the infrastructure the system is running in) as the most detailed description of a system. This information can be found on the very bottom of the pyramid, and is usually not contained in the gray area (i.e., the document) any more.

Figure 2 shows the visualization Space of Solutions. The rectangle represents the space of all possible software systems. Every dot in the rectangle stands for a specific implementation. The areas surrounded by a line enclose sub sets that fulfill the requirements in a document at a certain level of detail. SoS and RIP provide two different perspectives on the same requirements document, as can be seen in figures 2 and 3 for four levels of detail of the requirements documentation (denoted with the numbers 1 to 4). In the first phase only the overall goal is specified and thus the set of possible solutions is only roughly delimited. In the further phases more detailed information about the requirements is added to the document. As a result, the gray area in the RIP becomes bigger and expands to the lower part of the pyramid. At the same time, the set of possible solutions becomes smaller and excludes those solutions that are in conflict with requirement details. When the requirements documentation is completed, only a few solutions remain in the set of solutions (level of detail 3). Finally, an implementation of the software would be represented as a completely filled RIP and a single dot in the SoS (level of detail 4).

4.2 Visualizing weaknesses

In the following, we are going to apply the visualizations to the weaknesses listed in section 2 and provide a RIP and/or SoS for each of the weaknesses. The visualizations represent the information structure of the requirements document in a technology-free and domain-independent manner. Thus, they can be used as the basis for discussions between the stakeholders, domain experts and software engineers about the quality of the document.

Trivialities / Pinpointing details. A document that contains trivialities is visualized in drawing (A) in figure 4. It refines the information contained in the document very deeply. A document in which a single requirement is described on a very fine-grained level of detail is shown in drawing (B). It is obvious that the larger the gray area is the more difficult it is to maintain the overview and to modify the document.

Still, these drawings may represent appropriate requirements documents if the most detailed information is specifically demanded by the stakeholders.

However, if the details are not actual requirements of the stakeholders, they unnecessarily restrict the possible implementations. Figure 5 shows an SoS where—due to over specification—the set of implementations allowed by the document (white) is much smaller than the set of implementations that would satisfy the needs of the stakeholders (dotted border). Hence, the requirements document unnecessarily excludes a large number of implementations (dark stripes), some of which may be cheaper to implement or provide greater flexibility or usability than the solutions specified by the document.

Information out of scope. A representation of a document that contains information beyond the scope initially defined for the document can be seen in drawing (C). This information is a superfluous addition to the requirements document that only lengthens it without adding value for the implementation of the software.
effects of specifying superfluous constraints. 

recommendations [3] or [15]. [16] describes the harmful mentioned in [10, 14], but not at all addressed in the IEEE the further development process. In literature, this issue is tor for the efficient usage of the requirements documents in

describe overall goals, blue use cases describe user goals cases for different levels of abstraction. White use cases within the requirements document.

describe overall goals, blue use cases describe user goals etc. Cockburn emphasizes that all levels are necessary in order to get a complete description. The RIP can thus be seen as a generalization of Cockburn’s color system.

We pinpoint minimalism as a crucial characteristic in requirements documentation and see it as a main success factor for the efficient usage of the requirements documents in the further development process. In literature, this issue is mentioned in [10, 14], but not at all addressed in the IEEE recommendations [3] or [15]. [16] describes the harmful effects of specifying superfluous constraints.

5 Related Work

The definition and refinement of goals has been identified as an essential part of requirements engineering [10]. While these goal-oriented approaches provide methods for requirements elicitation as well as formal and semi-formal models for requirements specification, we focus on the information structure within a (usually textual) requirements document. We highlight that the root of the refinement must be the overall business goal, and that all refinement steps must be included in the document.

The refinement of goals to detailed descriptions of requirements is related to the traceability of requirements [11, 12]. These articles investigate the ability to follow the lifecycle of a requirement, all the way from the requirements description to a specific code line. Our concept does not link a requirements descriptions with a code fragment but with more abstract information about the requirement within the requirements document.

Cockburn [13] distinguishes differently “colored” use cases for different levels of abstraction. White use cases describe overall goals, blue use cases describe user goals etc. Cockburn emphasizes that all levels are necessary in order to get a complete description. The RIP can thus be seen as a generalization of Cockburn’s color system.

We pinpoint minimalism as a crucial characteristic in requirements documentation and see it as a main success factor for the efficient usage of the requirements documents in the further development process. In literature, this issue is mentioned in [10, 14], but not at all addressed in the IEEE recommendations [3] or [15]. [16] describes the harmful effects of specifying superfluous constraints.

6 Conclusion

This paper demonstrates that a requirements document may, although providing a complete and consistent specification of a software, contain flaws that render it nearly unusable in subsequent project phases. Issues such as the sheer size of a document, the restrictions it imposes on the possible implementations and its readability are widely ignored in scientific research, although they pose serious threats to a project in practice. We have argued that following a root-based refinement approach and keeping a focus on minimalism improves the quality of requirements documents, which is a prerequisite for their helpful use. As we have experienced in our daily work, the visualizations RIP and SoS with their technology-free nature serve as a simple and effective basis for discussions between stakeholders, domain experts and software engineers with the aim to identify, explain and resolve weaknesses in requirements documents.

References


[9] Luisa Mich, Mariangela Franca, and Pierluigi Inverardi. Market re-


Safe Runtime Reconfiguration in Component-Based Software Systems

Saleh Alhazbi¹, and Aman Jantan²
¹Computer Science Dept., Qatar University, Doha, Qatar
² School of Computer Science, Universiti Sains Malaysia, Penang, Malaysia

Abstract - Long life and critical software systems require high availability. Therefore, their reconfiguration should be achieved dynamically during their running time. This type of updating is not safe as static update because it lacks test phase in order to validate the modifications. This paper discusses different sources of risk with runtime reconfiguring, it presents Message-based Interaction in Component-based System (MICS) framework that facilitates updating the system online, and ensures system safety during and after system modification.

Keywords: Component-based, Dynamic Updating, Message-based Interaction, Dependency.

1 Introduction

Traditionally, software reconfiguration requires shutting down the system, applying new modifications, and restarting the system. This approach is not suitable for long-life and high-available systems, such as banking, telecommunication, air traffic controls systems, which require reconfiguration without service interruption. Runtime reconfiguration or dynamic updating implies modifying the system while it is running. In component-based software systems, dynamic update includes adding, deleting, or replacing one of its components while at least one component is running. Several approaches [1-4] have been proposed for developing dynamically component-based systems. The center attention of those techniques was on how to enable dynamic update without much attention to system safety during and after updating. We define system safety in context of dynamic updating as system's ability to work consistently during updating operation and after that. The remaining of this paper is organized as follows. In section 2, we discuss the sources of risk when updating the system dynamically. In section 3, MICS framework is presented for composing component-based systems. Section 4 shows how MICS framework addresses the safety problem with runtime reconfiguring. Conclusion and future work is presented in section 5.

2 Risk of runtime reconfiguration

The main goal of dynamic updating is to keep the system running without interruption. Therefore, it would not be acceptable that updating the system online leads to erroneous situation where system might stop or malfunction. The threat with updating a running system is due to lack test phase in order to verify changes to the system. Consequently, this updating may affect system correctness. There are four different sources of risk when updating the system dynamically:

2.1 Interrupting running process.

Generally, updating software online cannot be achieved at any arbitrary time because it might interrupt some process that would lead the system to inconsistent state. Well-timing point is needed for any approach to safely update the system online. In previous work, two different approaches are proposed to define suitable time to perform dynamic update:

1) the operation of update is postponed until the target module is inactive[5-8]. Moreover, the target module also should not be disturbed during the time of updating, so the other modules that dependent on the target one should be passivated in order not to send any request to the target module[9]

2) The update point is predefined in the system itself. This point either be found at runtime such as when new object is created in object-oriented programming[10] or it maybe specified in the code [11, 12] where dynamic updating is only performed when execution reaches a safe updating point that defined when coding the program.

2.2 Breaking components' dependencies.

Updating a component of the system should preserve other components' dependencies on this replaced one. For example, if component C1 has method M1 that is used by other components, replacing C1 with a new one that does not have M1 will cause a problem and might lead the system to crash. This type of errors is identified as interface faults [13, 14] or interface incompatibility [11]. In[15] an algorithm was proposed to validate new component's compatibility with the system. The compatibility is checked by calculating differences between old and new version where the result is one of five possibilities:

1. New component has the same interfaces as the old version which means same methods names and parameters, in this case the new version is compatible and consequently replacing that component is valid.

2. New component has methods as the old one (semantically the same with same parameters) but the
name of methods are different, in this case a glue code is needed to call methods in the new name through a method with the old name.

3. New component has same methods as old one but moreover it has extra functionalities. Those new functionalities are not used by the system. In this case, the component is considered compatible and replacement is valid.

4. New component missed functionalities, but not used by system. In this case also the replacement is valid and component consider compatible.

5. All other cases, the new component is considered not compatible.

Two approaches to handle incompatibility of interfaces: either Using adapters that serve as a translation layer between the new version and its dependents or by updating the dependents before update the target component[16].

2.3 Loosing the state of old version.

When replacing a component online, the new version should start from the point, which the old one stopped at. Therefore, the state of old component should be transferred to the replacing one. The problem here is how to represent component state in old and new version, and how to map old state to the new one that might be different type. Usually, it is assumed there is a relationship between old and new version and because such relation mostly depends on semantic knowledge, transferring the state is usually left to the programmer[17, 18]. For example, in object-oriented languages, the execution state is typically represented by the states of all instance variables and state of the runtime stack, when the object is replaced the values of those variables should be mapped in someway to the variables of the new version[19]. Bialek and Jul [20] defines a state transfer function (STF) to map between old version of component and new one. This function must be attached to update request which is consists of tuples { componentName, NewComponentDefinition, STF}[20]. Mapping the state of old version to the new one could be simple as casting int to long or hard as between different objects. In [21] two techniques are proposed to exchange state information between versions:

1) Direct State Transfer: The implementation of the old version is used directly. It is the responsibility of the new version to interpret and convert the state from the previous version; the state transition function depends on both the old and then new version of the component.

2) Indirect State Transfer: in the approach, an abstract representation of the state is created and all implementation specific program states are mapped to this abstract representation. When state transfer occurs, the old version exports its state in an abstract representation which is later used by the new version. The advantage of this approach is that it does not require knowledge of the implementation of the old component version.

2.4 Semantic errors and system malfunction

This kind of risk is related to component's logic. Since there is no test phase when updating the system dynamically, the behavior of the new component, when works with others in the system, can not be anticipated. Semantic errors are hard, if not impossible, to be identified automatically[11, 12]. Actually, this problem is common with static software upgrading, new version breaks exiting behavior. To solve this problem with static upgrading, Cook and Dage[14] proposed a framework that when upgrading a system it keeps multiple versions of a module running. In their framework, an Arbiter is placed between the external system and the component versions. The arbiter invokes all versions and uses Constraint Evaluator(CE) to select a result according to formal specifications of each version's addressed domain. The arbiter returns the result to the main system and logs the statistics on which version produced the result. Similar to the framework of Cook and Dage presented above, Liu and Richardson [22] proposed a specific architecture called Redundant Array of Independent Component (RAIC) that uses groups of similar or identical components to provide dependable services. The group uses services from one or more components in the group to provide services to the application. From the application view, the RAIC looks as single component. Components can be added or deleted to RAIC dynamically. The RAIC controller is responsible of making judgment about the return values from individual components in the redundant array.

3 MICS framework

In this section, we describe our java-based component model, which is utilized by our Message-Based Component-Based System (MICS) framework to develop safe dynamically software systems. In MICS framework, components send/receive messages through a soft bus to provide the functionalities of the system. Additionally, each component is hooked to the soft bus through a connector to facilitate message exchanges. Generally message-oriented pattern of interaction has the following advantages: (1) All dependencies are centralized and no explicit decencies between components which makes component integration easier[23]. (2) It reduces the architecture complexity of the system which means it’s more maintainable and adaptable [24, 25]. (3) Message-based systems are more upgradeable and reconfigurable as new components can be added for satisfying new requirements without changing the basic system architecture [25]. Figure 1 depicts MICS architecture.
3.1 Component Model.

In our model, components represent the essential part of the system. They are the locus of computation and the core providers of system functionalities. They merely services providers and consumers where the communication among them is facilitated by other entities called connectors. We should distinguish between two views of the software component: component type and component instance. The first one is a static piece of software that provides specific functionalities, we call it component type, and the second view is the instance that has run-time existence and state. In our model, components only interact through their interfaces, either provide service to other components or require services from them. Components need connectors to interact with each other, which are defined during composition phase by the integrator. This separation between computations and communications offers loosely architecture. It supports concepts of componentware as the components being easy pluggable and replaceable. Each component has an XML file that describes its interfaces which includes the services provided and required by the component.

3.2 Connectors.

Connectors in our framework are not computation parts of the system. They facilitate components interaction. Each component in MICS communicates with other components in the system through connectors, which hook the component up to the bus. Each connector represents the gateway between the component and the bus. We have two types of connectors Out-port and In-port. Out-port connector masks the services provided by the component, therefore this connector has the same methods as the component behind it. The task of this type is to interpret incoming messages and call the service from the component. On the other hand, out-port connector represents the gateway for the service required by the component.

3.3 Soft bus

Soft bus in our framework is a special component that is responsible of tracking and identifying all components connected to it, so it routes messages from sender components to the target ones. It simulates the concept of using bus with hardware, so the components can be plugged in or out easily.

3.4 Messages

MICS framework has two types of messages: Request message (RQ), and Response message (RS). Every message contains two parts: a message part (such as service required, service arguments), and a control part (such as message ID, message type). The types of messages as follows:
(1) Request message (RQ): this message is sent from a component to another asking for one of its provided services. The message is six tuple <Message type, Receiver, Service, no of arguments, arguments, sender>
(2) Response Message (RS): this message is sent as a successful response to a previous request, it carries the result back to the sender of the request. This message format is five tuple <Message type, Receiver, Result, Sender>, even thought the service might not returns any result, an RS message should send back to the requester component. RS considered as acknowledgment message of finishing the process.
3.5 Component Manager.

Component manager is responsible for tracking all component types in the system. When adding a new type to the system, component manager reads its XML description and add it to the list of system components. This description of components service will be used to check compatibility to other components when creating new dependency between this component and others. Component manager also removes the components from the list when it is not used in the system any more.

3.6 Dependency Manager

Updating the system dynamically requires exploring the effects of this modification on the rest of system's components in order not to lead the system to inconsistent state. Dependency Manager is responsible for ensuring the compatibility between components when creating dependency at the beginning of running the system. When replacing a component with a new version, it assures that new version has compatible interface with others. Analyzing component dependencies and how they are implemented are described in our work [26] with more details.

3.7 Update Manager

Update Manager controls all parts of the framework. When there is a request for updating the system on fly, it first checks the status of the target component. If that component is involved in some process, it waits until the component becomes idle. If the new version has the same interface with old ones, Update Manager sends a message to old component to flush its state, disconnect old one from the bus, attach the new one to the bus, and transfer status to the new version. If the new version has different interface from the old one, it uses dependency manager to track all dependents on this component and requires updating them first in order not to break dependencies.

4 Safety in MICS.

The essential goal of runtime reconfiguration is to allow the system running continuously. Therefore, any dynamic updating technique should preserve system safety, so system will not be crashed or malfunctions. Besides facilitating runtime reconfiguration, MICS frameworks supports safety as it is related to dynamic updating. MICS addresses the four sources of risk described in section 2 as follows:

- In MICS, a component during runtime is in one of the following statuses: 1) Idle: a component in this status can provide or required services. 2) Active: in this status, component is engaged in serving a request. 3) Blocked: when a component is blocked, it neither initiates a request nor provides a service. 4) Semi-Blocked: when a component is in semi-blocked status, it only can serve requests from other components but it cannot initiate a request. Any request to update an active component is postponed until the component becomes idle, then update manager sends a message to block the component. Using dependency manager, update manager also sends message to all its dependents to make them semi-blocked so they will not issue any request to the target one and will not disturb updating process.

- MICS framework keeps track of all dependencies among system's components. When a request is issued for updating the system, update manager consults dependency manager to verify compatibility of the new component with others. If it is incompatible, updating is rejected, so the system keeps running.

- As MICS is java-based framework, the class represents component concept and values of its instants fields represent its status. When a request is issued for updating a component, update manager creates a new instance of the new version. Programmer should provide update manager with small program that gets values of old version, and map them to the fields in the new object.

- MICS framework supports the concept of rollback to old version. When a component is updated, the new object is attached to the soft bus and old version is disconnected. This approach of communication among components provides flexibility to rollback to old version if the new version has logical errors. The decision of bringing old version back to the system is made by system administrator.

5 Conclusion and Future work

Runtime reconfiguring is an approach for updating software systems without shutting down the whole system. In this paper, we have presented four possible threats to system safety. We presented our proposed MICS framework, which is a message based interaction approach for composing component-based systems. MICS supports dynamic updating with focus on system safety during and after updating process. Future work in this area can be developing a visual tool that monitors the status of each component in the system and visualizes the interaction of running components in the system.

6 References

.3 Feng, N., S-Module Design for Software Hot Swapping, in System and Computer Engineering Department. 1999, Carleton University.


.11 Bialek, R., Dynamic Updates of Exiting Java Applications, in department of Computer Science. 2006, University of Copenhagen

.12 Hicks, M., Dynamic Software Updating, in Department of Computer and Information Science. 2001, University of Pennsylvania.


Specifying Correct Protocols of Multithreaded Java-like Programs

Clément Hurlin
INRIA Sophia Antipolis - Méditerranée
2004, route des Lucioles BP 93
06902 Sophia-Antipolis, France

Abstract—In the Design By Contract approach, programmers specify methods with pre and postconditions (also called contracts). Earlier work added protocols to the design by contract approach to describe allowed method call sequences for classes. We extend this work to deal with a variant of generic classes, bounded parallelism, and unbounded parallelism that allows to specify protocols of multithreaded programs. We present the semantical foundations of our extension. We describe a new technique to check that method contracts are correct w.r.t. to protocols. We describe how to generate proof obligations (POs) that represent that method contracts are correct w.r.t. to a protocol. Trying to prove these POs can provide useful feedback to programmers early in the development process.


I. INTRODUCTION

Over the last years, major work has been done towards software verification. Among the variety of methods to verify software, a method of major importance is Design By Contract (DBC) [1]. In the DBC approach, programmers specify methods with pre and postconditions (also called contracts). A precondition specifies what the client must provide at method entry, while a postcondition specifies what is ensured to the client at method exit. Tools for DBC include the Eiffel programming language [2], the Java Modeling Language (JML) [3] for Java, and the Spec# project for C# [4].

The aforementioned techniques for DBC both provide tool support to (1) dynamically check contracts (with Eiffel’s built-in facilities or with JML’s runtime assertion checker [5]) and (2) statically check contracts (with ESC/Java2 [6] for JML or with Boogie [7] for Spec#).

Typically, the DBC approach recommend to write method contracts before implementing the methods. The tools mentioned above, however, are useful to check implementation of methods against their contracts: In the typical development process outlined above, the feedback from static checking tools comes very late in the development process. In this paper, we propose a technique to use static checking much earlier in the development process. To do this, we extend earlier work on specifying and checking method call sequences in JML [8].

A method call sequence is a regular expression indicating what methods can be called on an object and in which order methods must be called. As explained earlier [8], many classes must be used in a specific way by clients: for these classes it is important to be able to concisely express allowed method call sequences. For example, instances of Java’s interface Iterator must call method remove() zero or one time after calling method next() [9] (and this sequence can be repeated zero or many times). This can be concisely specified with the following method call sequence:

\[ (\text{next}(), \text{remove}())^* \] (1)

The method call sequence above represents a call to next() followed by an optional call to remove() (and this sequence can be repeated zero or many times). Because method call sequences separate protocol specifications from method contracts (that express functional properties), they help writing concise and meaningful specifications.

In this paper, we extend [8] to deal with multithreaded programs and a variant of generic classes. For this we extend the specification language and the semantical foundations of the work mentioned above. We introduce the keyword protocol to specify allowed sequences of method calls on objects of a class. We maintain the spirit of [8] by providing a concise and intuitive regular expression-like notation to write protocols of multithreaded programs. The semantics of the execution of a multithreaded program is represented by a trace. A trace is a sequence of actions where an action is either method entering or method exiting. The semantics of a specification is represented by a set of traces.

Additionally, we provide a new technique to check if method contracts are correct w.r.t. to protocols. For example, given the protocol (1) above, a programmer has to make sure that next()’s postcondition implies remove()’s precondi- tion. Otherwise, it would be wrong to execute next() and then to execute remove() (although it is a correct execution w.r.t. to the protocol considered). We formalize this intuition by saying that method contracts should adhere to protocols. We describe how to generate proof obligations (POs) that represent the adherence of method contracts to protocols. For the protocol of Java’s Iterator shown above (1), there are three POs: (1) that next()’s postcondition implies remove()’s precondition (because remove() can be called after next()), (2) that remove()’s postcondition implies next()’s precondition (because the protocol can be repeated after a call to remove()), and (3) that next()’s postcondition implies next()’s precondition (because the call to remove() is optional, the protocol can be repeated after a call to next()). Proving these three POs suffices to show...
that the method contracts of $\text{next()}$ and $\text{remove()}$ adhere to class $\text{Sprite}$’s protocol. This technique can be applied early in the development process because it only require methods to be specified with contracts (not necessarily to be implemented). Thus, it can provide feedback to programmers early in the development process. The generation of POs is done in a way similar to proof graphs [10, Chap. VIII]. Method contracts are written with permission-accounting separation logic [11], [12] adapted for multithreaded Java [13].

This paper is organized as follows: Section II informally introduces our model language and presents an example of writing protocols in our language, Section III formally defines our language for specifying protocols, Section IV defines the semantics of multithreaded programs and the semantics of protocols, Section V shows how we generate POs representing the adherence of method contracts to protocols, Section VI evaluates our approach and discusses its limitations, Section VII describes related work and future work, and Section VIII concludes.

II. THE MODEL LANGUAGE

The complete description of our model language can be found in our earlier work [13]. Here, we simplify the presentation and only sketch the relevant features. The source code of our model language is similar to Java and we allow programmer to specify methods with pre and postconditions (also called contracts) à la JML [3]. A precondition specifies what the client must provide at method entry, while a postcondition specifies what is ensured to the client at method exit.

Contracts are specified using permission-accounting separation logic [11], [12]. Permission formulas include the $\text{Perm}(o.f, \pi)$ which represents permission $\pi$ to access the field $o.f$. A permission is a fraction between 0 (excluded) and 1 (included) [14]. Permission 1 grants write and read access while any permission less than 1 grants readonly access. Permission formulas also include abstract predicates of the form $o.\kappa<\pi>$ i.e., predicate $\kappa$ instantiated with parameters $\pi$ and applied to $o$. Abstract predicates represent complex permission formulas that express complex specifications. Although we do not use them in this paper, other formulas permit to specify functional (as opposed to permission-related) properties.

Separation logic comes with two new operators: $\ast$ (called “separating conjunction”) and $\rightarrow$ (called “magic wand”). In this paper, $\ast$ can be read as $\&$ and $\rightarrow$ can be read as $\Rightarrow$.

Figure 1 shows a generic class $\text{Sprite}$ written in our model language. In our model language, specifications are written as special comments which begin by //@ (as in JML). Class $\text{Sprite}$ defines an abstract predicate called $\text{state}$ which represents the permission to access fields $x$, $y$, and $col$. The state predicate is parameterized by permission $q$. Intuitively, given an object $o$ of type $\text{Sprite}$, $o.\text{state}<1>$ represents the permission to write and read fields $o.x$, $o.y$, and $o.col$ while $o.\text{state}<q>$ (for any $q<1$) represents the permission to read fields $o.x$, $o.y$, and $o.col$.

```java
class Sprite<permission p>{
  private int x,y,col;
  //@ pred state<permission q> =
  //@  Perm(x,q) * Perm(y,q) * Perm(col,q);
  //@ protocol =
  //@  ((p==1 ? update()),
  //@   display() || display())*
  //@ requires true;
  //@ ensures state<p>;
  public Sprite(int x, int y, int col){
    this.x = x; this.y = y; this.col = col;
  }
  //@ requires state<1>;
  //@ ensures state<1>;
  void update(){ x=...; y=..., col=...; }
  //@ requires state<p/2>;
  //@ ensures state<p/2>;
  void display() { ... } // only read
}
```

Fig. 1. A Class Annotated with Contracts and a Protocol

Class $\text{Sprite}$ is parameterized by permission $p$. Like generic types, generic permissions are instantiated at object creation. Here, the generic permission $p$ allows to construct modifiable sprites or immutable sprites. A modifiable sprite can be modified after construction while an immutable sprite cannot. This is enforced by the postcondition of class $\text{Sprite}$’s constructor which returns the abstract predicate $\text{state}<p>$ to the client. If $p<1$, this means that the sprite cannot be modified after creation.

Methods of class $\text{Sprite}$ are annotated with pre and postconditions. Method $\text{update()}$ requires the client to provide permission formula $\text{state}<1>$ because $\text{update()}$ writes to fields $x$, $y$, and $\text{col}$. Method $\text{display()}$ requires the client to provide permission formula $\text{state}<p/2>$ because $\text{display()}$ reads to fields $x$, $y$, and $\text{col}$.

Class $\text{Sprite}$ contains an example of our approach for specifying protocols. We use the keyword $\text{protocol}$ to declare a protocol for a given class. It specifies the allowed sequence of method calls on objects of this class. The protocol for class $\text{Sprite}$ should be understood as follows: first, $p == 1$ ? $\text{update()}$ means that $\text{update()}$ can be called only on modifiable sprites; second, $\text{display()} || \text{display()}$ means that two threads can call $\text{display()}$ in parallel on a given $\text{Sprite}$.

Finally, the $\ast$ at the end means that the whole protocol can be executed 0 or many times.

1For simplicity, we impose that $\text{display()}$ can only be called twice in parallel. However, if we had used methods parameterized by permissions (as our full system provides [13, §3]), we could allow unbounded parallelism (by replacing $\text{display()} || \text{display()}$ with $\text{display()}$).
III. A SIMPLE SPECIFICATION LANGUAGE FOR PROTOCOLS

Protocols are specified with the following grammar:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha \in \text{PermissionVar}$</td>
<td>$\text{permission variables}$</td>
</tr>
<tr>
<td>$\pi \in \text{Permission}$</td>
<td>$\alpha \mid \pi / \pi / 2$</td>
</tr>
<tr>
<td>$op \in \text{Operator}$</td>
<td>${=, &lt;, \leq, +, \ldots}$</td>
</tr>
<tr>
<td>$b \in \text{BoolExpr}$</td>
<td>$\text{true, false, } b \land b$</td>
</tr>
<tr>
<td>$m \in \text{Method}$</td>
<td>$m$</td>
</tr>
<tr>
<td>$s \in \text{Spec}$</td>
<td>$s, s', s \mid s'$</td>
</tr>
</tbody>
</table>

Our specification language consists of specifications for methods, sequential composition of specifications, composition of specifications with regular expression-like notations, conditional specifications, and parallel composition of specifications. This syntax permits to express concisely and intuitively many protocols of multithreaded programs.

The meaning of the specifications is straightforward: $m$ denotes $m$’s method, $s, s'$ denotes the sequential composition of $s$ and $s'$, $s \mid s'$ denotes $s$ and $s'$, $b ? s : s$ denotes $s$ if $b$ is true ($s'$ otherwise), $s ?$ denotes $s$ zero or one time, $s*$ denotes $s$ zero or many times, $s+$ denotes $s$ one or many times, $s \mid s'$ denotes $s$ in parallel with $s'$ (bounded parallelism), and $! s$ denotes one or many $s$ in parallel (unbounded parallelism).

Protocols can depend on class parameters in the case $b ? s : s'$. This permits to adapt protocols to the different behaviors of a class (as class Sprite shows).

Compared to [8], we do not allow to specify nested call sequences. This forbids to specify that, for example, given two methods $m$ and $n$, $m$ should call $n$. We do not consider this as a limitation since our goal is different from the cited work. We focus on public protocols (protocols that are used by clients), because checking adherence of method contracts to protocols (see Section V) is useful only for public protocols, whereas nested call sequences are useful only for private protocols (protocols that are used by class implementers). Contrary to [8], we allow to specify optional protocols (case $s ?$), to specify conditionals (case $b ? s : s$), to specify bounded parallelism (case $s \mid s$), and to specify unbounded parallelism (case $! s$).

IV. SEMANTICS

A. SEMANTICS OF MULTITHREADED PROGRAMS

The semantics of multithreaded programs is defined in terms of actions, permissions, and traces. An action is either method entering or method exiting and a trace is a pair of a list of permissions (denoted by $\pi$) and a sequence of actions:

$$ a \in \text{Action} ::= m.\text{enter} \mid m.\text{exit} $$
$$ \tau \in \text{SeqAction} ::= \epsilon \mid a \tau $$
$$ t \in \text{Trace} ::= (\pi, \tau) $$

Intuitively, given a distinguished object $o$ of (a possibly parameterized) class $C$, a trace keeps track of (1) the permissions used to instantiate $C$ when $o$ was created and (2) the sequence of method calls on $o$.

Traces contain permissions because protocols are parameterized by permissions (remember that, in the case $b ? s : s$, $b$ can refer to permission variables i.e., class parameters). The protocol of a given object depends on the permissions used to instantiate the class of this object. For example, given a modifiable Sprite object (i.e., an object of class Sprite$<$p$>$ with $p = 1$), it is correct to call update() on this object after its creation (as class Sprite’s protocol specifies). However, if the considered object was an immutable Sprite, it would be incorrect to call update() on this object after its creation. That is why, to interpret a sequence of actions w.r.t. to a protocol, we need to know with which permissions the class of the considered object has been instantiated.

The concatenation of a trace $\tau$ and an action $a$ is written $a \tau$. Similarly, the concatenation of two traces $\tau$ and $\tau'$ is written $\tau \tau'$. Note that, given the execution of a multithreaded program, we do not distinguish between actions from different threads. All actions of all threads form a single trace. This suffices to express the semantics of protocols.

B. SEMANTICS OF PROTOCOLS

The semantics of protocols is given by $[\cdot] : \text{Spec} \rightarrow 2^\text{SeqAction}$. Intuitively, $[s \mid (\pi)]$ returns the set of all possible traces that satisfy $s$ instantiated with $\pi$. For the cases different from $b ? s : s, s \mid s$, and $! s$, $[\cdot]$’s definition is standard:

$$ [m \mid (\pi)] \triangleq \{ m.\text{enter}, m.\text{exit} \} $$
$$ [s, s' \mid (\pi)] \triangleq \{ \tau \tau' \mid \tau \in [s \mid (\pi)], \tau' \in [s' \mid (\pi)] \} $$
$$ [s \mid s' \mid (\pi)] \triangleq [s] \cup [s \mid (\pi)] $$
$$ [s ?] \triangleq \{ \epsilon \} $$
$$ [s *] \triangleq \bigcup_{i \in \mathbb{N}} [s]^i \mid (\pi) $$
$$ [s +] \triangleq \bigcup_{i \in \mathbb{N}^+} [s]^i \mid (\pi) $$

where $[s]^n(\pi)$ is defined as follows:

$$ [s]_0(\pi) \triangleq \{ \epsilon \} $$
$$ [s]_i'(\pi) \triangleq \{ \tau \tau' \mid \tau \in [s] \mid (\pi), \tau' \in [s]_i-1 \mid (\pi) \} $$

To define the case $b ? s : s$ of the semantics of specifications, we define the semantics of Boolean expressions with $[\cdot] : \text{BoolExpr} \rightarrow \text{State} \rightarrow \{\bot, \top\}$. The case $b ? s : s$ is the only reason why we keep track of permissions in the semantics of specifications (because $b$ can refer to class parameters i.e., permissions). Then, we can define:

$$ [b ? s : s' \mid (\pi)] \triangleq \begin{cases} [s \mid (\pi)] & \text{iff } [b] \mid (\pi) \\ [s' \mid (\pi)] & \text{otherwise} \end{cases} $$
Note that, because generic parameters of a class are instantiated when objects are created, it suffices to compute the semantics of a protocol when the object considered is created.

To define the cases \( s \mid \tau \) and \(!s\mid \tau\) of the semantics of specifications, we define the interleaving of two sequences of actions with \(\llbracket \cdot \rrbracket: \text{SeqAction} \times \text{SeqAction} \rightarrow 2^{\text{SeqAction}}\):

\[
\begin{align*}
\epsilon \llbracket \tau & \triangleq \{ \{ \} \} \\
\tau \llbracket \epsilon & \triangleq \{ \{ \} \} \\
\alpha \llbracket \alpha' \tau' & \triangleq \{ \{ \alpha'' \mid \tau'' \in \tau \land \alpha' \tau' \} \cup \{ \alpha'' \mid \tau'' \in \alpha \tau \land \tau' \} \}
\end{align*}
\]

For example, given two methods \( m \) and \( n \), \( m.\text{enter} \land n.\text{exit} \land n.\text{enter} \land n.\text{exit} \) is:

\[
\{ m.\text{enter}, m.\text{exit}, n.\text{enter}, n.\text{exit}, m.\text{enter}, n.\text{enter}, n.\text{exit}, m.\text{enter}, n.\text{enter}, n.\text{exit}, n.\text{enter}, m.\text{enter}, n.\text{enter}, n.\text{exit}, n.\text{enter}, m.\text{enter}, n.\text{enter}, m.\text{exit} \}
\]

The \( \llbracket \cdot \rrbracket \) operator is extended to sets of sequences (\( \llbracket \cdot \rrbracket: 2^{\text{SeqAction}} \times 2^{\text{SeqAction}} \rightarrow 2^{\text{SeqAction}} \)) in the straightforward way. Then, we can define:

\[
\begin{align*}
[s \mid s'](\pi) & \triangleq [s](\pi) \llbracket [s'](\pi) \\
[s](\pi) & \triangleq \bigcup_{i \in \mathbb{N}^+} \llbracket^i s(\pi)
\end{align*}
\]

where \( \llbracket^i : \text{Spec} \rightarrow 2^{\text{SeqAction}} \) is defined as follows:

\[
\begin{align*}
\llbracket^1 s(\pi) & \triangleq \llbracket s(\pi) \\
\llbracket^i s(\pi) & \triangleq [s](\pi) \llbracket \llbracket^{i-1} s(\pi) \} & (\text{for } i \geq 2)
\end{align*}
\]

C. Satisfaction of Traces w.r.t. Protocols

A trace satisfies a protocol if its underlying sequence is the prefix of one of the sequences denoted by the protocol. We use the prefix of one of the sequences denoted by the protocol because we consider that not terminating a protocol is harmless. Formally:

\[
(\pi, \tau) \vdash s \iff \tau \models [s](\pi)
\]

where \(\tau \models [s](\pi)\) is defined as follows:

\[
\tau \models [s](\pi) \iff (\exists \tau')(\tau' = [s](\pi))
\]

V. Adherence of Method Contracts to Protocols

To help programmers check that method contracts adhere to protocols, we generate proof obligations (POs). Because our method for generating POs only depend on protocols and the contracts of methods mentioned in protocols, they can be generated early in the development process (before implementing methods). Then, if the POs are provable, the programmer is sure that method contracts adhere to protocols i.e., the order in which methods must be called is correct w.r.t. to the contracts of the methods considered.

POs only mention the receiver of the class considered (this): we do not consider properties of method parameters. We “project” all POs w.r.t. to the receiver, because our goal is to check if contracts relevant to the receiver (the object to which the protocol applies) adhere to the protocol considered.

As an example, imagine that method \text{display()} of class \text{Sprite} has a parameter \( o \) of type \text{Sprite} and that method \text{display()}’s precondition is as follows:

\[
\text{requires state}<1> \ast o.\text{state}<1>;
\]

Then, the corresponding projected precondition is:

\[
\text{requires state}<1>;
\]

Intuitively, we do not consider properties of method parameters and focus our analysis on the receiver. Formally, we use the function \( \llbracket \cdot \rrbracket: \text{Formula} \rightarrow \text{Formula} \) to project formulas w.r.t. to the receiver. For space reasons, we omit \( \llbracket \cdot \rrbracket \)’s definition (it is by induction on the structure of formulas).

To generate the POs, we need to compute (1) the precondition of a protocol i.e., a permission formula that suffices to soundly execute a program satisfying this protocol and (2) the postcondition of a protocol i.e., a permission formula that a programmer can assume after the execution of a program that satisfies the protocol considered.

To compute the precondition of a protocol, we first need to define the function \( \text{opt}: \text{Spec} \rightarrow \{ \bot, \top \} \). Given a specification \( s \), \( \text{opt}(s) \) returns true if \( s \) is satisfied by an empty trace (i.e., \( s \) can be skipped). The function opt is useful to compute the preconditions of a program of the form \( s, s' \) if \( s \) is not optional, the precondition of \( s, s' \) is the precondition of \( s \); while if \( s \) is optional, the precondition of \( s, s' \) is the precondition of \( s \) or the precondition of \( s' \).

\[
\begin{align*}
\text{opt}(m) & \triangleq \text{false} \\
\text{opt}(s, s') & \triangleq \text{opt}(s) \land \text{opt}(s') \\
\text{opt}(s | s') & \triangleq \text{opt}(s) \lor \text{opt}(s') \\
\text{opt}(s?) & \triangleq \text{true} \\
\text{opt}(s+) & \triangleq \text{opt}(s) \\
\text{opt}(b ? s : s') & \triangleq \text{opt}(s) \lor \text{opt}(s') \\
\text{opt}(s | s') & \triangleq \text{opt}(s) \land \text{opt}(s') \\
\text{opt}(!s) & \triangleq \text{opt}(s)
\end{align*}
\]

Then, we can define \( \text{pre}: \text{Spec} \rightarrow \text{Formula} \) to compute the precondition of a protocol. This function uses the syntactic function \( \text{pre}_{\text{syn}}: \text{Method} \rightarrow \text{Formula} \) that looks up the precondition of a method in the class table of the program considered. To deal with unbounded parallelism (case !s), we denote the maximum number of threads that can be created on a given Java Virtual Machine (JVM) with “max”. Because the value of max depends from the hardware and the operating system, we do not impose it here. In addition, for any permission formula \( F \), we define the *-conjunction of \( F \) with itself as follows:

\[
\begin{align*}
\otimes_0 F & \triangleq \text{true} \\
\otimes_i F & \triangleq F \ast \otimes_{i-1} F & (\text{for } i \geq 1)
\end{align*}
\]
Then, pre’s definition is as follows:

\[
\begin{align*}
\text{pre}(m) & \triangleq \downarrow \text{pre}_\text{syn}(m) \\
\text{pre}(s, s') & \triangleq \{ \text{pre}(s) \mid \text{pre} \iff \neg \text{opt}(s), \text{pre}(s') \} \text{ otherwise} \\
\text{pre}(s | s') & \triangleq \text{pre}(s) \cup \text{pre}(s') \\
\text{pre}(s?) & \triangleq \text{pre}(s) \\
\text{pre}(s+) & \triangleq \text{pre}(s) \\
\text{pre}(b ? s : s') & \triangleq (b \land \text{pre}(s)) \cup (!b \land \text{pre}(s')) \\
\text{pre}(s | s') & \triangleq \text{pre}(s) \ast \text{pre}(s') \\
\text{pre}(!s) & \triangleq \otimes_{\text{max}} \text{pre}(s)
\end{align*}
\]

The definition of pre corresponds to our intuitive interpretation of the composition of specifications. However, we would like to detail four cases of pre’s definition. First, the precondition of s, s’ is pre(s) if s is not optional (i.e., s cannot be skipped), but the precondition of s, s’ is pre(s) \mid pre(s’) to handle the case where s is optional. Second, the precondition of s | s’ is pre(s) & pre(s’), because a program that executes s or s’ must provide a permission formula that satisfies both pre(s) and pre(s’). Third, the precondition of s | s’ is pre(s) \ast pre(s’), because a program that executes s in parallel with s’ must provide two distinct permission formulas to satisfy both pre(s) and pre(s’). Fourth, the precondition of !s is \otimes_{\text{max}} pre(s), because a program, that executes s in parallel with itself up to max times, must provide max distinct permission formulas that all satisfy pre(s).

Similarly to pre(s), to compute the postcondition of a protocol, we define post : Spec \rightarrow Formula as follows:

\[
\begin{align*}
\text{post}(m) & \triangleq \downarrow \text{post}_\text{syn}(m) \\
\text{post}(s, s') & \triangleq \{ \text{post}(s') \mid \text{post} \iff \neg \text{opt}(s), \text{post}(s') \} \text{ otherwise} \\
\text{post}(s | s') & \triangleq \text{post}(s) \mid \text{post}(s') \\
\text{post}(s?) & \triangleq \text{post}(s) \\
\text{post}(s+) & \triangleq \text{post}(s) \\
\text{post}(b ? s : s') & \triangleq (b \land \text{post}(s)) \cup (!b \land \text{post}(s')) \\
\text{post}(s | s') & \triangleq \text{post}(s) \ast \text{post}(s') \\
\text{post}(!s) & \triangleq \otimes_{1} \text{post}(s) \mid \otimes_{2} \text{post}(s) \mid \ldots \mid \otimes_{\text{max}-1} \text{post}(s) \mid \otimes_{\text{max}} \text{post}(s)
\end{align*}
\]

The definition of post must be understood as follows: post(s) returns the postcondition obtained after s if s was executed (which might not be the case if s is optional). In the case s was not executed, post(s) is vacuous. We detail two cases of post’s definition. First, the postcondition of s | s’ is post(s) \mid post(s’) because we cannot know if s or s’ was executed. Second, the postcondition of !s considers all possible cases: \otimes_1 post(s) models that s can be executed by one thread, \otimes_2 post(s) models that s can be executed by two threads, \ldots, \otimes_{\text{max}-1} post(s) models that s can be executed by max - 1 threads, and \otimes_{\text{max}} post(s) models that s can be executed by max threads.

Now, we can define what POs must be proven to show that method contracts adhere to a given protocol. We define po^{\text{int}} : \text{Spec} \rightarrow 2^{\text{Formula}} to compute the POs representing the beginning of a protocol. Intuitively, proving the POs returned by po^{\text{int}} shows that, after object creation, the first method of a protocol can be entered. Before defining po^{\text{int}}, we define the syntactic function post^{\text{conc}} : \text{Class} \rightarrow 2^{\text{Formula}} to lookup the postconditions of the constructors of a given class. Then, because the permission formula obtained after object creation is the postcondition of the constructor called, po^{\text{int}}’s definition is as follows (where C is the class considered):

\[
\text{po}^{\text{int}}(s) \triangleq \{ F \rightarrow \text{pre}(s) \mid F \in \downarrow \text{post}^{\text{conc}}(C) \}
\]

We define po : \text{Spec} \rightarrow 2^{\text{Formula}} to compute the POs representing that method contracts adhere to the inside of a protocol. The definition of po is as follows:

\[
\begin{align*}
\text{po}^{\text{in}}(m) & \triangleq \emptyset \\
\text{po}^{\text{in}}(s, s’) & \triangleq \{ \text{post}(s) \ast \text{pre}(s’) \} \cup \text{po}^{\text{in}}(s) \cup \text{po}^{\text{in}}(s’) \\
\text{po}^{\text{in}}(s | s’) & \triangleq \text{po}^{\text{in}}(s) \cup \text{po}^{\text{in}}(s’) \\
\text{po}^{\text{in}}(s?) & \triangleq \text{po}^{\text{in}}(s) \\
\text{po}^{\text{in}}(s+) & \triangleq \text{po}^{\text{in}}(s) \\
\text{po}^{\text{in}}(b ? s : s’) & \triangleq \{ b \rightarrow F \mid F \in \text{po}^{\text{in}}(s) \} \\
\text{po}^{\text{in}}(s | s’) & \triangleq \text{po}^{\text{in}}(s) \cup \text{po}^{\text{in}}(s’) \\
\text{po}^{\text{in}}(!s) & \triangleq \text{po}^{\text{in}}(s)
\end{align*}
\]

In this paragraph, we detail three cases of po^{\text{in}}’s definition. First, the POs of s, s’ means that to execute s, s’ correctly it suffices that post(s) implies pre(s’) and that the execution of s and s’ are correct. Second, the POs of s+ means that to execute s* correctly it suffices that post(s) implies pre(s) (because the execution of s can be repeated like a loop) and that the execution of s is correct. Third, the POs of b ? s : s’ means that to execute b ? s : s’ correctly, it suffices to show that the execution of s is correct (knowing that b holds) and that the execution of s’ is correct (knowing that !b holds).

Finally, we define po : \text{Spec} \rightarrow 2^{\text{Formula}} to compute sufficient POs representing that method contracts adhere to a protocol:

\[
\text{po}(s) \triangleq \text{po}^{\text{int}}(s) \cup \text{po}^{\text{in}}(s)
\]

A. Example

The POs generated to show that the method contracts of class Sprite (visible in Figure 1) adhere to class Sprite’s protocol are:

\[
\begin{align*}
\text{state}<p> & \rightarrow (p = 1 \rightarrow \text{state}<1>) \\
\text{state}<p> & \rightarrow (\text{state}<p/2> \ast \text{state}<p/2>) \\
(p = 1 \ast \text{state}<1>) & \rightarrow (\text{state}<p/2> \ast \text{state}<p/2>) \\
(\text{state}<p/2> \ast \text{state}<p/2>) & \rightarrow (p = 1 \rightarrow \text{state}<1>) \\
(\text{state}<p/2> \ast \text{state}<p/2>) & \rightarrow (\text{state}<p/2> \ast \text{state}<p/2>)
\end{align*}
\]
POs (2) and (3) represent the beginning of the protocol i.e., that the postcondition of class Sprite’s constructor implies the precondition of the first method of the protocol. PO (2) models that update() can be called first while PO (3) models that display() can be called after update(). PO (4) models that display() can be called after display() while PO (6) models that display() after display(). PO (5) and (6) model that the protocol can loop: PO (5) models that update() can be called after display() while PO (6) models that display() after display(). PO (6) (if update() is skipped).

To work around this issue, the authors provide a workaround correct to execute this method at any point in the protocol. The second condition ensures that a method is called after display(). Finally, PO (5) and (6) model that the protocol can loop; PO (5) models that update() can be called after display() while PO (6) models that display() after display(). PO (6) (if update() is skipped).

All these POs can be proven with the proof system defined in our earlier work [13, §5]. This means that the contracts of class Sprite’s methods adhere class Sprite’s protocol.

### B. Innocuous Methods

In the work we extend [8], the authors mention that, often, classes contain too many methods to specify all possible protocols. To work around this issue, the authors provide a weak satisfaction relation that interprets a protocol w.r.t. to a (restricted) set of methods. Intuitively, if a method is not mentioned in the restricted set of methods, then it is considered correct to execute this method at any point in the protocol.

In our extended setting, this workaround still works, thus, we do not detail it further. However, we adapt this workaround to our static technique for checking that method contracts adhere to protocols. We say that a method is **innocuous** w.r.t. to a protocol if it can be executed at any point in a protocol without breaking the adherence of other methods to this protocol. We introduce the method modifier `innocuous` (enclosed in `/*@ ... @*/`) to indicate innocuous methods. Figure 2 shows class SpriteIDed, an extension of class Sprite, that contains the innocuous method `getId()`. Class SpriteIDed is similar to class Sprite except that it is extended with field `id`. Field `id` is written once during initialization. Later, field `id` can only be read by method `getId()`.

Generally, to check that a method `m` is innocuous w.r.t. to a protocol specification `s`, it suffices to show the following two conditions:

1. `post(m) → pre(m)`
2. `(∀m' ∈ s) ((pre(m') → pre(m)) ∧ post(m') → pre(m))`

The first condition ensures that innocuous methods never consume permissions formulas: innocuous methods return a permission formula stronger (or identical) than the permission they require to execute. The second condition ensures that innocuous methods can be called at any point in a protocol.

### VI. Evaluation and Limitations

We evaluated the usefulness of our protocol specifications with a number of examples. We found that a variety of multithreaded programs can be concisely specified with our extension. In particular, we found that classes using well-structured parallelism such as divide-and-conquer algorithms or barriers can be easily specified with protocols. However, more work is needed to see if our specification technique is flexible enough to handle complicated and unstructured parallelism patterns.

We evaluated our technique of generating POs to check correctness of specifications while we were trying to prove a concurrent `Iterator` class (described in details in our earlier work [15]). We found that, without the systematic approach of checking the adherence of method contracts to protocols, it was very time-consuming to find the right way to write specifications. It should be noted, though, that our technique for generating POs could be refined to generate necessary and sufficient POs. Precisely, the POs generated does not allow to put apart some permissions while they are not needed and to reuse them later (as separation logic’s frame rule permits [12] in usual program verification). Current work is tackling this issue: instead of generating POs directly, we first generate a “normalized” program representing the possible executions of a protocol. Then, the proof obligations consist in proving this program correct.

We believe that our approach is sound although we did not prove it. To prove soundness, a possible approach is to show that a protocol “induces” a normalized program, that executes the call sequence denoted by this protocol. Then, one should show that the POs generated to check the adherence of method contracts to this protocol are equivalent to the POs generated for verifying the corresponding normalized program with object-oriented separation logic [13].

### VII. Related Work and Future Work

Design by Contract (DBC) was first introduced by Meyer [2] in the Eiffel programming language. Eiffel fea-
tures method contracts, invariants, and built-in facilities to dynamically check contracts. Eiffel does not support protocols and there is no tool to statically check properties of Eiffel programs.

Support for DBC in Java is provided by several tools including JML [3], Jass [16], and ESC/Java2 [6]. The earlier work of Cheon and Perumandla [8] to provide method call specifications in JML greatly inspired our work. Jass permits to specify protocols in the style of CSP. Both the work of Cheon and Perumandla and Jass support dynamic checking of protocols. We do not provide an implementation for dynamically checking protocols but our goal is different: we use protocols to statically check the adherence of method contracts to protocols. ESC/Java2 permits to statically check many properties of Java programs. It does not support protocols.

Support for DBC in C# is provided by Spec# [4] and Boogie [7]. There is no support for protocols.

Separation logic [11] has been adapted to a Java-like setting by Parkinson [17]. This work focused on foundational issues and no support for protocols is provided.

Future work includes the implementation of a runtime checker to check (dynamically) that programs respect their protocols. For this, we need to adapt the technique outlined in [8]. In a nutshell, it would be necessary to (1) synchronize access to the executable representation of specifications (i.e., automata) and (2) extend automata generation to deal with the new cases of our specification language.

Future work also includes the study of inheritance. From a semantical point of view, as in the previous work on method call sequences [8], a possible interpretation of protocols inheritance is to conjoin inherited protocols to the protocols declared in the inheriting class. This means that a class would have to respect inherited protocols (possibly ignoring methods that are declared in this class) and would have to respect its own protocols. From the point of view of checking adherence of method contracts to protocols, the technique of behavioral subtyping for inheritance of method contracts [18], should permit to check adherence in a modular way. It means that an inheriting class whose method contracts respect behavioral subtyping should adhere to inherited protocols.

VIII. CONCLUSION

We provided a concise and intuitive regular expression-like notation to specify protocols of multithreaded Java-like programs that use a variant of generic classes. We presented the semantical foundations of our specification language.

We showed a new technique to check that method contracts are correct w.r.t. to protocols. For this, we generate proof obligations in a way similar to proof graphs. Trying to prove these proof obligations can provide useful feedback to programmers early in the development process. We provided a way to ignore some methods from protocols without breaking the usefulness of checking that method contracts are correct w.r.t. to protocols.

Acknowledgments We thank Christian Haack for interesting discussions about separation logic and Marieke Huisman and Gustavo Petri for their feedback on earlier versions of this paper. This work was supported in part by the European Commission IST-FET-2005-015905 Mobius project and by the Agence Nationale de la Recherche ANR-06-SETIN-010 ParSec project.

REFERENCES

Quality Evaluation of Web-sites of Indian E-commerce Domain

Priyanka Tripathi, Namita Shrivastava and M Kumar
Department of Computer Applications
Maulana Azad National Institute of Technology, Bhopal, INDIA

Abstract-This paper describes the development in the area of quality evaluation models for web-applications. The authors have proposed a design of quality evaluation method, its implementation and partial results in order to evaluate the Indian e-commerce web-sites. We have considered four quality factors namely usability, functionality, reliability and efficiency to evaluate the web-site to measure the quality preference outcome. Further, by combining the outcome of each factor, we find out the Global Quality Preference of the website. We have used Logic Scoring Preferences (LSP) grounded on Continuous Preference Logic as mathematical background.

Key-words: Quality, Evaluation, E-commerce.

1 Introduction

E-commerce is an excellent alternative for companies to get new customers fast in todays commercial world. The success of these e-stores depends heavily on their quality in terms of quality of service given by them. To evaluate the quality of e-commerce based websites we need to identify the attributes which contribute to the quality. Quality is an intrinsic and multifaceted characteristics of a product[1]. Quality is not an absolute, it depends on the appraiser’s perspective. Hence any quality measure must be subjective, summarizing the impression of a given class of individuals that interact with the product[2]. The main purpose of software quality evaluation is to supply referential quantitative results to the software products that are reliable, understandable and acceptable to anyone’s interest [3]. Identification of the requirements of what you should built is the hardest part of the design process. This leads to the design of excellent web shops from average ones[4]. Because of the increasing size, complexity, quality needs and market demands for web-applications, several problems have frequently been reported [5].

The World Wide Web is a universal information space overcoming barriers created by humans towards people with different cultures or physical limitations [6]. Evaluating the quality of a web site requires expensive methods such as heuristic evaluations or empirical usability tests. In the first case a group of specialists or expert evaluators apply their experience to conduct independent evaluations and usually it does not permit to find problems related to typical users of the site. In the second case a group of users with different background, age, and skills characteristics are called to browse the web site in order to evaluate their satisfaction in using it. Lack of measurement criteria to evaluate aspects related to the quality in use, such as usability and accessibility of a web application is giving a significant rise in number of poor quality web-application.

2 Literature Survey

Different approaches are defined for web-site quality evaluation. An important initiative towards web-site quality in cultural environment is MINERVA[7] (Ministerial NEtwoRk for Valorising Activities in Digitisation). The quality criteria have therefore a double objective, they represent the quality factors for evaluating the quality of a cultural site on the web; they direct and support the process of design and development of a cultural website. Mich et. al [8] presents very flexible approach to evaluate a generic web site. The 2QC3V3Q, also called 7-loci, is a conceptual model to evaluate web site quality based on seven dimensions: who-what-why-when-where-
how, and feasibility. A metamodel called 2QCV3Q (in Latin V stands for U), which takes its name from the initials of the Ciceronian loci of classical rhetoric that it is based on [9]. Also called 7Loci, the meta-model takes into special consideration the inherently communicative nature of a Web site [8,10]. This concept is used to identify the fundamental dimensions of a Web site, resulting in a framework that when compared with existing models can be seen as a meta-model for classification of diverse criteria for quality (table 1). In other words, the seven loci or dimensions constitute the general framework of the “quality models”, which is independent of the sites under analysis.

1. The first dimension, **Identity**, regards the image that the organisation projects and therefore all elements that come together in defining the identity of the owner of the site.

2. **Content** to the information

3. **Services** refer to the services available for users.

4. **Location** regards the visibility of a site; it also refers to the ability of the site to offer a space where users can communicate with each other and with the organisation.

5. **Maintenance** comprises all activities that guarantee proper functioning and operability of the site.

6. **Usability** determines how efficiently and effectively the site’s content and services are made available to the user.

7. **Feasibility** includes all aspects related to project management.

Table 1. Dimensions of the 2QCV3Q - 7Loci meta-model

<table>
<thead>
<tr>
<th>QVIS(Who?)</th>
<th>QVID(What?)</th>
<th>CVR(Why?)</th>
<th>VBI(Where?)</th>
<th>QVANDO(When?)</th>
<th>QVOMODO(How?)</th>
<th>QVIBUS AVXILIIS(With what means?)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identity</td>
<td>Content</td>
<td>Services</td>
<td>Location</td>
<td>Maintenance</td>
<td>Usability</td>
<td>Feasibility</td>
</tr>
</tbody>
</table>

ISO9126, ISO9241 AND ISO13407 [3,11,12] are very popular software quality standards. ISO9126, ISO9241 AND ISO13407 describe the standards for quality of software, usability, user-centered product. Conformance to standard is also the basis of W3C quality assurance initiative [13]. The 7Loci meta-model supports a systemic approach to evaluating web site quality that takes into account these diverse components coming together at a site and the importance of satisfying the needs of all actors. A usability-focused evaluation method for hypermedia application is MiLE, based on a combination of inspection from expert evaluator and empirical testing through panels of end users [14]. The evaluation model here is based on two heuristic concepts: abstract and concrete tasks. An analytic web site quality model is proposed by Etnoteam [15]. It is based on six attributes: communication, content, functionality, usability, management and accessibility. The model can be personalized. The sub-attributes are weighted depending on the site category. An evolution of this model is described in [16]. Another analytical approach described in [17] proposes a Web-site Quality Evaluation Method based on a logic multi-attribute decision model and procedures intended to be a useful tool to evaluate artifact quality in the operational phase. Studies for quality of products and processes for the web are very recent and still there are no widely spread evaluation methods and techniques for quality assessment.

Fleming 2001[18], Mendes et al. 2001[19], Olsina and Rossi 2001[20], Pooley et al. 2002[21] have given the quality attributes and also some of them have evaluated the quality of different web-application domains. The ISO/IEC 9126 series standard introduced a hierarchical model with six major quality characteristics, each very broad in nature. They are subdivided into 27 sub-characteristics, which contribute to external quality, and 21 sub-characteristics which contribute to internal quality. Since the characteristics and sub-characteristics are not properly defined, ISO9126 does not provide a conceptual framework within which comparable measurements may be made by different parties with different views of software quality, such as users, developers and managers. The Authors have given a Quality Evaluation Framework in order to evaluate the quality of Indian e-commerce based web-sites in Section 3. In our previous research, we have validated factors and sub-factors from user’s point of view [25] and
Designing the quality evaluation framework

A general model of the evaluation framework is given in figure 2. It shows an initial phase which includes the identification of the evaluation requirements, a design phase in which the evaluation plan and techniques are defined, and a final ranking as per the quality and suggestion phase.

I. Quality requirements specification:
What is the mission of the web-application. Specify the functional Usability, Functionality and non-functional quality requirements such as Reliability, Efficiency of the web-applications. What is the priority associated with each of the factors. Identify the category of users. web-application users needs vary according to the category of users. Outcome of this phase is evaluation goals. In designing a web-application developer must see other web-applications having similar objectives. The reason for this is because the users always compare a web application with the other existing applications and uses the most convenient or in other words web-application which has a greater Usability.

II. Metrics selection: Select the metrics, their corresponding weights the scale of measurement, and for summation. Outcome of this phase is evaluation plans.

III. Measurement: Measurement implementation and result calculation.

IV. Result: Ranking of web-applications as per the Global Quality Preference. Suggestions for the evaluated web-application.
Figure 2: Design of the Quality Evaluation Framework

Figure 4. Homepage of http://www.ebay.in
4 Implementing the quality evaluation framework

The evaluation process is done by using small, controlled and well-planned experiments [22], used the students of MCA and also other software professionals working with the companies of international repute. They have been given a list of on-line shopping Indian web-sites and were provided the evaluation scale for each metric. After evaluating the data of selected web-sites of Indian e-commerce domain, an evaluation model has been selected. We have selected non-linear multi-scoring criteria model. The evaluation process is based on Logic Score of Preferences LSP[23,24] and Continuous Preference Logic as mathematical background. This can be achieved by means of a preference aggregation function, called generalized conjunction /disjunction or andor, combining weighted power means to obtain the Global Quality Preference. After evaluating the factors of quality of the selected web-sites Global Quality Preference is being calculated.

5 Results and discussion

In this work, we have presented the design and implementation of a systematic and quantitative engineering-based evaluation method for web-applications. The block diagram given in Figure 3 shows the quality factors along with their weights in Global Quality Preference of the web-site. It is expected that organizations need to define their own quality factors, criteria and metrics specific to their system context. In future, we suggest creation of a knowledge base of web-based software’s quality factors, criteria, and metrics. Software professionals can then make use of already defined model that suits them or find the closest model and modify it according to their needs. We have evaluated the web-site of ebay http://www.ebay.in and the results are shown in Table 2. We further wish to evaluate other selected Indian e-commerce based web-sites to calculate their Global Quality Preference, in order to rank these web-sites according to their quality.

Table 2. Quality preference of factors of http://www.ebay.in

<table>
<thead>
<tr>
<th>URL of the web-site</th>
<th>Usability %</th>
<th>Functionality %</th>
<th>Reliability %</th>
<th>Efficiency %</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.ebay.in">http://www.ebay.in</a></td>
<td>75.6</td>
<td>87.23</td>
<td>91</td>
<td>93.3</td>
</tr>
<tr>
<td>Global Quality Preference</td>
<td>85.16%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
6 References


[7.] Ministerial NETwoRK for Valorising Activities in digitations, project web site: http://www.minervaevurope.org/


***
SESSION

EVALUATION

Chair(s)

TBA
Conformance Testing and Interoperability: A Case Study in Healthcare Data Exchange

L. Gebase¹, R. Snelick¹, and M. Skall¹
¹National Institute of Standards and Technology (NIST), Gaithersburg, MD, State, USA

Abstract - Correct data exchange is critical for ensuring reliable healthcare systems. Standards based systems are the foundation for achieving this goal. However, standards alone are not enough to ensure this promise; conformance and interoperability testing are essential. We present and compare conformance testing strategies for a widely used healthcare clinical data exchange messaging standard. We discuss in detail an actor-based testing framework and give insight on the approach used in developing the framework. We present an architecture that extends this framework to support testing of integrated healthcare systems using multiple messaging and document data exchange standards.

Keywords: Conformance; Interoperability; Messaging Systems; Testing Framework; Test Strategies.

1 Introduction

A major challenge for the healthcare industry is achieving interoperability among proprietary applications provided by different vendors. Each hospital department may use multiple applications to share clinical and administrative data. Interoperability can be better achieved through the use of standardized interfaces. Even though the applications may implement the same standard, there is no assurance of interoperating. There are two primary reasons for this problem. One is that the applications don’t implement the same set of options allowed by the standard. This problem can be addressed with conformance provisions offered by the standard. The second problem is that applications implement the standard incorrectly. This is addressed with conformance testing. Applying conformance processes and successfully conducting conformance testing will not ensure interoperability, but they will increase the likelihood of implementations interoperating. Employing a comprehensive testing program at the onset of an implementation leads to more reliable systems, and ultimately, reduced costs.

We propose and examine testing strategies for the Health Level Seven version 2.x (hereafter HL7) messaging standard [1]. HL7 is a widely used standard for the exchange of clinical and administrative data among healthcare applications. HL7 provides an interesting testing challenge due to the wide array of options allowed by the standard. To reduce the number of choices implementers are confronted with and increase the likelihood of different implementations interoperating, the HL7 standard has introduced a conformance section that allows implementers to support a subset of the functionality offered by the standard. By reducing the large set of options allowed by the standard, implementers are able to significantly increase the likelihood of interoperating. The principle mechanism to constrain the allowed set of options is a message profile. Message profiles not only aid interoperability, they also enhance the capabilities and effectiveness of conformance testing and the overall testing process.

We are interested in establishing conformance metrics for HL7 implementations and evaluating vendors’ adherence to those metrics in a pragmatic environment designed to simulate a real world environment that does not require changes to the vendor implementation. We examine two approaches for evaluating conformance. One approach employs an Upper Tester, which sits above the application being tested and makes use of whatever user interface—possibly an application programming interface (API)—that the application provides, along with a Lower Tester which acts as a peer application and drives the testing. The second approach we examine employs actors to interact with the application being tested. Actors are autonomous, relatively small modules, generally run on separate execution threads that support a well defined subclass of the total functionality defined by the standard. Finally, while our initial focus is on HL7 testing, our objective is to develop tools and methodologies that can readily be applied more broadly to environments outside of HL7.

2 Conformance and Interoperability

Standards, no matter how good they are, are just pieces of paper. They are a means to an end. The goal of any standard is the eventual binding of the requirements in the standard into correct, reliable software. To accomplish this goal, the standard must be a clear, precise, unambiguous, complete and testable enumeration of detailed requirements. Using the English language to provide this detailed specification is a challenge in itself because English is not a precise language and lends itself to ambiguities. There are, however, principles that one can implement to help ensure a
precise, testable standard. A good standard should 1) define what/who needs to implement the standard, 2) distinguish between normative (mandatory) and informative sections of the standard, 3) use universally accepted key words to specify requirements, 4) be modular with minimal redundancy, 5) be adaptable as things change, and 6) be technology and design-independent. If a standard encapsulates these principles it stands a good chance of being implemented correctly. However, in order to substantially increase the likelihood of developing correct implementations, tests need to be developed to determine conformance.

Conformance is defined as the fulfillment of a product, process, or service of specified requirements [6]. Conformance is essential to any standard because it specifies who needs to conform to the standard and what they need to do to claim conformance.

Conformance testing is a way to determine directly or indirectly that all relevant requirements in a standard have been implemented correctly. Conformance testing is black box testing. In black box testing, the tester does not have knowledge of the implementation’s internal structure or have access to the source code. A tester examines that requirements have been met by probing the implementation through a series of test cases comprised of both valid and invalid input and examines the output for correctness, as defined by the standard. This is contrasted with white box testing where the internal structure of the code is known to the tester. With white box testing, the tester chooses inputs that exercise paths through the code in order to determine if the implementation is working correctly.

There is a relationship, much like a three-legged stool among standards, implementations, and conformance testing. If one leg of our stool does not work correctly, we will not have confidence that our requirements have been faithfully implemented. The implementation is tested (via conformance testing) against the requirements in the specification to determine if all requirements are met. There are only two possible outcomes. If any of the tests result in at least one error, then we know to a certainty that the implementation does not conform. However, ironically, if the implementation passes all of the tests, we don’t know anything for certain. Either the implementation does indeed conform or the tests are not comprehensive enough to find the non-conformity. This is another way of stating that conformance testing can never be exhaustive. Conformance testing can only prove the presence, not the absence, of errors.

The goal of interoperability testing, on the other hand, is to ensure that diverse systems can “work together” and thus interoperate. In the world of messaging standards, interoperability will result in implementations reliably exchanging messages without error. Note that conformance testing is a pre-requisite for interoperability testing since we need to ensure that the information being exchanged is the correct information. However, interoperability testing requires another layer of testing after conformance has been ascertained. Interoperability plus conformance ensures that both systems are speaking the same language. Systems can send and receive messages and respond with appropriate messages and ultimately incorporate the information into their systems and workflow.

Conformance testing is often (but not always) performed by testing laboratories, with a resulting issuance of a certificate to implementations that pass all the tests. This process is called certification. However, even if certification is not the goal, conformance testing is still necessary, since it is the only way to ascertain if requirements in the standard have been correctly implemented. Additionally, conformance testing serves as a communication between buyers and sellers allowing sellers to substantiate their claims and buyers to increase their confidence in the product.

![Figure 1: Certification Building Blocks](image)

There is a relationship among the standard, conformance testing, conformity assessment and certification as a set of inter-connected building blocks much like a Russian nested doll with the standard as the inter shell (see Figure 1). None of the higher-level blocks can be performed unless the box beneath it has been completed. Thus, conformance testing can not be performed unless the standard (with its conformance clause and clear, testable requirements) has been completed. Conformity assessment (processes and policies for testing) can not be implemented until the standard and the conformance testing test suite are in place. Finally, certification can only be accomplished when all of the three lower level building blocks are in place. Also, one can stop anywhere along this spectrum. Many standards exist without conformance testing or certification. Some standards have associated conformance tests but no certification regime while some standards contain all the components all the way up through certification.
3 Testing HL7 Healthcare Systems

Typical healthcare organizations have many proprietary heterogeneous information systems that must exchange data reliably. Not only are the systems heterogeneous but standards for exchanging data among them are different. Seamlessly sharing data among systems and testing them is complex. In this section we focus on homogeneous systems for the exchange of clinical data using the HL7 messaging standard. We provide an overview of HL7 version 2 and our conformance approach, an analysis of testing strategies, and tools that facilitate testing and interoperability. We then focus on an actor-based testing framework. In the section that follows, we extend the framework to a heterogeneous healthcare system that uses multiple message types and employs more than one document exchange standard.

3.1 HL7 and Conformance

The Health Level Seven (HL7) version 2.x is a data exchange messaging standard for moving clinical and administrative information among healthcare applications [1]. Typical HL7 messages include admitting a patient to a hospital or requesting a lab order for a blood test. HL7 messages are structured hierarchically, but the hierarchy is limited to exactly four levels and composed of building blocks generically called elements. These elements are segments, fields, components, and sub-components. Each element has associated attributes that may constrain it. These include the degree of options allowed, repeatability, value set, length, and data type attributes. Segments can contain additional elements, fields and components can contain additional elements or be primitive elements; sub-components are strictly primitive elements. Primitive elements are those that can hold a data value and have no descendant structure. Additionally, a container element called a group can be used to group a related collection of segments.

This four-tiered hierarchical structure appears simple enough, but the real complexity in the message structure is revealed when the possible sequence of segments and fields making up a message is examined [8]. Every HL7 message can be identified by its message type. The type limits the segments allowed in the message, but generally, even for a specific message type, a great deal of variation is possible. Segments may be designated as required or optional; required segments may also be allowed to repeat an arbitrary number of times, or they may be required to repeat a specified number of times. Optional segments may be absent or they may be present and repeat an arbitrary number of times. For any message, a segment present in one instance of the message may not be present in another; repeating segments may occur multiple times in one instance and not at all in another. The message content is further complicated by the fact that the sequence of fields making up each segment may themselves be optional or required and also may or may not repeat. An application capable of processing messages of one type may be incapable of processing messages of a different type, and an application capable of processing a specific message type may not be able to process all instances of the type. Clearly the realm of message possibilities is large and for applications to have a reasonable chance of interoperating, the spectrum of possibilities has to be constrained. The conformance section of the HL7 standard has defined a message profile (also commonly referred to as conformance profiles or profiles) for precisely this purpose.

Message profiles define processing rules and provide an unambiguous description of HL7 messages. Vendors agreeing to a common profile are more likely to interoperate. Furthermore, the profile provides a measure for evaluating the validity of the messages exchanged among vendors. Vendors may employ tools specifically designed to facilitate message validation. This may be the first step in the overall process of evaluating conformance. A message profile can be represented as an XML document (see the profile snippet shown). The document includes each element allowed in the message along with its associated attributes. For a more detailed description of a message profile refer to version 2.5 of the HL7 standard [1].

Profiles reduce the number of possibilities to a manageable set, and their use helps to ensure that systems attempting to communicate with each other implement compatible sets of possibilities. A profile defines a set of constraints on the options allowed by the standard. When the profile is specified in XML, it also may be machine processed, thereby greatly facilitating the effort required to produce an implementation and reducing the likelihood of errors.
3.2 HL7 Conformance Testing

Conformance testing focuses on evaluating an implementation's external behavior and assessing its adherence to the standard. For HL7 implementations, assessing external behavior generally encompasses determining the implementation's state and evaluating the content of the data it transmits in its current state. Messaging protocols are typically stateless, but nevertheless an HL7 application can be treated as simple state machines. Initially a responding HL7 application is in a wait state, ready to receive HL7 messages from an initiating application. On receipt of a message, it transitions to a send state in which an acknowledgement message must be returned to the initiating application. An initiating application reverses the responding application's state transitions and is initially ready to send and then transitions to a wait state.

To evaluate a responding application, it is necessary to measure the content of messages it receives and the acknowledgement message it returns. To ensure the content of sent messages is correct, the messages must first be validated. Validation encompasses message parsing to ensure correct structure and syntax and semantic checking to ensure values are correctly constrained. Criteria that must be satisfied by the returned message can be established from the message that was sent. However, the content of the returned message may also depend on the state of the responding application's database. To track the content of the database, it is necessary to control the initialization of the database and then track changes to it.

Evaluating the behavior of an initiating application largely reduces to validating the messages sent by the application. But when testing both initiating and responding applications, the ability to measure the content of messages sent by an initiator may be limited. In general the exact content of the message cannot be determined without knowing the user request that triggered the sending of the message, but without access to the application's user interface, this is often not possible. Nevertheless, in the case of HL7, message validation with some limitations can still performed. This is possible if the HL7 message profile is used to enforce proper message structure and syntax and to enforce value constraints. It is also possible to conduct more robust testing by constructing messages with invalid values, or with an invalid structure, and evaluating an implementation's reaction to the receipt of the invalid messages.

3.3 Testing with a Lower and Upper Tester

One commonly employed approach to black box testing is to place the implementation being tested—commonly referred to as the system-under test (SUT)—between a so called Upper Tester (UT) and Lower Tester (LT). Figure 2 illustrates the approach.

The approach has been widely employed in conducting protocol testing. With this approach, the SUT communicates with the LT via the protocol defined by the specification and the UT takes the place of the user or the business application supported by the SUT. No additional requirements are placed on the SUT to enable communications with the LT, since the environment does not differ from the operational environment the SUT would otherwise function in. The LT drives the testing. Acting as an initiating application, the LT sends messages to the SUT and evaluates the SUT's behavior based on the acknowledgment messages returned from the SUT. The UT may be used to evaluate the SUT's service interface. The LT may also direct the UT to issue requests to the SUT and in this way the SUT's role as an initiating application can be evaluated.

This approach allows for effectively evaluating the externally observable behavior of the application, but places no requirements on the internal structure of the application or the methods it uses to satisfy the requirements.

Employing an LT to replace a peer application in this environment can usually be achieved without difficulty, but the same is not true for employing an UT. An UT must typically be deployed outside the tester's local environment, in the environment running the SUT. This places requirements on the SUT that may not be easy to achieve, particularly when a standardized user interface for the SUT
is not defined, as is often the case. Testing can still be conducted without the use of an UT, but it does place some limitations on the capabilities of the test system and what can be tested.

A further limitation of this form of black box testing is that it cannot be applied in an environment made up of multiple communicating applications, nor can it be used if there are multiple systems to be tested simultaneously.

3.4 Actor Based HL7 Test Framework

A typical HL7 environment is made up of many communicating HL7 applications. A representative environment is shown in Figure 3.

Actors are autonomous implementations, typically running on separate execution threads that the testing framework employs in place of HL7 applications needed to simulate the operational environment in which the SUT functions. In general HL7 actors can serve in place of any HL7 system. This enables the construction of an environment completely controlled by the test system that mimics the real world operational environment in which the SUT operates. In this environment the test system can track and monitor all message exchanges with the SUT. The actor based architecture is shown in Figure 4. To the SUT and any other HL7 implementation that is part of the testing environment, the actors are indistinguishable from HL7 applications they might interact with in an operational environment. The actors are distinguishable from other HL7 systems only in that they are driven by a test script and provide complete logging of all activities.

According to Figure 5, making up the operational environment. Figure 5 shows a scenario in which message transactions begin with the ADT actor sending a message to the Order Placer. The receipt of this message by the Order Placer SUT is expected to trigger a series of message exchanges. The Order Placer must acknowledge the message sent by the ADT actor, and in addition it is expected to send a message to the Order Filler. But there are no differences between the Order Filler actor and the Order Filler itself that are apparent to the SUT. Thus, no changes are necessary to test the Order Placer.

The messages exchanged in this environment are constrained by a message profile that all participating entities agree to. The profile restricts the set of messages exchanged from the broad spectrum allowed by the standard to a manageable set.

This actor based methodology to testing offers some advantages over the Upper-Lower Tester approach. It is easily extensible; regardless of how many applications are employed in the operational environment, actors can always be employed to replace them in the testing environment. The approach also lends itself better to deployment in more complex environments, such as the environment depicted in Figure 4 where more than one SUT is being tested simultaneously. Actors and applications may be mixed arbitrarily in this environment, which cannot be readily accomplished in the Upper-Lower Tester environment.

Although message validation is not strictly required prior to performing conformance testing, validating messages in advance can greatly facilitate the process. If it is not performed in advance, it must be conducted simultaneously with the testing. Evaluating message content is critical to accurately conducting conformance testing. If the test system is to evaluate the message responses returning from an HL7 system, it cannot do so accurately unless it is certain of precisely what the content of each message is.

The testing framework does not vary depending on whether or not message validation has been done in advance. However, message validation can be turned off if it was done previously, and, if it is employed, testing should proceed smoothly when validation has been done in advance.
As with the black box testing described above, testing with actors is used to evaluate only external behavior. The approach does not employ any counterpart to an UT and this places some limitations on what can be tested; messages sent and received can be evaluated, but evaluating the functionality provided to the user or the business application cannot be done. Without a counterpart to an UT, triggering an HL7 implementation to initiate a message exchange can be problematic; it may be possible as the result of receiving a message, but short of this some means of accessing the application's service interface is necessary.

3.5 Message Profiles and Interoperability

A message profile applies implementation specific constraints to the standard that eliminate the potential ambiguities that the standard permits as implementation alternatives and thus increase the likelihood of implementations interoperating [3]. Message profiles are an integral part of a testing framework. They provide the message template upon which better test message generation and message validation can be performed.

A desktop tool for creating and documenting message profiles in a common format is the Messaging Workbench (MWB) [2]. The MWB supports all the HL7 version 2.x artifacts in the form of libraries that are readily available within the tool for use in message profile composition. An XML representation of the message profile is an important output of the tool.

An important aspect for achieving interoperability is determining if communicating applications have correctly implemented an interface based on a message profile. To achieve this, the existence of a well-defined and extensive set of test messages is paramount. At NIST, we have developed utilities for message generation that can be incorporated into a testing environment. The utilities are delivered as a collection of APIs, web services, and as a desktop application called Message Maker [3,4,5].

A critical function of testing is message validation utilities. NIST has developed tools to validate messages instances based on a message profile and has extended functionality to support content testing based on a given test scenario. A Java message validation API provides the core functionality. Additionally the functionality has been built into a desktop application, web services [9], and a web application [10]. The APIs and web service APIs can be used to integrate validation services into a testing framework.

4 Extending the Testing Framework

We have described an approach for assessing the conformance and interoperability of HL7 healthcare systems utilizing a set of actors designed to simulate HL7 application behavior. However, healthcare organizations exchange data using a number of messaging and document standards—since there is not a single standard to cover all aspects of data exchange among healthcare systems. Some standards are needed for moving clinical data, others for medical images, and yet others for personal health records, for example. A testing framework can be used to evaluate systems with complex integration requirements. The Integrating the Healthcare Enterprise (IHE) initiative [7] has defined numerous integrated test scenarios for various healthcare domains (e.g., radiology). IHE hosts an annual connect-a-thon event [12] where numerous vendors implementations are evaluated using the testing scenarios. The IHE Gazelle project [13] is an effort to develop a testing framework to automate the testing of the integrated test scenarios. The Gazelle framework extends the actor based testing approach to a heterogeneous environment.

![Figure 6: Gazelle Testing Architecture](image-url)
Since the Gazelle system is made up of multiple diverse systems with no common protocol, the test engine is faced with the problem of how to communicate with each system without requiring support for a separate communications technique for each system that is part of the environment. To deal with this problem, a web service interface is employed. Each actor in the system supports a web service interface that the test engine uses to communicate with the actor. The challenge in defining the web service interface is to define a suitable interface that is common to all actors, rather than employing different definitions for each type of system. Since the common thread running through each homogeneous system is that it is actor based, it is possible to abstract the requirements so that a single interface will serve the requirements for all systems. Clearly doing so not only simplifies development of the test engine, it also means that it can readily be extended to incorporate any number of new systems.

5 Conclusions

We have analyzed two techniques for conducting conformance testing, one employing an Upper Tester and Lower Tester, the other actor based. We have shown that the Upper-Lower Tester method can be effectively utilized in conducting conformance testing, but that the method does not scale well to environments incorporating multiple applications or requiring multiple systems to be tested simultaneously. For these systems we have shown that actors—autonomous systems providing limited, but well defined functionality—can be effectively employed for testing. Moreover, we have shown that the actor based approach can be readily extended beyond the HL7 environment. We have also shown that properly conducting conformance testing requires careful evaluation of message content and that while conformance testing cannot ensure that implementations that undergo successful conformance testing will interoperate, it will increase the likelihood of them doing so.

6 References


[4] Message Maker; Developed by the National Institute of Standards and Technology (NIST) in conjunction with the HL7 Conformance Special Interest Group; http://www.nist.gov/messagemaker.


[13] Gazelle Testing Framework. A collaboration effort led by S. Moore (Washington University of St. Louis) and E. Poiseau (INRIA) to build a testing framework to support IHE test scenarios.

Assessing the Complexity of Software Architecture Using Coupling and Cohesion

Hassan Reza, Joel Carver, and Emanuel S. Grant
School of Aerospace Science
University of North Dakota
Grand Forks, North Dakota. 58202, USA
reza@aero.und.edu

ABSTRACT
In this work, we propose a qualitative method to assess the complexity of software system at the architectural level. More specifically, we focus on the roles architectural constructs, such as components and connectors, play in the assessing the complexity of software system. To this end, we have introduced a qualitative framework to measure the coupling among elements of a system that work with different architectural styles.

Keywords: Software architecture, coupling and cohesions, software coordination, architectural styles, software complexity

1. INTRODUCTION
The most important step in solving any complex problem is first to understand the problem. The next step is designing the solution by selecting the right architecture. Software architecture specifies the behaviors of a system in terms of computations performed by components, communication between these components, and their overall configurations.

In this work we attempt to answer the following research question: Given the factors that may affect coupling and cohesions at the architectural level, is it possible to evaluate the complexity of architectural styles? A software system is considered complex because of the inherent complexity of the problem domain and the development method. The complexity related to the problem domain, for most part, is related to the computations and communications, and constraints and configuration aspects of the system under construction. The communication among components which indicates the increased interactions and coupling of a system plays a big role in other aspects of system level qualities, such as the testability, reliability, maintainability, reusability [22]. In focusing at complexity there are two elements (at least) to be studied: cohesion and coupling [3]. When looking at coupling we are looking at the level of commitment and agreement between two interacting components. A lower degree of coupling is the clear indication of the lesser degree of commitments and hence freedom in any relationship, which in turn, is the indication of lesser complexity [2][3]. When addressing software complexity in terms of cohesion, we are looking for a high cohesion and glue among the internal elements of a system or subsystems working on one common task [2] [4]. This, in turn, indicates that a high cohesion among system elements increases understandability because it is less complex.

2 RELATED WORKS
As far as the authors are concerned, the work related to the notion of coupling and its role in the assessing the complexity of software architecture is very rare or nonexistence. Therefore, this work has initially been influenced by original work in the traditional structured design [19] and objected oriented design [8], [20], [21]. The work related to the assessing the complexity of software architecture by [6] has also influenced this work, and [22], which attempts to clarify the notion of software connectors and the relationship between coupling and software connectors.

3 METHOD
There exist many definitions and interpretations for software complexity yet there is no general agreement about what really constitute the complexity of a system. Complexity, for most parts, stems from both the logical complexity of software elements (i.e., components having too many states or having too many obligations) and the complex relationships (i.e., very unpredictable and non-
obvious way) among even small number of components in small systems. In general a system is said to be complex if the system consists of more than a handful of components that interact in nonlinear and unpredictable ways. Examples of the complex systems includes safety and mission critical systems such as avionics and aerospace systems, military and weapon guided systems, and medical monitoring systems.

Complexity is one of the key factor in generating bugs and errors. In general, the more complex and difficult the software system is, the more challenging it is to understand it and hence make it reliable.

To quantify complexity, we need to identify factors that may have certain impact on the complexity of software architecture. For example, the complexity factors attributed in object-oriented software design include complexity associated with inheritance and polymorphic dependencies, which increase the number of possible object type that may interact with different and totally wrong implementation, complexity related to implicit control dependency, which is hard to identify and possible number of concurrent access to shared memory and call interactions.

3.1 Design Complexity

Improper decomposition and modularization technique to decompose a system has direct correlation with complexity, because it may result undesirable degree of dependency. This, in turn can increase complexity of the system. The complexity, in turn, has direct impact on the level of effort needed in both front-end engineering, such as design, and back-end engineering, such as software validation and maintenance.

Traditionally, proper decomposition and modularization techniques based on separation of concerns and locality have decreased complexity. For example, layered architectural style has shown to be effective in using proper modularization to manage and hence reduce complexity because it models the system using divide and conquer, and separation of concerns design strategies. This decomposition, in turn, provides a context in which no designer is responsible to understand all the details. Therefore, it makes sense to include localization, divide and conquer, and separation of concerns to manage the complexity.

3.2 Complexity, Cohesion, and Coupling

Software design coupling (dependency) and cohesion (glue or stickiness) have been used as qualitative criteria to measure the complexity attribute of software at module level by describing the goodness and effectiveness of the low level design (i.e., modular design). Cohesion (or computation independence) is a direct dividend of design principles such as modularity, abstraction, and locality, separation of concerns, and information hiding and encapsulation [8]. A component being cohesive is identical to saying that a component accountable and has a crisp abstraction with explicit responsibility.

Cohesion refers to the goodness of glue among the internal elements within a component to address a specific set of requirements corresponding to a specific functionality. The goodness of cohesion can be evaluated against the variations (or dimensions) of cohesion such as coincidental, logical, temporal, procedural, communicational, and functional where coincidental is perceived to be the worst case of a cohesion, and functional is considered to be the best form of a component cohesion [9].

Cohesion in object-oriented design uses the cohesion of classes. For example, the lack of cohesion in methods (LCOM) [15], which is used to measure the sickness of methods defined in a class, is one of the most established and studied work.

In contrast to cohesion, coupling (intermodular connections and bindings) refers to the degree of connections needed among modules in order to perform their tasks. Two components, say C1 and C2 are said to be closely coupled if understanding the behaviors of C1 requires understanding the behaviors of C2 [10].

In classical structural design, the goodness of components coupling is evaluated against the variations or dimensions of coupling such as content, common, control, stamp, data, coupling where content coupling and data coupling represent the high and low coupling respectively [9]. Tight (high) coupling among the elements of system also makes it difficult to understand and hence to modify the system.

3.3 The Role of Coupling and Cohesion at Architectural Level

A persistent problem in designing complex system is how to measure and manage the complexity of
the design. The complexity issue is constantly tackled by the introduction of innovative software models and processes. Software architecture is an example of such a new model that has received much attention, because it facilitates a specific view of a system in which design qualities such as understandability, simplicity, reusability, and analyzability can be achieved.

Software architecture has many definitions. Example includes component and connector definition [13]. In this work, we have adopted the component and connector definition of software architecture in which the software architecture of a system is defined in terms of architectural constructs, namely, components (or locus of computation), connectors (or locus of communications), and configurations (pattern of compositions) [12][13]. In general, components represent standalone black-box computational entities (i.e., they can be executed autonomously) with well-defined interfaces and well-specified semantics. At architectural levels, components are perceived as subsystems having their own configurations.

Complexity assessment for components at architectural level based on component and connector [13] definition requires to investigate the cohesion of the subcomponents (i.e., autonomy and semantically meaningfulness), the coupling (i.e., dependence and commitment) of connectors. The collective assessments of cohesion and coupling for both components and connectors can help software architect to select less structurally complex software architecture that satisfies desirable properties (functional and nonfunctional) of a system.

Effective software architectural design, among other things, demands both components, which are the locus of computation and the unit of cohesion, to be functionally independent, and connectors, which are the locus of communications and the unit of coupling, to be treated separately from components. In the context of software architecture, a highly cohesive component refers to a component that must address a set of architecturally significant requirements.

We have adopted the universal design principle as the criteria to assess the cohesiveness of components [14]. The criteria are general enough to be used as the factors influencing the cohesion of components at architectural. The criteria include:

1. **Degree of separation of concerns**: It is a design strategy dealing with aspects (or concerns) in isolation. Examples of concerns include quality attributes, views, parts, etc. [14].

2. **Degree of modularity**: The relative decomposition of a complex system into a set of cohesive and loosely coupled parts. It helps separation of concerns when we are refining the component and integrating it with other components.

3. **Degree of generality**: A concrete component has a high degree of cohesiveness comparing to a very generic (or parametric) component which can be instantiated in many ways.

4. **Degree of information hiding and encapsulation**: It provides explicit shield between abstract and concrete representation of a component. This, in turn, will provide a level of privacy to a component and its external environment.

5. **Degree of data persistence**: The degree by which object (data or local variable) exists after the execution of component.

6. **Degree of concurrency**: A component may run on a single process or multiple processes. A component executing across multiple CPUs and address spaces is more complex and less cohesive than the ones running on a single process and having a single address space.

From conceptual point of view, components at architectural level represent standalone black-box elements having the highest level of cohesion, but their structural refinements and implementations by the corresponding low level of design can be less cohesive.

Components can communicate with each other using different types of connections as defined in connectors’ taxonomy in [11]. Connectors can be simple or complex (or composite). Examples of simple connector types and connections include: procedure call, event, shared memory data access, linkage, stream, arbitrator, adaptor, distributor connection. Complex and composite connectors are those connectors that can be organized hierarchically. An example of a complex connector includes OSI communication protocol.

An example of a connector with a high degree of coupling and complexity include interaction and coordination among components accessing a shared memory. This type of connector requires certain degree of synchronization and coordination among components racing to access the same data item at the same time. An example of a connector with a low degree of coupling and less complexity is asynchronized flow of information between
Coupling among components at architectural level is affected by the coordination mechanisms that are rules or protocols controlling the communications and interactions between components. More specifically, coordination mechanisms require both communication and synchronicity [17]; these are critical elements of coordination mechanisms in real-time and concurrent system [15].

Synchronicity plays an important role in the dependency among components at architectural level. Synchronicity refers to the degree of blocking (or waiting) of a component until certain specified condition becomes satisfied.

Communication is the mechanism by which information is transferred between components. An example includes a synchronized communication (call or chat rooms) which requires time coupling among all interacting components. Asynchronized communication does not require time coupling, and requires less binding among the participating components. Example includes sending email.

To better characterize the notion of coupling associated with connectors and to understand their behaviors, we have examined and adopted some of the features relevant to the notion of coupling defined in the connector classification framework discussed in [11], [15], [17] as critical factors that may influence the coupling between components.

Therefore, the factors that may influence coupling using coordination can be divided into 1) communications and, 2) synchronicity factors. Each factor, be it communication or synchronization factor, in turn, can be further characterized by sub-factors. Each subfactor may have only one symbolic value assigned from a set of symbolic values (i.e., low, medium, and high). For example, a coupling value for a method invocation in event-based systems is low (or loose), because an event will affect those components (or listeners) who are vested in the event. On the other hand, the coupling value for the communication using a shared memory (or state) is tight, because any changes made to the value of a shared memory may affect other components.

In what follows, we define the factors affecting coupling based on the coordination mechanisms, which consists of communication and synchronization factors:

1. Synchronicity of connector (blocked, semi-blocked, and non-blocked): Synchronicity and communication are critical elements of coordination mechanisms in real-time and concurrent system. Synchronicity can be divided into classes: synchronized (blocking) and asynchronized (no blocking) ones. The synchronized mechanisms are composed of exclusion and priority constraints as discussed in [18]. Exclusion ones block components when certain condition is true. Examples of synchronization mechanism enforcing exclusion are semaphores which restrict access to the shared resources in multiprogramming environment, or mutual exclusion to restrict the concurrent access of a shared resource. Prioritization ones have to do with serial ordering (or scheduling) of the tasks/requests executing concurrently using time stamps or locking mechanism.

2. Naming (or identifying): It is an important coupling factor; it is related to the manner by which components engaging in a communication are identified.

3. Concurrency (sequential, parallel): sequential system requires higher degree of coupling and synchronization, but parallel and distributed system demand high degree of coordination.

4. The cardinality of interface (the size of interface): The total number of interaction points characterizes the degree of dependency and complexity among the components. To this end, Fan-in/Fan-out which represents the total number of roles (In and Out), DF/US which represents the total number of define-use connection points, and AN/RG which represents the total number of announcers and registers have been used to measure the complexity of interface.

5. Mode of Connection (Point-to-Point, Multicast, and Broadcast): Point-to-point connections (e.g. P/F) provide 1-to-1 connectivity between two nodes or machines using link or channel. A 1-to-m (or point-to-multipoint connections) provides one-to-many connectivity between one node and one or more nodes using a set of routing paths. Finally, a m-to-m (or multipoint-to-multipoint) supports a very complex interaction by providing many-to-many connections among a set of computer connected by a complex communication protocol.

6. Type of Dependency: This factor represents various type of dependencies associated to change (e.g. change at run-time, compile time, and source code). For example a shared memory (SM) dependency is dependency due to change to shared memory affecting changes.
to all knowledge sources or programs. Method invocation (MI) is a connection dependency; it is a dependency at invocation-time (run-time). Push-and-poll (PP) is a dependency heavily used in the data streamed based systems such as consumer and producer system. Compiled dependency (CD) is a dependency at compile time.

7. **Type of Access Control**: The manner by which components collaborate and access resources have a direct impact on coupling and dependency among them. To this end, we have identified the following access controls. Directed access control refers to the access control mechanism that requires an authorized component to access the system shared resources and services. Examples of these mechanisms include semaphore, monitors, mutual exclusions, authentication protocols. Queue (QU) is an access control that is being used for both control and data channels. In the event a queue is full, then the producing components are blocked to access the queue; in the event a queue is empty, then the component playing the role of a consumer is blocked to access the queue. More complex queues have also been used in the federated distributed systems and wireless mobile system to control both data and messages. Simple queues working as links have also been used in layered and pipe and filter architectures with various levels of simplicity and dependency.

8. **Dependency direction**: It is an important factor to highlight the direction of change dependency among components. We have identified three types of change dependencies, namely, unary, binary, n-ary. Unary dependency direction is a dependency change that may require a change in a behavior of one component affects the behavior of another component. Example of unary-direction includes Pipe/Filter in which a change in the behavior of a component playing the source (signaling the end-of file) may require a change in a behavior of a component playing the role of sink. Binary is a dependency change is bidirectional. Example is client/server in which any changes in a server enforce a dependency change on client or the other way around. N-ary dependency change requires a change in one component changes n-components related to this component.

In table 1 we identify the attributes relevant to the above factors (i.e., factors for both communications and coordination) for architectural styles such as blackboard (BKD), Simple client server (SC/S), Replicated Clients and Servers (RC/S), pipe and filter (P/F), Layered (LYR), Explicit Object Oriented Explicit (OEI), and Implicit Innovation Methods (OII) [13]. For example, in table 1, the synchronicity factor for the blackboard architecture has been specified as an asynchronous because knowledgebase sources can access the shared memory in the opportunistic and non-blocking manner. In case, there is a contention for a certain shared resource, then the accesses to these resources are controlled to prevent any inconsistency.

In table 2 we assigned and quantified the symbolic values (e.g. Low) for each factors assigned in table 1. For example, a symbolic value of low, which is quantified by 1, is assigned to the naming attribute of blackboard, because BLK does not require any naming constraints for knowledge sources attempting to communicate with one another. In table 2 the columns having high total coupling number implies the complexity of a style. For example, the total couplings value for the blackboard is 19 indicating the relative complexity of this style comparing to pipe and filter, which is 10.
### Table 1: The Relationship between Architectural Styles and Coupling Factors.

<table>
<thead>
<tr>
<th>STYLES</th>
<th>BLK</th>
<th>SC/S</th>
<th>RC/S</th>
<th>P/F</th>
<th>LYR</th>
<th>OEI</th>
<th>OII</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronicity</td>
<td>Asynch</td>
<td>Asynch</td>
<td>Synch</td>
<td>Asynch</td>
<td>Synch</td>
<td>Synch</td>
<td>Asynch</td>
</tr>
<tr>
<td>Naming</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Concurrency</td>
<td>PR</td>
<td>SQ</td>
<td>PR</td>
<td>SQ</td>
<td>SQ</td>
<td>SQ</td>
<td>PR</td>
</tr>
<tr>
<td>Cardinality</td>
<td>FI/FO</td>
<td>DF/US</td>
<td>DF/US</td>
<td>FI/FO</td>
<td>FI/FO</td>
<td>DF/US</td>
<td>AN/RG</td>
</tr>
<tr>
<td>Mode of Connection</td>
<td>m-to-m</td>
<td>1-to-1</td>
<td>m-to-m</td>
<td>1-to-1</td>
<td>1-to-1</td>
<td>1-to-1</td>
<td>1-to-m</td>
</tr>
<tr>
<td>Type of Dependency</td>
<td>SM</td>
<td>MI</td>
<td>MI</td>
<td>PP</td>
<td>CD</td>
<td>MI</td>
<td>MI</td>
</tr>
<tr>
<td>Type of Access control</td>
<td>DA</td>
<td>DA</td>
<td>DA</td>
<td>QU</td>
<td>QU</td>
<td>MI</td>
<td>MI</td>
</tr>
<tr>
<td>Dependency Direction</td>
<td>MD</td>
<td>BD</td>
<td>BD</td>
<td>UD</td>
<td>BD</td>
<td>BD</td>
<td>BD</td>
</tr>
</tbody>
</table>

**Keys:**

- SM: shared memory
- MI: Method Invocation
- FI/FO: Fan-in/Fan-out
- DF/US: Define/Use
- PP: Push-and-Poll
- AN/RG: Announcer/Registers
- DA: Directed access
- BF: Buffering
- QU: Queuing

### Table 2: The Comparison of architectural styles using coupling values.

<table>
<thead>
<tr>
<th>STYLES</th>
<th>BLK</th>
<th>SC/S</th>
<th>RC/S</th>
<th>P/F</th>
<th>LYR</th>
<th>OEI</th>
<th>OII</th>
</tr>
</thead>
<tbody>
<tr>
<td>synchronicity</td>
<td>M=2</td>
<td>L=1</td>
<td>H=3</td>
<td>L=1</td>
<td>M=2</td>
<td>M=2</td>
<td>L=1</td>
</tr>
<tr>
<td>Naming</td>
<td>L=1</td>
<td>H=3</td>
<td>H=3</td>
<td>L=1</td>
<td>M=2</td>
<td>M=2</td>
<td>L=1</td>
</tr>
<tr>
<td>Concurrency</td>
<td>H=3</td>
<td>L=1</td>
<td>H=3</td>
<td>L=1</td>
<td>H=3</td>
<td>H=3</td>
<td>L=1</td>
</tr>
<tr>
<td>Cardinality</td>
<td>L=1</td>
<td>H=3</td>
<td>L=1</td>
<td>L=1</td>
<td>L=1</td>
<td>L=1</td>
<td>M=2</td>
</tr>
<tr>
<td>Connection type</td>
<td>H=3</td>
<td>M=2</td>
<td>M=2</td>
<td>L=1</td>
<td>M=2</td>
<td>M=2</td>
<td>L=1</td>
</tr>
<tr>
<td>Access control</td>
<td>H=3</td>
<td>L=1</td>
<td>H=3</td>
<td>L=1</td>
<td>M=2</td>
<td>M=2</td>
<td>M=2</td>
</tr>
<tr>
<td>Dependency Direction</td>
<td>H=3</td>
<td>M=2</td>
<td>M=2</td>
<td>L=1</td>
<td>M=2</td>
<td>M=2</td>
<td>M=2</td>
</tr>
<tr>
<td>Total Coupling Value</td>
<td>19</td>
<td>16</td>
<td>19</td>
<td>10</td>
<td>14</td>
<td>17</td>
<td>13</td>
</tr>
</tbody>
</table>

5 DISCUSSION AND CONCLUSION

In this work we have attempted to establish the relationship between coupling and cohesions, and the complexity of architectural styles using a qualitative framework. This work is not complete by any means. It is just the beginning. Therefore, the future work will include, among other things, the role of coupling and cohesion, and complexity in designing the systems that should meet reliability, maintainability, safety, availability, reusability requirements at architectural level. Toward these goals, we are currently working to identify architectural styles that support these qualities. For example, to describe reliability that is one of the most critical aspects of mission critical systems [2], we are planning to establish a proper correlation...
between reliability and complexity, complexity and testability, and reliability and testability.

Design for reliability requires employing strategies to reduce complexity. Complexity has an important impact on the testability. Testability, in turn, affects reliability, because reliability is estimated using testing strategies (e.g., unit and integration testing) and design principles such as abstractions.

Future work also includes performing few empirical studies on systems in the domain critical systems (e.g., avionic and aerospace systems) to validate our proposed approach. The steps by which we are planning to carry out our experimentation include the following steps:

1. Identifying the coupling and cohesion factors;
2. Defining attributes of these factors;
3. Performing case studies from the domain of critical systems such as robotic system, and avionic and aerospace software systems to demonstrate the feasibility of our work.
4. Collecting and analyzing the data;
5. Revising these factors and their corresponding attributes.

6 ACKNOWLEDGEMENTS

This work has been partially funded with a grant from the Rockwell Collins Corporation. The authors would also like to acknowledge anonymous referees for their comments.

7 REFERENCES

Evaluating the Use of Project Glossaries in Automated Trace Retrieval

Xuchang Zou, Raffaella Settimi, and Jane Cleland-Huang
College of Computing and Digital Media, DePaul University, Chicago, IL, USA

Abstract - Automated traceability methods use information retrieval techniques to dynamically generate traceability links, however they suffer from precision problems. This paper extends our previous work in using a project glossary to improve trace results and presents criteria for evaluating whether an existing project glossary can be used to enhance results in a given project. A new approach for automatically extracting a set of important terms and phrases is also described. Our experimental results suggest that these terms and phrases can be used effectively in lieu of a project glossary to help improve precision of the retrieved traces.

Keywords: Requirement traceability, automated trace retrieval, information retrieval models.

1 Introduction

Requirements traceability supports a number of critical software engineering tasks such as requirements validation and verification, change management and software maintenance. It has been widely recognized as an essential task in an increasing number of process improvement standards and regulatory requirements [5]. However, traditional traceability methods require extensive human effort to implement effectively.

Several researchers have therefore developed dynamic traceability algorithms which use Information Retrieval (IR) techniques to dynamically generate traceability links on an “as-needed” basis [1,2,6,9]. Because software artifacts such as requirements, design documents and source code contain large amounts of textual information, their dependencies can often be discovered using IR techniques such as the Vector Space Model [6], Latent Semantic Indexing model [1,9], and Probabilistic Network model [2].

The effectiveness of such IR-based requirements tracing tools are typically assessed using two standard IR metrics: recall defined as the proportion of true links that are retrieved by the tool out of all the true links, and precision defined as the proportion of retrieved links which are true links [4]. In the context of requirements traceability, a tracing tool must obtain high recall, i.e. it must retrieve as many true links as possible, as it is easier for the analyst to examine the subset of retrieved links to filter out the unwanted ones, than to search through the entire document collection to find missing true links. Furthermore, we have found that practitioners are not willing to use an automated tracing tool unless they are confident that it can find all of the true links. Previous empirical studies in dynamic tracing retrieval indicate that at high recall levels of 90%, precision is usually below 40% and sometimes even less than 10% [1,2,6,9]. Although using an automated IR approach significantly reduces the effort required to manually perform tracing, the low precision of the results means that analysts still have to evaluate a large number of candidate links in order to find the true ones. The low precision of the trace retrieval results may affect the analyst’s trust in the accuracy of the tool and could limit the adoption of IR-based automated traceability methods in industry.

In an effort to improve the precision of the results, our previous work [16] proposed a trace retrieval approach that utilized the information in the glossary of a software project. Because terms and phrases defined in the project glossary tend to capture many of the critical concepts of a project, links between two artifacts in which project terms or phrases co-occur are assigned higher relevance scores by the retrieval algorithm. The preliminary results have suggested that this approach can improve precision especially among the top retrieved links. However, the effectiveness of the glossary approach relies on the availability of a meaningful glossary which has been consistently utilized to construct all traceable artifacts. This paper addresses the problem that occurs when either no project glossary is available, or when the glossary has not been used consistently.

2 Using the project glossary

Our previous work [2] described a Probabilistic Network (PN) model [14] for dynamically generating and retrieving links between requirements and software artifacts. Relevance scores are computed between two sets of artifacts, known as queries and documents. In this paper the queries are individual requirements. The probabilistic model assumes that a software artifact \(d\) containing a set of terms co-occurring in a requirement \(q\) is likely to be related to that requirement, and a trace should be established between \(d\) and \(q\). The probability score \(p(d|q)\), representing the likelihood of a link between \(q\) and \(d\), is defined in terms of
the frequency \( freq(d, t_i) \) of terms \( t_i \) co-occurring in both \( q \) and \( d \), and is calculated as
\[
p(d | q) = \sum p(d | t_i) p(t_i | q) / p(q)\]
This expression is based on the tf-idf [13] standard weighting strategy. The first component
\[
p(d | t_i) = freq(d, t_i) / \sum freq(d, t_i)
\]
the relative frequency of term \( t_i \) in document \( d \) and increases
with the number of occurrences of \( t_i \) in \( d \). The second component
\[
p(t_i | q) = freq(q, t_i) / n_i
\]
with \( n_i \) being the number of traceable documents containing term \( t_i \), represents
the inverse document frequency. This component assumes
smaller values for frequently used terms that appear in many
documents and therefore provide weak information about
potential links. The last component
\[
p(q) = \sum p(q, t)\]
in the query. Potential traces to a requirement \( q \) are
identified by retrieving all documents \( d_i \) whose probability
score \( p(d|q) \) falls above a certain threshold. A high value of
\( p(d|q) \) represents strong evidence that a potential link
between \( q \) and \( d_i \) exists.

2.1 Retrieval algorithm utilizing the glossary

Previous studies analyzing the application of the basic
PN algorithm to requirements tracing have shown that high
recall levels are typically achieved at the expense of
precision. For instance, recall levels of 80-90% correspond
to low precision values ranging from 40% to levels as low
as 10% [2]. To address the low precision problem, an
approach was previously proposed to enhance the basic PN
algorithm by considering the additional information
included in the glossary of software projects. Key terms and
phrases in the project glossary tend to capture the critical
concepts of a project. Hence two documents that contain the
same glossary items are likely to be related to the same
specific concept. The enhanced retrieval algorithm therefore
assigns heavier weights to glossary terms and phrases, and
computes a higher probability score for a link between
a requirement and a traceable document that share the same
glossary items. The glossary approach is typically applied
along with phrasing [16], an enhancement method that
computes the probability score between two documents
considering not only single terms but also phrases co-
occurring in both documents. The reasons for incorporating
the glossary approach into the phrasing method are as
follows:

1) Phrases are detected in the phrasing method by using
QTag, a freely available Part-of-Speech (POS) tagger [15]
that uses a dictionary to identify the syntactic category of
each token in the text. All phrases are assumed to be equally
meaningful and contribute equally to strengthening the
belief in a link. However, there are usually certain phrases
or terms in a project that are more significant for capturing
its core concepts. As an example, in the IBS dataset [12], a
system that describes the requirements and design of a
public works department system for managing the de-icing
of roads, four phrases were identified from one requirement
by QTag: summary report, weather report, weather
conditions and time period. Higher weights should be
assigned to phrases that are related to core concepts for IBS
such as weather report and weather conditions rather than
to summary report and time period. The two more
meaningful phrases appear in the glossary of IBS. Thus
using the glossary enables the IR based traceability tool to
identify more critical phrases and to weight them more
heavily.

2) The project glossary may also contain additional
phrases that the syntactic parser was unable to discover.
These phrases may for example not fit into the prescribed
grammatical template or may contain more than the
standard number of terms.

The probabilistic information retrieval using the project
glossary is defined as follows. The set \( S_{PG} = \{k_1, k_2, \ldots, k_m \} \)
contains keywords defined in the project glossary, and
\( S_{PG} = \{t_1, t_2, \ldots, t_j \} \) is the set of terms in phrases defined in
the glossary. The new probability \( p_{PG}(d|q) \) of a link between a
document \( d \) and a query \( q \) is computed considering single
terms and phrases co-occurring in both documents as well as
additional information in the project glossary. The link
probability score \( p_{PG}(d|q) \) is defined as follows:
\[
p_{PG}(d|q) = p_{PG}(d|q) + p_{PG}(d|q) +
\]
\[
\delta \sum_{k \in S_{PG}} p(d|k_i)p(q) + \delta \sum_{i \in S_{PG}} p_i(d|k_i)p_i(q) (1)
\]
where the component \( p(d|k) \) is the basic probability score
defined in section 2, the value \( p_{PG}(d|q) \) represents the
contribution of phrases to the link probability score and is
computed similarly to \( p(d|q) \) but only considering the
frequency of phrase terms co-occurring in both \( d \) and \( q \).
The remaining two components represent the use of project
glossary terms and phrases in the new IR based tracing
approach. The expression in (1) assigns heavier weights to
project glossary keywords and phrases co-occurring in both
\( d \) and \( q \), and their contribution to the overall probability
\( p_{PG}(d|q) \) depends on the value of a parameter \( \delta \geq 0 \) to be set
by the analyst. In our experiments, the parameter \( \delta \) is set
equal to 0.5.

A preliminary evaluation [16] of the results of the
glossary approach application to trace retrieval indicates that
a higher percentage of true links are found at the top of the
retrieved links list. The results described in section 2.2.2
show that for the relatively large IBS dataset a significant
increase of 17% (from 47% to 64%) in precision is achieved
at recall level of 10%, indicating that more true links were
among the top ranked retrieved links. With such
improvement it is likely that important links will be found
earlier in the analysis process and this might raise the
confidence of the analyst on the tracing tool. However, the
2.2 Evaluating the project glossary

Although some experiments show its effectiveness in improving precision for the top ranked retrieved links, the project glossary approach has some limitations. It assumes that a project glossary exists and furthermore, that it was used consistently throughout the design of software artifacts. In reality many software projects do not have a pre-constructed glossary available or have a “weak” glossary that has not been consistently used. It would be helpful to predict the effect of the project glossary usage in link retrieval prior to running the automated trace retrieval technique. Thus, the following research problems were investigated:

1. What characteristics must an existing project glossary have in order to be potentially useful for improving the retrieval results?
2. For projects in which a glossary is unavailable, can a set of key terms and phrases be extracted and used in lieu of the project glossary to help improve the retrieval results?

To answer the first question, an empirical study was conducted to analyze the retrieval results using existing project glossaries. The study identified a set of characteristics for the existing project glossaries that can predict whether the glossary approach is effective in improving the precision of tracing results.

2.2.1 Criteria for a useful project glossary

Three project glossary characteristics have been investigated and the following criteria have been established:

**Criterion # 1:** Project glossary items should be consistently used in the traced documents. A project glossary is usually provided as part of the requirements documentation to reflect the terminology used in a specific software project. It is strongly suggested to avoid using synonyms of the glossary items during the creation of the software artifacts as they can introduce ambiguity and could potentially lead to misinterpretation of those artifacts. However, existing project glossaries are frequently not consistently followed during the project development. As a result, using such project glossary information in the automated trace retrieval tool may have little or no impact on the retrieval results.

One approach to evaluate whether or not the glossary terms have been followed is to detect the occurrence of the synonyms of individual glossary terms. The presence of synonyms of a given term, even a single occurrence in the traced documents, hints that the term may not have been consistently used.

**Criterion # 2:** Glossary items should have high term specificity. Term specificity of individual glossary items is also a critical factor that can affect the effectiveness of the glossary approach. Low term specificity indicates that the term may be associated to a range of possible concepts and consequently it may yield low precision in search results as a number of irrelevant documents containing the term will be retrieved. Various measures for term specificity have been extensively investigated in IR [7]. One commonly used measure for term specificity is the inverse document frequency for term $t$ defined as $idf(t) = \ln(|D|/|D_t|)$, where $|D|$ is the total number of documents in the collection, and $|D_t|$ represents the number of documents containing $t$. In the context of this research, glossary terms specificity is computed within the collection of requirements, and represents the assumption that terms occurring in fewer requirements should be considered more specific.

**Criterion # 3:** Glossary items should be domain specific. The third factor investigated in our study was domain specificity of the glossary items. A term is considered domain-specific when it occurs more frequently in domain-specific documents than in general documents. Some terms, such as “software”, are very common in the general technical documents and may deteriorate the retrieval precision when used in the glossary approach. Therefore, glossary items should be domain-specific in order to contribute effectively to the glossary approach in retrieving more true links.

One simple measure of domain specificity $DS(t)$ for a term $t$, is estimated as the term relative frequency within domain-specific documents versus the relative frequency computed in a general technical corpus [8], and can be calculated as follows:

$$DS(t) = \ln \left( \frac{\sum_{D} freq(t,D)}{\sum_{G} freq(t,G)} \right)$$

(2)

where $freq(t,D)$ is the number of occurrences of term $t$ in the domain-specific document collection $D$, and $G$ represents the general corpus that contains documents from various domains. The overall domain specificity of a given project glossary can be evaluated as the average domain specificity of all items defined in the glossary. In our experiments, the domain-specific document collection $D$ used to compute $DS(t)$ in expression (2) for a specific project glossary consists of the requirements associated to that project glossary. The general corpus $G$ contains all requirement specifications in the available software projects excluding the software project used in $D$. We are currently expanding the general corpus in order to obtain a more accurate measure of domain specificity. (Notice that in our experiments, when a term is unique to the given project, i.e. $freq(t,G)=0$, the domain-specificity value for that term is set by default equal to the high value of $\ln(10000)$).

2.2.2 Criteria validation

The effectiveness of the criteria proposed in section 2.2.1 that evaluate the existing project glossaries was validated for the following two datasets. The IBS dataset
[12] is a relatively large dataset with a complete and thoroughly validated trace matrix. It describes the requirements and design of a public-works department system for managing roads de-icing. The system activities include prediction of icy road conditions, work order management, truck management, and inventory control. The tracing task in IBS is focused on identifying links between 164 requirements and 71 UML class. The SE450 dataset contains 15 anonymous student term projects for a MS level Software Engineering class at DePaul University. The students used Java to implement a traffic simulation system. The dataset consists of 46 requirements that specify road maps, road segments, vehicles and stoplights, and 15 sets of Java source code which vary greatly in structure. The tracing task for this dataset is from the same set of requirements to the 15 sets of Java source code. Each student created a trace matrix for his/her own project to trace java classes to individual requirements. All the matrices were manually validated by two researchers to ensure the correctness of the matrices.

The project glossary of IBS contains 6 keywords and 28 phrases. For SE450, the glossary contains only 10 entries consisting of 4 keywords and 6 phrases. The following analysis discusses in detail the application of the three criteria to the IBS dataset and to one of the fifteen SE450 projects.

Criterion 1 evaluates the presence of synonyms of glossary terms and was implemented using WordNet [1], a general purpose thesaurus which groups English words into sets of synonym and provides the semantic relationships among different synonym sets. No glossary synonyms were found in the traced UML classes of the IBS dataset. However, in the SE450 chosen project, synonyms of two glossary entries, “vehicle” and “obstacle”, were detected. The student did not follow the provided glossary terms but instead used “car” and “obstruction” throughout his/her code.

The term specificity and the domain specificity values for the two glossaries were also evaluated. The average term specificity for the SE450 project was 0.78, less than 50% of the score of 1.63 for IBS. A lower score for the SE450 project may have been caused by glossary entries, such as “road segment” and “vehicle” that were used in a large number of requirements and therefore had relatively low idf values. The average domain-specificity of the glossary items in SE450 was 5.03, about 53% lower than the score of 7.69 for the IBS glossary. The analysis reveals that the SE450 project glossary shows some weakness based on the proposed criteria. Synonyms of glossary terms were used in the traced documents of most fifteen projects, and for all projects both average term specificity and domain specificity scores were significantly lower than the corresponding values in IBS. This indicates that the project glossary approach applied to the SE450 fifteen projects is not expected to produce any significant improvement in the retrieval results precision when compared to the basic approach.

To validate the criteria above, the enhanced trace retrieval algorithm using the project glossary information was applied to the IBS and the SE450 datasets. The results of these experiments are summarized in Figure 1 which displays the precision of the trace retrieval results at various recall levels using the basic PN, phrasing and the glossary approach. Figure 1a indicates that for the IBS dataset the glossary approach retrieves a higher percentage of correct links among the top ranked retrieved links, which correspond to low recall levels. In fact the highest increase of 17% in precision occurs at 10% recall level. The results also show that the precision of the three automated retrieval tools did not change significantly for high levels of recall. This means that the project glossary approach manages to push more true links to the top, but there are still true links that are scored relatively low and can be identified only if a larger list of links is retrieved.

The application of the project glossary approach produced no significant improvement in the retrieval results for the SE450 project compared to the other two methods. Figure 1b shows that the precision at the top of the candidate links list decreased 16% when project glossary information was used in the phrasing algorithm. An analysis of the results revealed that several links between requirements and documents containing the glossary...
keyword “vehicle” were incorrectly assigned higher scores and pushed towards the top of the retrieved links list. The keyword “vehicle” is frequently used in the SE450 project documents and has low a specificity value as it is used to describe different concepts. These results suggest that the criteria described above provide a simple way to detect if a project glossary is meaningful and can be used to improve the automated trace retrieval.

2.3 Keyword and phrase extraction method

This section presents an automated technique for extracting a set of critical keywords and phrases which can be used in lieu of a project glossary to improve the precision of the retrieval algorithm when project glossaries are unavailable or “weak”.

Keywords extraction methods have been extensively investigated in IR. Several algorithms have been proposed to automatically extract key terms from document collections in a variety of applications from text mining to web page retrieval. For example, Matsuo et al [10] proposed a method to extract keywords using statistical information based on terms co-occurrence. This approach, like many other statistical keywords extraction algorithms, does not capture the syntactical information of the terms.

Our proposed approach is based on a syntactical method that identifies critical keywords and phrases from the requirements collection. The categories of glossary items considered in this study are single nouns and noun phrases since they are the most common types in a glossary. The extraction of glossary keywords and phrases consists of the following two steps.

Step 1: Generate candidate keywords and phrases. A Part-of-Speech (POS) tagger identifies single nouns and phrases. All single nouns and two-noun phrases are considered candidate glossary items. The experiments described in this paper consider only phrases consisting of two nouns.

Step 2: Filtering. The candidate glossary items generated from step 1 will inevitably include some unimportant terms. The following filters are applied to remove those terms:

Filter A: Term specificity. Keywords and phrase with a term specificity value (idf(t)) below a certain threshold are filtered out as only high specificity terms are likely to help improve precision in retrieval results. The threshold varies and depends on the characteristics of the given document set.

Filter B: Terms domain-specificity. Keywords and phrases with a domain specificity value (DS(t)) below a certain threshold are filtered out, as terms with high domain specificity are more likely to capture project core concepts.

Filter C: Nouns filtering. The last noun (head) in the phrase is assumed to convey significant syntactical information about the phrase. As an example, in the phrase “road map”, the word “road” is used to modify the head noun “map”. As “road map” is more specific than single noun “map”, it is necessary to remove “map” from the keywords set. In general if a noun-noun phrase is kept as a key phrase in the extracted set, the head noun of this phrase cannot be a single keyword in the extracted set.

3 Analysis of the extraction method

The proposed keywords and phrases extraction method was analyzed for the SE450 dataset and the CM1 dataset. CM1 is a large dataset extracted from a NASA project for a science instrument and is available at the Promise Data Repository [11]. The dataset consists of 235 high-level, and 220 low-level requirements, with a traceability matrix containing 361 true links between these requirements. The matrix was constructed by NASA and manually verified and modified by Hayes et al. [6].

The extraction of important keywords and phrases in the two datasets follows the procedure described in section 2.3. First candidate keywords and phrases were extracted from the requirements by using QTag. The filtering step (A) based on the terms specificity is defined so that only keywords occurring in no more than 3 requirements in projects with less than 60 requirements or 5% of the total requirements for larger requirements sets are kept in the...
candidate set.

In the domain-specificity based filtering step (B), all items that are unique to this project (i.e., freq(t,G)=0) were kept in the candidate glossary set. For the remaining candidate items, the top 50% items ranked in terms of domain-specificity were kept while the bottom 50% items were removed. It is worth noting that the thresholds for both term-specificity and domain-specificity were selected empirically. Methods to determine the optimal thresholds for different projects are still under investigation. The extraction method identified 5 keywords and 30 phrases in the SE450 project documents. Only two terms from the existing project glossary occur also in the extracted set, as they have relatively high term specificity and domain specificity. For CM1, the set contains 223 entries with 59 keywords and 164 phrases.

3.1 Trace retrieval with extracted set

The project glossary approach was then applied to the two datasets using the extracted key terms and phrases in lieu of the missing or weak project glossaries. The graph in Figure 2a displays changes in precision at 10% and 20% recall for the SE450 projects comparing results from the enhanced trace retrieval approach incorporating both phrasing and the extracted set and from the basic PN algorithm. A significant improvement in precision is observed for both recall levels, indicating a higher precision among the top retrieved links. The improvement was especially significant for project 14 for which the precision at 10% recall level increased by 43%. In this case, both the phrasing and the extraction method identified the key phrase “segment length” that is included in several true links. Those true links were consequently assigned higher scores by the enhanced retrieval algorithm and pushed to the top of the retrieved links list. Project 12 experienced a noticeable decrease in precision of about 4% at 10% recall level with respect to the basic algorithm. The decrease was in fact due to the presence of the phrase “road segment” occurring in several false links. Their probabilities were incorrectly enhanced by phrasing, not by the glossary approach as this phrase has relatively low term specificity and was not included in the extracted key terms and phrases set.

Similarly to IBS, the change in precision at high recall levels after the new algorithm was applied was also not noticeable in SE450. Neither phrasing nor the glossary method were able to enhance the probability scores for most of the missed true links as there were very few or even no shared terms and phrases among the query-document pairs.

To demonstrate the improvement of using the extracted key terms set over the existing SE450 glossary, Figure 2b displays the change of precision at 10% and 20% recall levels after using the extracted set in the glossary approach compared to using the provided standard glossary for this dataset. As displayed in Figure 2b, the precision changes are mostly positive, indicating that the extracted key terms and phrases were more useful in identifying correct links than an existing weak project glossary. A significant 30% increase in precision was observed in project 4 at 10% recall level.

Using the extracted key terms and phrases in the retrieval algorithm also increased the precision of the top ranked retrieved links for CM1. Figure 3 displays the scatter plot of the recall-precision values for three retrieval methods in this dataset. Compared to the basic PN algorithm, the glossary approach utilizing the extracted keywords and phrases increased the precision value from 22% to 31% at 10% recall level, with an additional increase of 4% with respect to the precision for the phrasing method only. In conclusion, these results show that the extracted key terms and phrases can be effectively used to improve precision in the top retrieved links.

4 Conclusions

An extension to our previous project glossary retrieval algorithm is proposed to overcome some limitations of that approach. This paper suggests simple techniques to evaluate the quality of information in a project glossary and its usefulness in improving trace retrieval results. Occurrence of synonyms in the traceable documents and frequency of project glossary terms may provide information on the meaningfulness of a project glossary to describe critical concepts.

A new procedure to extract critical keywords and phrases from requirements is proposed. Empirical results show that this set of keywords and phrases can be used in the project glossary trace retrieval algorithm when the glossary is unavailable or provides weak information about the project. The keywords and phrases extraction method introduced here is still at an early stage and will be evaluated against more extensive datasets in the future. An interesting question to be addressed in future work is whether extracted key terms and phrases provide additional relevant information about correct links even when a meaningful project glossary is available.
The new retrieval algorithm using either the project glossary information or the set of extracted key terms and phrases often achieves higher precision among the top ranked retrieved links. Although this improvement can increase the analyst’s trust on the tracing tool, some true links are still missed, and are hard to be retrieved using only textual content information. Alternative methods using additional sources of information have been proposed [2], but more work to improve precision and recall is needed.

Reference

Systematic Test Data Generation for Embedded Software

Justyna Zander-Nowicka\(^1\), Xuezheng Xiong\(^1\)
\(^1\)MOTION, Fraunhofer FOKUS
Kaiserin-Augusta-Allee 31
10589 Berlin, Germany
{justyna.zander-nowicka, xuezheng.xiong}@fokus.fraunhofer.de

Ina Schieferdecker\(^1, 2\)
\(^2\)Technische Universität Berlin, Faculty IV,
Straße des 17. Juni 135
10623 Berlin, Germany
ina.schieferdecker@fokus.fraunhofer.de

Abstract – Functional testing of software dedicated for hybrid embedded systems should start at the early development phase and requires analysis of discrete and continuous signals, where timing constraints play a significant role. Test data generation is done manually nowadays, though it should be automated to the highest possible extent. Hereby, a concept for testing at the model level is introduced. Firstly, the test evaluation design based on the requirements is specified. Then, an automatic and systematic derivation of test data structures appears and finally, generation of test data variants in combinations occurs. The application of signals’ features is proposed for both – test data generation and test evaluation. Furthermore, patterns of generators for such features are built, concrete test stimuli are produced and default sequencing algorithms are created.

Keywords: automatic test data generation, functional test, test evaluation, hybrid behavior

1 Introduction

Testing costs at least 50\% [3] of the effort while software development. To reduce this cost and guarantee software quality, testing should emerge earlier in the software development cycle. Numerous works have contributed to make testing more efficient and effortless (i.e., automatic). However, a systematic and automated test data generation is still a research problem as its progress depends on a lot of factors (e.g., the testing method applied, interfaces and type of the system under test (SUT) or evaluation algorithm). In this paper our goal is to support dynamic black-box testing of hybrid embedded software. We support testing on the Model-in-the-Loop Level [5], when neither code, nor hardware exists yet so as to reduce the resources and cost of the entire software development.

On the basis of our preliminary work [23] regarding the test specification and test evaluation design, this paper introduces a test data generation approach. The first concepts on the generation of test data sets and structures have been already introduced in [24]. The test data patterns can be retrieved from the test evaluation design automatically. Hence, we focus on the test data variants generation, their management and combination of the retrieved signals within a test case. We investigate different types of signals’ features, their dependencies and influences on each other in terms of the test data retrieval. Finally, synchronization of the obtained test data is considered and test control is discussed.

The paper is structured as follows. In Subsection 1.1 a short characteristic of hybrid embedded systems is given. Subsection 1.2 introduces our previous work being the basis for this research. Section 2 provides the details on the structure of the generated test data sets. Also, an overview on the transformation targeting this structure is depicted. Further on, in Section 3 the concrete variants of the test data for every single test case are given. In Section 4 a case study from the automotive domain is provided. In Section 5 related work is discussed. Section 6 evaluates the proposed test approach. Conclusions complete the paper.

1.1 Hybrid embedded systems

Embedded control systems are usually reactive with continuous-time dynamics, discrete events, and discrete mode changes. They must meet specific timing constraints and they use continuous signals for monitoring and controlling their environment via sensors and actuators and discrete signals for communication and coordination between system components [19]. There are tens of electronic control units in an embedded system (e.g., a vehicle) and software has a major influence on their performance and safety. To handle the growing complexity of this software, model-based development is nowadays applied. It allows for models specification, code generation and subsequently for their primary execution [5], [6]. Such models and their requirements serve as a basis for testing, which is a central and significant task within software production [18].

1.2 Proposed test methodology

The support of functional, dynamic testing requires an analysis of the SUT specification, test data selection, test evaluation algorithms and an execution or simulation environment. Figure 1 presents a simplified view on the test harness.
The SUT requirements specify its functionality and indicate the test objectives. Together with the interfaces of the executable SUT model they drive both – test data generation and test evaluation.

Our objective is to handle both discrete and continuous signals and the timing constraints between them. Similarly to [9] we aim to describe the test of an SUT by signal’s features. Under feature we understand an identifiable, descriptive property of a signal. Giving examples – increase, step response characteristics, step, maximum etc. are considerable signal’s features. We compare the features extracted from a signal with the expected values given in the requirements. We distinguish different feature types considering the feature availability on the one hand and the identification delay on the other hand [13]. However, their classification is out of the scope of this paper.

The test evaluation process happens by applying the so called validation functions (VFs). A VF is created out of a single requirement following a generic conditional rule:

\[ \text{IF preconditions set THEN assertions set} \]

**Preconditions set** consists of a number of SUT signals’ feature extractors related to each other by temporal, logical or quantitative dependencies (e.g., \( \text{FeatureA AND after(time1)FeatureB OR NOT(FeatureC)} \)); a comparator for every single feature or dependency; and a unit for preconditions synchronization (PS).

**Assertions set** looks similar, however includes a unit for preconditions and assertions synchronization (PAS), respectively. The details concerning synchronization problems are described in [13].

We define patterns of VFs that are able to continuously update the verdicts for a test case already during SUT execution. They are defined to be independent of the currently applied test data and can set the verdict for all possible test data vectors and activate themselves (i.e., their assertions) only if the predefined conditions are fulfilled. Hence, a pattern\(^1\) for a validation function (shown in Figure 2) consists of a preconditions block which activates the assertions block, where the comparison of actual and expected signal values happens. The activation and by that the actual evaluation proceeds only if the preconditions are fulfilled.

The easiest assertions block is built following the schema shown in Figure 3.

\[ \text{IF preconditions set THEN generators set} \]

\(^1\) The patterns are designed in MATLAB/Simulink/Stateflow® [14]

---

2 Test Data Structure Preparation

2.1 Systematic test data structure retrieval

In Figure 4 a pattern for the test model providing test data sets is presented. As discussed in [24] for each data set determined by a single preconditions set, a block is provided, where features generation happens. The test data can be generated following a conditional rule in the form:

\[ \text{IF preconditions set THEN generators set} \]

Additionally, SUT output signals may be checked, if necessary. The features generation is activated by a Stateflow diagram (Std) sequencing the features in time according to the default temporal constraints (e.g., \( \text{after(time1)} \)). A switch is needed for each SUT input to handle the dependencies between generated signals. Initialization & Stabilization block enables to reset the obtained signal so that there are no influences of one test case on another.

---

2.2 Transformation rules

The transformation from VFs to the generators of test signal’s feature is implemented following a set of rules. These are summarized in Table 1 for illustration purpose. The general principle is that if a given fact in the source is detected, then the action to generate the target is performed.
### 3.1 Test data selection

The test data are understood as the stimuli for the SUT. These are the test signals which stimulate the SUT to invoke a given behavioral scenario. In our approach a particular set of feature generators producing selected signals corresponds to the stimuli used within a particular test case.

The concrete test data variants are provided based on the generation patterns obtained from the transformations discussed in the previous section. The assumption for applying the variants generation method is the definition of the signal ranges and partition points on all the stimuli signals according to the requirements or engineer’s experience. A number of algorithms are proposed for signal variants generation depending on the signal’s feature type. Together with this analysis, equivalence partitioning and boundaries are used in different combinations to produce the concrete test data variants.

As a matter of example, feature – **increase** is considered in the following. A number of generation options given in Table 2 are possible to produce the signal variants systematically. We consider three possibilities showing two variants for each of them. For simplicity we take the ramp as a shape-representative of an increase. In the first option, **timing constraint** and **signal range** are the factors indicating the generation rule. Hence, we obtain two different ramps, both covering the entire range and preserving the proper time constraint ($t_1$, $t_2$ respectively). In the second option, **signal range** and the **tangent of the angle** play a significant role. Thus, no matter how long, but the signal variants must hold within the given tan(angle) along the entire value range. Finally, in the third option, **timing constraint** and **tangent of the angle** determine the generation rule. At this point only one default boundary of the signal range is considered, the tan(angle) is preserved and the predefined $t_1$ indicates the duration of a signal generation.

#### Table 2: Options for Generation of Increase

<table>
<thead>
<tr>
<th>No.</th>
<th>Criteria</th>
<th>Variant 1</th>
<th>Variant 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>time</td>
<td>signal (t)</td>
<td>signal (t)</td>
</tr>
<tr>
<td>2</td>
<td>signal range</td>
<td>max</td>
<td>max</td>
</tr>
<tr>
<td></td>
<td>tangent (angle)</td>
<td>$t_1$</td>
<td>$t_2$</td>
</tr>
<tr>
<td>3</td>
<td>signal range</td>
<td>min</td>
<td>min</td>
</tr>
<tr>
<td></td>
<td>tangent (angle)</td>
<td>$t_1$</td>
<td>$t_2$</td>
</tr>
<tr>
<td>4</td>
<td>signal range</td>
<td>min</td>
<td>min</td>
</tr>
<tr>
<td></td>
<td>tangent (angle)</td>
<td>$t_1$</td>
<td>$t_2$</td>
</tr>
</tbody>
</table>

Let us consider the **generate increase** in terms of a real world signal, i.e., vehicle velocity. Its value range is between $< -10, 70 >$. Additionally, $\{0\}$ is taken into account as the car changes its driving direction from backwards to forwards at this point. We choose the third generation option from Table 2. Our algorithm computes 10% of the current range around all boundaries and partition points. Hence, the following increases are obtained as representatives: $< -10, -9 >$, $< -1, 0 >$, $< 0, 7 >$ and $< 63, 70 >$. The duration of those increases can be either derived from the VFs, or set on default values, or changed manually. The
steps of computing the representatives on the value axis are illustrated in Figure 5.

Figure 5: Steps of Computing the Representatives

Firstly, the boundaries of the range or the physical limits in the environment are identified (step A). Then, all risk-based partition points are included (step B). They result from the specification or the test engineer’s experience. The increase ranges are calculated and the variants are generated (step C). Finally, durations of the features are added on the time axis.

### 3.2 Variants sequencing

When a test involves multiple signals, the combination of different signals and their variants should be established. Several combination strategies are known to construct the test cases – *minimal combination, one factor at a time*, and *n-wise combination* [10]. Combination strategies are the selection methods where test cases are identified by combining different values of test data parameters according to some predefined criteria. We implemented the first two strategies.

If the test data variants are available and the combination strategy has been applied, the test cases can be established. Every signal’s feature generators set constitutes a test purpose for a test case. All the sets together form a test suite and are sequenced in the test control. Before we go into the details of the test control, the sequencing of the test data within the test data sets will be explained. Hence, every single generated signal is activated for a given predefined period of time. The signal’s features within one set should be synchronized so that the timing is the same for all of them. Then, the Stateflow diagram *(Std)* on the test data level, called ‘Sequencing of test data in time due to Preconditions’ controls the activation of a given variants combination applying a predefined time too. This activation must be synchronized with the timing given in the test control algorithm.

Thus, in *Std* a time-related parameter needs to be added. It results from the test control specification. It enables to synchronize the starting point of a selected test data set forming a test case on the test data level with the starting point of a test case within the test control.

The first test case in the test control starts without any delay, so the parameter should be equal to 0. The duration of this first test case specified on the test control level determines the starting point of the next test case. Hence, the starting of the following test case appears after the former finishes, which means after a specified period of time. The same applies to the activation of the test data sets on the test data level. It starts after the same period of time as the test case specified on the test control level. Further on, the next following test case starts after all the previous finish. Thus, the same applies to the activation of the further test data set. Since we do not include the duration of all the previous test cases on the test data level (i.e., within the *Std*), we need to calculate this parameter (called *pre_time*) following the formula:

\[
pre\_time = \sum \text{durations of all previous test cases}
\] (3)

Let us consider an example for this algorithm. Assume that a test control depicted on the left hand side of Figure 6 is given the parameters are calculated as follows.

![Figure 6: Test Control and its Implication on the Test Data Sequencing](image)

This algorithm applies only to the first iteration within the test control. If more iterations are needed (i.e., if more variant sets are applied for a particular test execution loop over the test control), a more complex algorithm should be used depending on the current number of the iteration.

The test cases are executed one after another according to the sequence specified in the test control. After execution of one set of variants for such a sequence the next set of variants is chosen and the sequence of new test stimuli repeats.

### 4 Case study

This section demonstrates the application of the presented concepts for a selected case study. A simplified component – Pedal Interpretation of an Adaptive Cruise Controller developed by Daimler AG is used. This subsystem can be employed as pre-processing component for various vehicle control systems. It interprets the current, normalized positions of acceleration and brake pedal (phi_Acc, phi_Brake) by using the actual vehicle speed (v_act) as desired torques for driving and brake (T_des_Drive, T_des_Brake) [5]. Furthermore, two flags (AccPedal, BrakePedal) are calculated, which indicate whether the pedals are pressed or not. Some excerpts of these functional requirements are given in Table 3, whereas the SUT interfaces are presented in Table 4.

We selected the Matlab/Simulink/Stateflow [14] environment to show the feasibility of our solution. It provides a simulation engine, which allows for the execution of tests, thus their dynamic analysis. It supports
hybrid systems development and allows for using the same development language for both – system design and test design – so as to integrate testing and system modeling [16].

Table 3: Requirements for Pedal Interpretation [5] (excerpt)

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Interpretation of pedal positions</td>
</tr>
<tr>
<td></td>
<td>Normalized pedal positions for the accelerator and brake pedal should be interpreted as desired torques. This should take both comfort and consumption aspects into account.</td>
</tr>
<tr>
<td>1.1</td>
<td>Interpretation of brake pedal position</td>
</tr>
<tr>
<td></td>
<td>Normalized brake pedal position should be interpreted as desired brake torque ( T_{\text{des Brake}} ) [Nm]. The desired brake torque is determined when actual pedal position is set to maximal brake torque ( T_{\text{max Brake}} ).</td>
</tr>
<tr>
<td>1.2</td>
<td>Interpretation of accelerator pedal position</td>
</tr>
<tr>
<td></td>
<td>Normalized accelerator pedal position should be interpreted as desired driving torque ( T_{\text{des Drive}} ) [Nm]. The desired driving torque is scaled in the non-negative range in such a way that the higher the velocity is given, the lower driving torque is obtained.</td>
</tr>
</tbody>
</table>

Table 4: SUT Inputs of Pedal Interpretation Component

<table>
<thead>
<tr>
<th>Velocity (v)</th>
<th>Acceleration pedal (phi_Acc)</th>
<th>Brake pedal (phi_Brake)</th>
<th>SUT Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;-10, 70&gt;</td>
<td>&lt;0, 100&gt;</td>
<td>&lt;0, 100&gt;</td>
<td>Permitted values range</td>
</tr>
</tbody>
</table>

Let us analyze requirement 1.2 for illustration purpose. It is interpreted as the following conditional rules:

- IF \( v \) is constant AND \( \phi_{\text{Acc}} \) increases AND \( T_{\text{des Drive}} \) is non-negative THEN \( T_{\text{des Drive}} \) increases.
- IF \( v \) increases AND \( \phi_{\text{Acc}} \) is constant AND \( T_{\text{des Drive}} \) is non-negative THEN \( T_{\text{des Drive}} \) does not increase.
- IF \( v \) is constant AND \( \phi_{\text{Acc}} \) decreases AND \( T_{\text{des Drive}} \) is non-negative THEN \( T_{\text{des Drive}} \) decreases.
- IF \( v \) is constant AND \( \phi_{\text{Acc}} \) decreases AND \( T_{\text{des Drive}} \) is negative THEN \( T_{\text{des Drive}} \) decreases.
- IF \( v \) is constant AND \( \phi_{\text{Acc}} \) increases AND \( T_{\text{des Drive}} \) is negative THEN \( T_{\text{des Drive}} \) increases.
- IF \( v \) is constant AND \( \phi_{\text{Acc}} \) is constant THEN \( T_{\text{des Drive}} \) is constant.

The VFs for the formalized IF-THEN rules are designed as shown in Figure 7. Apart from the actual feature check, we deal here with preconditions that must be assured so as to activate the proper assertion. A similar cause-effect analysis resulting in scenario patterns is discussed in [20]. Note that only simplified excerpts of VF blocks are presented in the following. For further details please refer to [23].

Let us focus on the test data patterns retrieval from the test evaluation design for this concrete example. Using the preconditions from Figure 7 and the pattern for test data generation from Figure 4, we obtain the design given in Figure 8 automatically as a result of transformations. Then, we apply the test data generator (TDG) to derive the representative test stimuli.

The number of preconditions blocks in Figure 8 suits the number of VFs appearing in Figure 7. Sequencing of the features generation is performed in the Stateflow diagram. Signal switches are used for connecting different features with each other according to their dependencies as well as for completing the rest of unconstrained SUT inputs with user-defined, deterministic data, when necessary (e.g., \( \phi_{\text{Brake}} \)).

Considering the first VF and the first test data set from Figures 7 and 8 respectively, the following applies. If the velocity is constant and an increase in the acceleration pedal position is detected then the assertion is activated. Thus, as shown in Figure 9 (middle part) – a constant signal for velocity is generated; its value is constrained by the velocity limits <-10, 70>. The partition point is 0 as discussed.

---

\[ A \] A direct interpretation of pedal position as motor torque would cause the undesired jump of engine torque while changing the gear maintaining the same pedal position.
in Subsection 3.1. The TDG produces five variants out of this specification. These belong to the set: \{-10, 5, 0, 35, 70\}.

Furthermore, it is checked whether the driving torque is non negative. It is the condition allowing the generation of the proper stimuli in the test execution. For the acceleration pedal position limited by the range \(<0, 100\> \text{ an increase feature is utilized. The entire situation is depicted in Figure 9 (bottom part). The insights of } \text{Generate increase } \text{subsystem are shown to illustrate the variants generation. Here, two variants of the test data are produced. These are the } \text{increases in the ranges } \(<0,10> \text{ and } <90,100>. \text{ They last 20 seconds each (here, default timing is used). Moreover, there is a connection of variants activation on the test data level with the test control level. It happens due to the application of From block deriving the variant number from Goto block specified on the test control level (see Figure 6).}

The brake pedal position is arbitrary set since it is not constrained by the preconditions.

Evaluating this particular test case the following applies: if the driving torque increases as expected, a pass verdict is delivered, otherwise a fail verdict appears.

We implemented the test concepts as a Simulink library [14]. It contains VFs (i.e., feature extractors, comparators), signal’s feature generators, data generators, patterns and other elements needed to generate Simulink/Stateflow test models. Additionally, transformation functions are available. They are responsible for enriching the SUT with test evaluation patterns, for detecting particular signal’s feature extractors present in preconditions of VFs and for retrieving the test data sets based on this detection. TDG produces the concrete test data variants.

The execution of the test using the test data resulting from the application of TDG in the case study let us discover that several issues have not been designed in the VFs. While testing the SUT, some ‘fails’ appeared. The reason for them was that the VFs were designed not precisely enough in respect to the constrained ranges of data. Thus, we refined our test system (i.e., VFs) and we generated the test data once again. The SUT issued a pass verdict at that point.

5 Related work

In the following, some of the available methodologies for testing the dynamic functional behavior of embedded systems are reviewed. The evaluation is based on the test design availability, test data selection algorithms and the automation grade of the method.

Model-checking approaches used for checking temporal safety critical requirements are available in Safety Checker Blockset [17] and EmbeddedValidator [1]. Together with the validation support given in [15] and watchdogs in [6] they contribute to the concept of signals’ features in our technique. The fact is especially valuable in the context of the test oracle design and test evaluation. We aim to cover complex properties and dependencies between continuous signals, although we aim to structure and systematize our solution. Reactis Validator [4] provides a test framework to graphically express assertions that check an SUT for potential errors, and user-defined targets that monitor system behavior in order to detect the presence of certain desirable test cases. If a failure occurs, a test execution sequence is delivered and it leads to the place where it happens. However, no predefined patterns enabling a scalable and systematic test design are available.

With the help of Classification Tree Editor for Embedded Systems (CTE/ES) [5], it is possible to set up value ranges for the individual ports and to combine these ranges in sequences. To represent test scenarios in an abstract way, they are decomposed into individual test steps [7]. Each test step defines the input situation at a certain time. A sequence of such test steps is called test sequence.

The Time Partition Testing (TPT) [11], [12] supports the manual selection of test data on the semantic basis of so-called testlets and the syntactic techniques Direct Definition, Time Partitioning and Data Partitioning which are used to build testlets. Testlets facilitate an exact description of test data. TPT is platform-independent and can be used at several embedded software development stages, which we do not directly support with our solution. The verdict and arbitration concepts known from such standards like UML 2.0 Testing Profile (U2TP) [16] or Testing and Test Control Notation – version 3 (TTCN-3) [8] are used in our method too.

Our approach differs from the related work as we start with designing the test evaluation model. Further on, we retrieve the test data sets following the structure of the test evaluation. The TDG concretizes the data. Its functionality has some similarities to the method applied in CTE/ES [5]. We use the SUT input partitions and boundaries to find the meaningful representatives. Additionally, we consider the SUT outputs too. The related work on model-based testing is provided in the surveys by [2], [21].

6 Evaluation

Evaluating the discussed test approach, we propose to use some black-box test criteria related to our method. With this practice, different types of test coverage based on the
functional relevance are supported. Hence, we define and apply such test quality metrics like e.g.: *SUT input/output range coverage*, *test data variants coverage*, *input range consistency coverage*, *variants related-precondition activation coverage*, *variants related-assertion activation coverage*. Due to the lack of space we consider them to be beyond the scope of this paper and we show only a few as a matter of examples.

The assumption is that the generated test data variants should provide representatives to cover all the input/output partitions and boundaries. These should be selected on the base of the predefined criteria for every single type of signal’s feature. The simplest metric – *SUT input/output coverage* achieves 100% if all possible variants of a given signal, based on the values analysis, are generated and applied during the test execution.

The *range* of a given signal must be consistent to the constraints given in the VFs’ preconditions (i.e., such that the constrained values do not cross beyond the allowed range). *Input range consistency coverage* checks this consistency in our approach.

Evaluating our test method further on, it provides a set of patterns which support the systematic test design and enable to reuse a number of structures for modeling the test. Additionally, some of the steps are automatic, which reduces the time of the test development.

7 Summary

In this paper a method for a systematic test data variants generation based on our preliminary work has been introduced. A special attention was drawn to: the automatic generation of the test data sets using transformation functions, production of the test data variants and their sequencing. The test control in respect to the obtained test data has been also mentioned. Finally, a case study was provided. Since, its test system was rather easy to implement we achieved 100% of *input and output coverage*. However, we expect that additional manual refinements are needed, especially when integration level test is considered. This part of our research is still under investigation with some results described in [22].

The scalability of the proposed test design is supported by a library/patterns extension mechanism. Systems may be analyzed either using the predefined elements or by creating new ones using the given patterns. We support only positive tests by now, thus the reciprocals of the generated data would be suitable candidates for negative testing. However, the VFs would have to be specified for them too.

The transformation of requirements into conditional rules may be complex for some cases, especially when timing constraints are included. Thus, we still want to explore the specification driven tests and their relation to VFs modeling and test data generation deeper, based on more case studies.

8 References

Towards the automatic generation of real time operating systems applying UML/MDA

Yessine Hadj kacem 1, Adel Mahfoudhi 1,2, Hedi Tmar 1 and Mohamed Abid 1

1 National Engineering School of Sfax Road Soukra km 3,5
Computer & Embedded Systems Laboratory (CES)
B.P. : w -- 3038 Sfax TUNISIA
2 Department of Computer Science, Science Faculty of Sfax
Road Soukra km 3,5 BP : 1171 -- 3000 Sfax TUNISIA

Abstract - This paper presents our contributions to the specification and the design of Real time embedded systems. It is not enough to reach runtime guarantees, performance and timing constraints, but it is desirable to employ Real time operating system RTOS. With the model driven approach MDA, and especially, with a UML profile, software designers can focus on their business logic. In other words MDA enables them to specify the functions and the properties of RTOS with a platform independent model.

This work is one step of the RTOS modeling, based on MDA. The MDE-based solution proposes to model the structure of an RTOS. It suggests the implementation of statecharts relating to the state of a process. Using this approach, real time constraints can be described by defining the semantics variants of statecharts. The main goal of the suggested proposition is to generate the code automatically.

Keywords: RTOS modeling, statecharts semantics, statecharts implementation, UML/MDA.

1 Introduction

The correctness of the computed results in real time embedded systems depends not only on the right results but also on the time during which they are provided. Their bad function may have serious effects (economic, legal, human, etc) because of the overload or the deadline expiry of some services. These systems require runtime guarantees from their underlying environment. To reach these guarantees, performance and timing constraints, these systems must be provided by software called RTOS.

So, several constraints, namely, real time ones such as deadline and release time, are imposed during their design phase. The checking of the system properties at a preliminary stage could reduce the problem impact. In fact, the real time design passes through different abstraction layers in order to automate the transition between them. Regarding the bottom layers, there exists many synthetic tools; the only problem concerns the CAD (Computer Aiding Design) of the highest level, which is the concern of the present work.

Currently, the object oriented modeling supported by UML (Unified Modeling Language) standard brings effective solution to the problems related to the real time systems design. Its realization is possible through the extension and/or the restriction of this standard via UML profile. However, the capacities of the real time behaviour specification of a given application have not been completely satisfactory yet. Indeed, these profiles provide solutions in terms of concurrent applications, but they remain insufficient especially, for the expression of the non-functional properties and the integration of the RTOS modeling.

This paper proposes an approach that supports the RTOS modeling starting from high level design, and ending up with the implementation code which can be used in different platforms.

The rest of this paper is organized as follow. Section 2 provides a brief discussion of some related work. The proposed approach is presented in section 3, in which the models of the RTOS structure and the scheduler are introduced through the implementation of statecharts. Section 4 presents a case study. Eventually, the paper closes with some final conclusions and an outlook on future work.

2 Related Work

When the specificity of each UML profile such as SPT [12], QoS/FT [12] and MARTES [11] is examined, it is concluded that the focus is on to the description of the material architecture and the application. These profiles are founded on an abstraction level higher than other approaches like ROOM, SDL, ADL, Petri Net. They also aim at the applications to data flow predominance rather than those to control. Even though these works briefly tackle the temporal aspects, they cannot cover the RTOS modeling. They are criticized for the lack of temporal and transitional semantics common to the models, as well as for the absence of tools which support them. In fact, these works have not enabled us to guarantee the reliability of the system yet, i.e, its determinism aspect. These models do not support the integration of real time characteristics sufficiently, and therefore, they do not consider the RTOS related to a specific architecture and application. The
simulation approaches need a simulation time long enough to give a relatively reliable sight of operation.

In [13], the authors’ work is based on two independent class diagrams: a diagram describing the structure and another describing the scheduler. These models related to the structure and the scheduler are explicitly separated. They suffer from major limitations; namely the coherence between the used diagrams and the lack of temporal semantics definition. In fact, the diagram used to characterise the scheduler is a static one. Thus, it can not cover the temporal behaviour of the RTOS. It must also be complementary to the structure model via a good expression of the follow-up of the real time process evolution. Therefore, a methodology assuring the coherence between the used diagrams and the support of scheduling model is important.

Based on a real time library VxWorks written in C, DAV et al. [4] carry out the transformations necessary to lead to a UML diagram. This downward transformation leads to some entities specifying the components of a real time system. The bond between them is left the designer’s responsibility. This approach is restricted with a static description. Thus, the behaviour can be dealt with introducing attributes describing the task progression, by defining a reflexive precedence relation or by adding an attribute showing time evolution. This technique is called the definition of operational semantics [2].

According to [10], a scheduling algorithm can be modeled using the sequence diagram and some stereotypes provided by the SPT profile. This proposal handicaps resides at the existence of a great number of scheduling algorithms. Consequently, the designer will face several scheduling algorithms using a succession of sequence diagrams and he will encounter the problem of integrating all these diagrams in MDA (Model driven architecture) process.

For the suggested models, the structure of the RTOS is described through a class diagram which includes the definition of operational semantics. Then, the behaviour of a task which constitutes the core of the RTOS is defined, in order to ensure coherence between various UML diagrams. In order to reach the model of scheduling, the temporal and transitional semantic of the statecharts relative to the various states of a real time process is defined.

3 The Proposed Approach

3.1 Overview

To overcome the limitations of the previously mentioned work, the proposed approach presents a step ensuring coherence between the various UML diagrams used and covering the behavioural aspect of the system like real time constraints (see Figure 1). First, the model of the RTOS structure is defined. Then, a statechart diagram related to the entity Task presents the temporal behaviour of a real time task. This diagram is annotated with OCL constraints. After that, the temporal semantic which is presented by the statecharts [1] is defined. While defining the variant semantic points of the statecharts, some techniques such as the reification and the enumeration of the states and the events are applied. The integration of design patterns is chosen for the re-use of existing and testing software components, rather than to recreate new models for the implementation of the statecharts. The last model corresponds to the target model during the stage of model transformation. As a final stage, the code is generated automatically.

Figure 1: The Proposed Approach

3.2 RTOS Structure

Two diagrams are proposed for the description of the RTOS structure; a class diagram describing the major components of the RTOS, and a statecharts diagram modeling the behavioural aspect of a real time Task. To guarantee the correction quality of the system, the statecharts diagram is annotated by some OCL rules.

The class diagram which is presented by Figure 2 is considered as the source model during the stage of model transformation that has an important role in Model Driven Engineering. It is represented by the following entities:

- Task: It is the most important component of the RTOS core. A task must acquire a great number of information in order to manage their scheduling
- Event: It causes the change of a task state
- ISR: Interrupt Server Routine: It is the routine in charge of the interruption processing. In this context, it makes the relay between the material interruption mechanism and the software one.
- Alarm: Based on a meter, an alarm could activate a task, impose an event or activate an alarmCallBack.
- Counter: It presents a software/hardware source for an alarm. It is an object intended for the recording of "ticks" coming from a timer.
- Resource: This entity is used to coordinate the concurrent accesses to shared resources. It is similar to semaphores.
- MeanOfCommunication: It is an abstract interface which manages data between active objects. The class ProtectedVar, which implements this interface, associates a mechanism of data protection (semaphore). In addition, LettreBox uses a file of messages. The read and write methods of this interface can update or get the value of the protectedVar class. This entity ensures data protection.
- Watchdog: The ISR contains one or more watchdog timers. The watchdog could possibly provide debugging information.
- Precedes: It illustrates the dependence of a task on another one.

For the dynamic model of the RTOS structure, it is described by the statecharts diagram. Before presenting the appropriate diagram, it should be taken into account that each state of a task running on RTOS can take only one of the following values:
- Waiting: waiting for synchronization;
- Running: running on the processor;
- Ready: waiting to be selected by the RTOS to enter the Running state;
- Suspended: task finished or stopped by the scheduler;
- Created: new task.

Figure 2: Static Model of the RTOS Structure
The structure of the statechart diagrams is, nevertheless, given a precise specification [14], which is required for tool interoperability. It can not easily be understood. So, UML 2.0 Statecharts present some semantic variation points. The definition of this semantics will be given in details in the next section. It corresponds to the target model during the model transformation.

```
procedure step()
begin
  eventSet := eventPool.select();
  @noEvent := eventSet.choice();
  transitionSet := getFirableTransition(event).select();
  @transition := transitionSet.choice();
  @transition.fire();
end.
```

**Figure 3:** The run-to-completion procedure

### 3.3 Scheduling model

The statecharts have been adapted to an informally or undefined semantics. The semantic variation points principally concern three aspects: time management (synchronous vs. asynchronous), the event selection policy, and the transition selection policy.

Harel [8] represents the semantics of the statecharts based on the description of a run-to-completion step as illustrated in Figure 3.

A set of approaches [7, 9] was proposed in the literature in order to define this semantics and implement the statecharts. For this work, the approach proposed by [5] is chosen. This technique is based on the enumeration and the reification.

The state of the Task entity can take the following values : { created, new, waiting, ready, running, stopped}. As for an event, it has these values : {terminate, activate, start, wait, preempt, release, create}.

The reification consists in the transformation of states into specific class hierarchy through the application of the design patterns.

A solution to separate the behaviour related to a state in an object, is to reify states through the utilisation of the state pattern [6].

To reify and select the right transition events, the command pattern [6] is applied to the entity Task. (See Figure 4)

In the light of the previously-given solutions and in order to ensure the progression of the automat, it is necessary to focus on the deterministic aspect of the system. It is also essential to determine the state running of the automat and the behaviour to be adopted according to the event which has occurred.

**Figure 4:** Application of the state and command pattern to the Task entity
Regarding the enumeration of the states and the events, the code reacting the progression of the automat is localized in the method processEvent(). As for the enumeration of the states and the reification of the events, the code will be set out again between the method processEvent() and execute() of each class. Regarding the reification of the states and the enumeration of the events, the code will be distributed between the method processEvent() and the method processEventPlay() of each class state. Finally, when the states and the events are reified, the code is distributed between the method processEvent() principal class, the methods processEvent() of the state class and the methods execute() of the class called event.

The last solutions based on enumeration and reification do not allow the representation of the concept of file messages related to the automat progression. Time is not taken into account. To overcome this problem, the use of the pattern Active-Object is, therefore, essential. This alone is thus effective for the achievement of the various policies of parallelism as it is shown in Figure 5.

Following the application of the reification of the states and the events, as well as the illustration of the evolution of the automat, the final model represented in Figure 6 will be considered as the target model during the transformation process.

**Figure 5:** Active-Object applied to Task entity

**Figure 6:** RTOS scheduler model
3.4 Code generation

The objective of this step is to transform an XML source model obtained automatically from an UML source model to an XML target model. The transformation model is based on ATL (ATLAS Transformation Language) language. To describe the model transformation, the KM3 (Kernel MetaMetaModel) language is used. It makes it possible to define models according to meta-model MOF (Meta Object Facility) in a textual form.

The source model transformed corresponds to the diagram of class presented by Figure 2. The code corresponding to XMI (XML Metadata Interchange) based on XML offers a tree structure to the present model by presenting the classes and the attributes in textual form.

The target model is the model that is aimed to be obtained after applying the transformations to the source model. It is presented in Figure 6.

4 Case study

It should be noted that the example used at the time of the transformation is taken adequately since the objective of this work is to show right the feasibility of the use of the MDE (model driven engineering) for the integration of RTOS modeling for embedded system design. So, transformations are focused just on ensuring tasks scheduling.

```java
rule TaskToTask
from
  s : RTOSModelTask
to
  w : RTOSCheduleTask { 
    idTask  <- s.idTask, 
    taskState  <- s.taskState, 
    priority  <- s.priority, 
    dateFirstActivation  <- s.dateFirstActivation, 
    deadline  <- s.deadline, 
    duration  <- s.duration, 
    period  <- s.period, 
    C1  <- s.C1, 
    T1i  <- s.T1i, 
    progress  <- "0"
}

helper content RTOSModelTask def :IntTask :String = 
  RTOSCheduleEventRule.allInstances() -
  >collect(e.Priority.toInteger()).isSet())>select(e | e > 0)-
  >list() ->tostring();
```

Figure 7: Example of Transformation rules

In order to do that, four tasks are taken with various characteristics. During the writing of the transformation rules, the scheduling of these tasks is made according to the scheduling algorithm Rate Monotonic [3] as shown in Figure 7. This code portion presented by Figure 7 shows the transformation of all class Task instances of the source model to that of the target model. It also shows through the helper, how to select the task having the highest priority.

5 Conclusions

The present paper demonstrates that the RTOS can be modeled in high level design. The challenge consists of the use of existing UML profiles in order to integrate the RTOS modeling by the definition of the transition and the temporal semantics.

At the level of the integration of the RTOS modeling in MDE approach, concepts endure abstract and are independent from realisation and specific platform execution.

The implementation of the semantic variation points offered by UML statecharts provides an efficient way to specify task management and real time scheduling.

Future work includes the focus on the transformation rules to generate the code.

6 References

[7] Gergely Pinter and Istvan Majzik. Impact of Statechart Implementation Techniques on the Effectiveness of


[10] Maria Cruz Valiente, Gonzalo Genova, Jesus Carretero. UML 2.0 Notation for Modeling Real Time Task Scheduling. Carlos III University of Madrid JOURNAL


Applying BSUP to Optimize RiskIT Methodology Based on UML 2.0

Ramin Nassiri  
Tehran Azad University  
R_nasiri@iauctb.ac.ir

Ali Moeini  
Tehran University  
moeini@ut.ac.ir

Pooya Khosravyan  
Islamic Azad University (Shahrekord)  
Khosravyan@gmail.com

Abstract
RiskIT is a now a world-wide risk management methodology deployed by a number of expert software engineering communities since its first rollout by J. Kontio et. al. [12][13][14][15] BSUP has been our proprietary Business to software modeling approach introduced for the first time in 2003[17][18]. In this paper we aim to apply the capabilities inherent in BSUP to optimize RiskIT process model. Considering the infrastructure of BSUP, UML 2.0 widely deployed anytime a model was to be made.

Keywords: BSUP, Fuzzy, RiskIT, RUP, UML.

1. Introduction
The RiskIT method for software engineering risk management is widely in use because of its sound theoretical foundation and its major focus on qualitative cognition of risks before their possible quantification, in addition to its capability to provide a defined process for conducting risk management. Today, it is being supported by various tools, techniques and also rich guidelines. But the fascinating feature may be that the use of RiskIT does not preclude the use of other risk management approaches [12] [14][15]. Since the early days of software development, risks had been perceived inevitable because of various unanticipated problems which cause development team to go over budget, miss deadlines, or finally deliver less than satisfactory artifacts and so on. Although risks neither could be eliminated nor might be ignored but one may strictly monitor and manage them to control and mitigate their potential harmful effects. Reflective and proactive methods are alternatives to achieve the Goal. There are a few factors which make the risk management process to be so sophisticated where most of current risk management methodologies fall into paralysis stage. Among them following major factors can be considered as major driving forces of the shortcomings of current methods:

- Risk is very seldom a crisp straightforward concept and mostly it is perceived as a non deterministic concept which is quite close to Fuzzy concepts and measures. This is also true while the impression of risk on various stakeholders is measured.
- Risks may influence each others in different ways. Risks may strengthen or diminish each other. This feature may make a very complicated scenario while analyzing the potential effects of risks.
- Clarity of methods and cost-effectiveness of many current risk management methods are totally in doubts since they are costly perceived as complex or too costly to use.
1.1 RiskIT Methodology
The RiskIT method is considered to be a suitable solution to address the issues such as those listed above. Its main characteristics can be described by the following principles.

1) The RiskIT method provides precise and unambiguous definitions for risks.
2) The RiskIT method results in explicit definition of objectives, constraints and other.
3) The RiskIT method is aimed at modeling and documenting risks qualitatively. The RiskIT method provides conceptual and graphical tools to model different aspects of risks qualitatively, instead of requiring quantitative estimation of risk probability and impact to take place early in the project. Given the difficulty of these estimations and the ambiguous interpretations of risks – the margins of error in risk quantification are easily high. By emphasizing the qualitative understanding of risks, there is a better basis for understanding and communicating about risk.
4) The RiskIT method can use both ratio and ordinal scale risk ranking information to prioritize risks reliably.
5) The RiskIT method uses the concept of utility loss to rank the loss associated with risk.
6) Different stakeholder perspectives are explicitly modeled in the RiskIT method.
7) The RiskIT method has an operational definition and training support.
8) The RiskIT Analysis graph, Fig 1, is a graphical formalism that is used to define the different aspects of risk more formally. The RiskIT analysis graph can be seen both as a conceptual template for defining risks, as well as a well-defined graphical modeling formalism. [15]

![Fig. 1 A conceptual view of the elements in the RiskIT Analysis Graph](image)

1.2 BSUP Methodology
There were always pros and cons about the importance of business modeling in the software development process which recently accepted and recommended as an essential stage prior to any further development by a few research communities in addition to commercial solutions such as Rational Unified Process (RUP). The Basic idea on behind is the transformation of traditional business to modern E-business environments which in turn promotes software from an external supportive element to a core resided component with direct and robust interaction with other core elements. Considering the new role of software in the E-business, one cannot proceed in the development process without complete and clear understanding of tight relationships between the business elements.
And software's which have mutual robust impression on each other. This is where the role of business modeling may be shown brilliant. By modeling the business from software engineering viewpoint, one can make sure that nearly all subtle effects and side-effects from both parties being captured in a straightforward manner without any unresolved ambiguity. We already proposed a method called BSUP, stands for Business to Software Unified Process, which models a business from five orthogonal aspects to build up a complete integrated view of business. In BSUP the business model is composed of five major components including: goal, process, rule, role and resource models. Fig.2 demonstrates the anatomy of the BSUP's business model. As one might expect, Business process model plays the major role and resides in the focal point of the whole model to integrate it. To reconcile the different nature of business model and correspondent software model(s), one may use the attractive strengths of UML to model the focused business. These approach ends to building of models which simply being perceived by software teams in addition to business experts with limited computer talent. (For details about using UML for business modeling, refer to [17]). Taking into account the robust capabilities provided by the UML 2.0, nearly all the five pillars of the model, as depicted above, may be implemented while some extension is done on it. The most significant parts of the business model reside in the Business goal and Business process models. Fig.3 and Fig.4 represents the overall schema of these two core models.

**Fig.2** Business Model core-Components diagram in BSUP

**Fig.3** Business Goal Meta-Model in BSUP
2. Applying BSUP on RiskIT

The core of RiskIT method resides in its analysis graph in addition to its definition of process that supports risk management activities throughout the project. As already mentioned above, The RiskIT analysis graph, Fig. 3, is a graphical formalism that aims to define various aspects of risk in a formal way. Basically the graph brings its idea from the what first developed by Rowe et.al. [20] and later completed by extension done by Knotio.[19] Considering the model proposed in BSUP(Fig. 4). The Analysis graph may be extended to be a complete well formed process model. Fig. 5 represented the new model. As it is visible in the figure, the proposed model transforms the RiskIT analysis graph to be more integrated and also clustered. The new model generated a complete process encompassing all the requirements and elements of a process including goal(s), resource(s), role(s) and etc by deploying the new model any inconsistency and/or inconvenience in the model could be easily recognized and rectified. It means that the model would be something more than an informative graph. The next step toward optimization would be the process definition in RiskIT. The definition in RiskIT is well-designed while major characteristic may be considered as:

- Ability of risk information prioritization based on available ordinal metrics.
- Providing method to distinguish goals and stakeholders within the scope of the software project.
- Providing a method to describe the risk management mandate including scope, procedures and authorities etc.
Fig. 6 depicts process definition schema in RiskIT as proposed by Kontio.

<table>
<thead>
<tr>
<th>Purpose:</th>
<th>Provide project and organization management with accurate and timely information of the risks in a project. Define and implement cost efficient actions to control risks.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>Monitor and manage risks continuously in a project.</td>
</tr>
<tr>
<td>Entry criteria:</td>
<td>Project planning has been initiated.</td>
</tr>
<tr>
<td>Input:</td>
<td>Project authorization information, goals, resources, schedule, budget, context and history information about the organization and its process.</td>
</tr>
<tr>
<td>Output:</td>
<td>Continually updated information about risks. Defined and implemented risk controlling actions. Experience and data about risks and risk management process.</td>
</tr>
<tr>
<td>Responsibility:</td>
<td>Project manager.</td>
</tr>
<tr>
<td>Exit criteria:</td>
<td>Project has been completed or terminated.</td>
</tr>
</tbody>
</table>

Fig. 6 Process definition for the whole RiskIT process

Again this design may be optimized by deploying the BSUP process model capability. The process model for RiskIT may be optimized as Fig. 7.

9- Conclusions

This paper presented an operational optimization of the RiskIT method by using the BSUP, our proprietary methodology for business software modeling. It must be granted that the scope of optimization for RiskIT is much larger than what shown in the paper. But it has been clarified the strengths of BSUP and typical areas that those strengths can be employed to enrich the RiskIT models. To some degree the operational definition given here is perhaps too general for casual readers who want to read an overview of the method. So it is strongly encouraged for such readers to review relevant documents in addition to this paper. With using this method and applying these changes mentioned in this paper we can do achieve better model(s) for software risk management in a project.

10- References

Computer Society Press. Washington, DC.
[23] UML 2.0, Martin Fowler, 2006
Error Detection in Multiple State Diagrams

Mohammad N. Alanazi and David A. Gustafson
Computing and Information Science Department
College of Engineering
Kansas State University
Manhattan, KS 66506, USA

Abstract - This paper presents Super State Analysis (SSA) for analyzing UML multiple state and sequence diagrams to detect the inconsistencies. SSA model uses transition set that captures relationship information that is not specifiable in UML diagrams. The SSA model uses the transition set to link transitions of multiple state diagrams together. The analysis identifies five types of inconsistencies: impossible super states, unreachable super states, illegal transitions, missing transitions, and illegal sequences.

Keywords: UML, Modeling Languages, State Diagrams, Sequence Diagrams.

1 Introduction

Unified Modeling Language (UML) has been widely used as a standard language for modeling software. UML 2.0 [10] consists of thirteen types of diagrams: class, composite structure, component, deployment, object, package, activity, use case, statechart, sequence, communication, interaction overview, and timing. Each diagram is dedicated to a different design aspect. Many different UML diagrams are usually involved in software development. Using more than one diagram to design a system is necessary but can leave the system in an inconsistent state and hence produce errors. Finding inconsistencies in software design before the design is implemented is very important. “Error detection and correction in the design phase can reduce total costs and time to market” [12].

A consistency problem may arise due to the fact that some aspects of the model will be described by more than one diagram. Hence, we should pay much attention to the consistency in the early phases of the system development and it is important that the consistency of a system should be checked before implementing it [9]. To avoid such errors, we should check the consistency among the diagrams and make sure that the diagrams are consistent.

Many researchers found that the problem of ensuring consistency between UML diagrams has not been solved yet [3]. The UML specification does not enforce many consistency requirements between the information contained in the sequence and state diagrams. While this does allow for greater flexibility in how UML can be used, it can lead to inconsistent views of the system being modeled. “The problem of relating state-based intraagent (or intraobject) behavioral descriptions with scenario-based interagent (or interobject) descriptions has recently focused much interest among the software engineering community” [1]. Identifying inconsistencies between UML diagrams can help the developers to find errors and fix them at early stages. Furthermore, current UML CASE-tools (e.g. Rational Rose) provide poor support for maintaining consistency between UML diagrams. So, helping to solve this problem can make a great contribution to the software development process.

2 The Super State

Our approach for consistency analysis combines the state information of multiple state diagrams into a composite super state, SS. The super state has the form [s₁, s₂, ..., sₙ] where sᵢ is the state of object i and n is the total number of objects. This super state details all of the possible composite states the objects can be in as well as the transition pairs which lead from one composite state to another. In this way the super state provides the complete collaborative view of a set of objects in the model.

SS is changed after each message call. For every call we have <SS₀, call, SS₀> where SS₀ is the super state before call and SS₀ is the super state after the message call has been called. In SS₀, only the state of one object is changed. This object must be the destination object of the message call. The state of the other objects remains in the same state as they were before the call. We calculate the super state of multiple state diagrams after each valid transition and that is used to evaluate each sequence diagram. A valid sequence diagram should be a subsequence of the set of sequences that are possible in a super state. Invalid and impossible sequences can be identified.

3 Super State Analysis (SSA)

The information in UML diagrams is related to each other and represent different views of a system. Hence, they can be validated against each other. Given a statechart diagram, researchers [9] have shown how to validate it
against a sequence diagram. On the other hand, given a sequence diagram, it can be validated against a statechart diagram [2, 13].

However, we are proposing a new approach to check the consistency between multiple state diagrams and one or more sequence diagrams. Our analysis, the Super State Analysis (SSA), focuses on multiple state diagrams instead of a single state diagram.

The diagram on Figure 1 shows the complete analysis process and the relationships between the different sources of information. Some information is known from the domain knowledge and provided by the developer while some other information is extracted from the existing information and generated automatically. SSA uses the provided information to generate some information automatically. Comparing the information from different sources allows us to detect the inconsistencies. SSA includes some inconsistencies that can be detected by the computer and some other faults that can be identified by the human. SSA performs five types of comparisons to detect the inconsistencies.

The SSA model on Figure 1 includes the 12 information sets that are involved in SSA model. The system developer provides the UML state diagrams, the transition set and UML sequence diagrams (D1, D2, and D3). The developer identifies the valid super states, invalid super states, valid single step transitions, and the invalid single step transitions (H1, H2, H3, and H4). The SSA tool is automatically generates three large sets: set of all generated super states, set of all generated single step transitions, and set of all generated sequences (T1, T2, and T3). These sets are generated using the UML state diagrams and the provided transition set. The valid sequences (S) are extracted from the UML sequence diagram.

The SSA tool uses the UML state diagram (D1) and the transition set (D2) to generate the set of all generated Super States (T1). Also, the tool uses the transition set (D2) to compute the set of all generated sequences (T3). Moreover, the tool uses the transition set to compute the set of all generated single step transitions (T2). The developer uses the domain knowledge to identify the valid super states, invalid super states, valid single step transitions, and invalid single step transitions. Furthermore, the UML sequence diagram is used to extract the sequences which will be compared to the set of all generated sequences.

The Super State Analysis consists of five types of comparisons to detect the inconsistencies in the multiple state diagrams and sequence diagrams.

1. C1: Compares the set of all generated super states (T1) with the set of valid super states (H1).
2. C2: Compares the set of all generated super states (T1) with the set of invalid super states (H2).
3. C3: Compares the set of all generated single step transitions (T2) with the set of valid single step transitions (H3).
4. C4: Compares the set of all generated single step transitions (T2) with the set of invalid single step transitions (H4).
5. C5: Compares the set of all generated sequences (T3) with the set of sequences (S) which are extracted from the provided UML sequence diagrams.

C1 and C2 detect the valid and invalid super states while C3 and C4 identify the illegal and missing transitions. C5 detects the invalid sequences. This comparison is fully
automated since both T3 and S are generated automatically. The other four comparisons can be automated if we formalize the four sets: H1, H2, H3, and H4 and feed them to the system. By comparing these four sets to the generated sets: T1 and T2 the inconsistencies can be detected automatically.

4 Inconsistency Detection

Super State Analysis (SSA) discovers inconsistencies in super states, single step transitions, and sequences.

4.1 State Inconsistencies

The valid and invalid states will possibly be identified by SSA. If a Super State (SS) is generated by Box T1, but it is not in the set of valid states (Box H1) then the state is an invalid SS. This could happen if there is a wrong transition in the transition set. On the other hand, if a Super State is in the set of valid states (Box H1), but it is not generated by Box T1, then this SS is a valid super state and should be generated. SS wouldn’t be generated if there is a missing transition in the transition set or in the state diagram.

The following kinds of inconsistencies can be discovered by this analysis:
   i. Impossible super states
   ii. Unreachable super states

4.2 Single Step Transitions Inconsistencies

The valid and invalid single step transitions (Box H3 and Box H4) are known from the domain knowledge. The set of all generated single step transitions (Box T2) are generated automatically using the transition set. Comparing those sets will discover some legal and illegal transitions.

If a valid transition does not appear in the set of all generated single step transitions that means this transition is missing. Furthermore, if an invalid transition appears in the set of all generated single step transitions that mean this transition is illegal.

The following kinds of inconsistencies are discovered by this analysis:
   i. Illegal transitions
   ii. Missing transitions

4.3 Sequence Inconsistencies

The tool generates the sequences using the transition matrix. To validate a UML sequence diagram, the tool extracts the sequences first (Box S), then, compares them to the set of all generated sequences (Box T3). If there is a matching sequence in that set, this sequence is valid. Otherwise, it is an invalid sequence.

The following kinds of inconsistencies are discovered by this analysis:
   i. Illegal sequences

The SSA tool uses the UML state diagrams and the transition set to generate the set of all generated Super States (SS). Also, the tool uses the transition set to compute the set of all generated sequences. Moreover, the tool uses the transition set to compute the Set of all generated single step transitions.

From the domain knowledge, we identify the sets of valid and invalid Super States (SS) and the valid and invalid single step transitions.

The UML sequence diagram is used to extract the sequences which will be compared to the set of all generated sequences.

The inconsistency can be fixed by several ways. It can be fixed by adding or removing a fact to the domain knowledge. Another way to fix the inconsistencies is correcting the state diagram by adding a new transition (or removing one).

5 The Transition Matrix

The transition matrix details the possible global states of the system based on a vector of states of individual instances of classes and the possible transitions between the states in the super state (SS). Consider a program that has class X and class Y. Let class X have an initial state A and two other states, B and C, while class Y has an initial state D and a second state E. Figure 2 shows the state diagram for class X and Figure 3 shows the state diagram for class Y. The state diagrams depict how instances of X and Y can transition between those states. Let class Y makes the transition between state D and state E whenever class X makes the transition from state A to state B. Table 1 shows possible transitions in the super state that is the cross-product of all states with one instance of X and one instance of Y.

An entry in a cell in T1 (Table 1) shows that in one step, the system can transition from the state of the row to the state of the column. Taking the product of T1 by itself gives a matrix that contains the transitions possible with two steps. The closure of T1 is the sum of products, \( T_1 + T_1*T_1 + T_1*T_1*T_1 + \ldots \). The closure shows all possible transitions in any number of steps. Although the closure is represented as an infinite sum, it can be calculated in at most the number of products equal to the rank of the initial matrix. In most cases, it is even smaller than that number.
Table 1: Super state transition matrix $T_1$

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>AD</th>
<th>BD</th>
<th>CD</th>
<th>AE</th>
<th>BE</th>
<th>CE</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>BD</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CD</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>AE</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>BE</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>CE</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

6 Library System Example

This example describes the interaction between a patron of a library and the copies of books the library holds. In order to simplify the model the library holds only one copy of each book. Figure 4 shows the class diagram for this model. Figure 5 and Figure 6 are the state diagrams for the patron and book objects. Note that the transitions in the state diagrams are numbered for ease of reference. This example originally was created by a team of students trying to create a correct model of a simple library system.

The patron object can be in one of three states: Good Standing, Too Many Books, and Overdue Fines. We will call these states $G$, $T$, and $F$, respectively. A patron starts in $G$ until the number of books the patron has checked out is equal to $MAX$ or the patron returns an overdue book. In the former, the patron will transition to state $T$ where they will remain until they return a book. In the latter, the patron will transition to $F$ where they will not be able to do anything until they pay the fine that is owed.

A book object has six states: On Shelf, Missing, On Hold, Checked Out, Overdue, and Returned. We will call these states $O$, $M$, $H$, $C$, $D$, and $R$, respectively.

The two transitions from $C$ labeled check represent the library determining if the book is overdue. If the book is overdue it will transition to $D$. Otherwise, it will transition to $R$ where it will remain until the library places it back on the shelf.

For our analysis we will assume the library has only one patron and three books. We now pair the transitions from the patron and book objects that can occur together. An ‘X’ indicates that we are not concerned about the state of the object. The transition set is shown in Table 2.

The initial transition matrix $A_1$ has column and row headings with a quadruple representing the states of the four objects. For this model there are $3^4 = 81$ combinations of the four objects. Table 3 shows a portion of the initial transition matrix $A_1$.

<table>
<thead>
<tr>
<th>$A_1$</th>
<th>GOOO</th>
<th>GOCO</th>
<th>GODO</th>
<th>GORO</th>
<th>GCOO</th>
</tr>
</thead>
<tbody>
<tr>
<td>GOOO</td>
<td>1,21</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GOCO</td>
<td>26</td>
<td>23</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GODO</td>
<td></td>
<td></td>
<td></td>
<td>2,22</td>
<td></td>
</tr>
<tr>
<td>GORO</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>GCOO</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The row headings are the initial states and the column headings are the final states. The numbers in the table arise...
Table 2: Transition set for Library Example

<table>
<thead>
<tr>
<th>( SS_{pre} \rightarrow SS_{post} )</th>
<th>Transition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GOXX (\rightarrow) GCXX</td>
<td>checkout([n&lt;MAX]), checkout</td>
<td>Check out a book (if at least one X = O | R )</td>
</tr>
<tr>
<td>GXOX (\rightarrow) GXCX</td>
<td>checkout([n&lt;MAX]), checkout</td>
<td></td>
</tr>
<tr>
<td>GXOX (\rightarrow) GXCX</td>
<td>checkout([n&lt;MAX]), checkout</td>
<td></td>
</tr>
<tr>
<td>GOXX (\rightarrow) TCXX</td>
<td>checkout([n&lt;MAX]), checkout</td>
<td>Check out a book (if X = C | H | D)</td>
</tr>
<tr>
<td>GXOX (\rightarrow) TXCX</td>
<td>checkout([n&lt;MAX]), checkout</td>
<td></td>
</tr>
<tr>
<td>GXOX (\rightarrow) TXCX</td>
<td>checkout([n=MAX]), checkout</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) GRXX</td>
<td>return, return</td>
<td>Return book on time</td>
</tr>
<tr>
<td>GCXX (\rightarrow) GRXX</td>
<td>return, return</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) GRXX</td>
<td>return, return</td>
<td></td>
</tr>
<tr>
<td>GDXX (\rightarrow) FRXX</td>
<td>return({\text{returnDate}&gt;\text{due Date}}), return</td>
<td>Return an over due book</td>
</tr>
<tr>
<td>GDXX (\rightarrow) FRXX</td>
<td>return({\text{returnDate}&gt;\text{due Date}}), return</td>
<td></td>
</tr>
<tr>
<td>GDXX (\rightarrow) FRXX</td>
<td>return({\text{returnDate}&gt;\text{due Date}}), return</td>
<td></td>
</tr>
<tr>
<td>TCXX (\rightarrow) GRXX</td>
<td>return({\text{returnDate=\text{due Date}}}), return</td>
<td>Patron with MAX books returns a book on time</td>
</tr>
<tr>
<td>TCXX (\rightarrow) GRXX</td>
<td>return({\text{returnDate}=\text{due Date}}), return</td>
<td></td>
</tr>
<tr>
<td>TCXX (\rightarrow) GRXX</td>
<td>return({\text{returnDate}=\text{due Date}}), return</td>
<td></td>
</tr>
<tr>
<td>TXOX (\rightarrow) FRXX</td>
<td>return({\text{returnDate}&gt;\text{due Date}}), return</td>
<td>Patron with MAX books returns an over due book</td>
</tr>
<tr>
<td>TXOX (\rightarrow) FRXX</td>
<td>return({\text{returnDate}&gt;\text{due Date}}), return</td>
<td></td>
</tr>
<tr>
<td>TXOX (\rightarrow) FRXX</td>
<td>return({\text{returnDate}&gt;\text{due Date}}), return</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td>Patron lost a book</td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FXMX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FXMX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FXMX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FXMX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
<tr>
<td>GCXX (\rightarrow) FMXX</td>
<td>lose_by_patron, lose_by_patron</td>
<td></td>
</tr>
</tbody>
</table>

From Figure 5 and Figure 6. For the purpose of clarification we have assigned unique numeric identifiers to the transitions for each instance of an object in our system. The book object has three numeric identifiers for each transition since we have three instances of that object. For example, \( GOOO \rightarrow GOCO \) represents a patron in good standing checking out the second book. The 1 indicates the patron took the transition labeled checkout \([n<MAX]\) and the 21 indicates the second book took the transition labeled checkout. If there is an entry for a cell in the matrix then the transition is valid. \( A_2 \) is defined as \( A_1 \ast A_1 \) which identifies all the states we can reach in two steps. Table 4 shows a portion of \( A_2 \).

<table>
<thead>
<tr>
<th>( A_2 )</th>
<th>( GOOO )</th>
<th>( GOCO )</th>
<th>( GODO )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( GOOO )</td>
<td>(1,2)(26)</td>
<td>(1,2)(23)</td>
<td></td>
</tr>
<tr>
<td>( GOCO )</td>
<td>(2,22)(25)</td>
<td>(26)(26)</td>
<td>(26)(23)</td>
</tr>
<tr>
<td>( GODO )</td>
<td>(25)(21)</td>
<td>(25)(12)</td>
<td></td>
</tr>
<tr>
<td>( GCOO )</td>
<td>(2,12)(15)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

From Table 4 we can observe that it is possible to go from \( GOCO \) to \( GOOO \) by first returning the second book and then shelving it.

For this model, the unreachable states include two sets. The first set includes the states where the patron is in \( T \) and one of the three books is in \( O \) or \( R \). Clearly the patron cannot have \( MAX \) books checked out if one of the books is not checked out. The other set of unreachable states occurs when the patron is in \( F \) and all books are in \( C \) or \( D \). In order for the patron to be in \( F \), one of the three books would have had to have been returned. An analysis of \( A^* \) for this model shows that the columns for these unreachable states are empty.

Some of the faults in the design of the library example can be discovered by simply analyzing the transition matrix. One such fault was a missing transition. From \( FCRO \) and \( FCRO \) there is no valid single step transition to \( FRRO \). This means that if one book is returned late, the patron goes to \( F \) status and cannot return the other book until the fine is paid.

7 Related Work

There are several different approaches that have been proposed to perform consistency checking between UML diagrams. Some approaches use transformation to convert one diagram to another [12, 3, 13, 16, 14, 15] while others detect the inconsistencies by comparing one diagram to another using consistency rules [9, 4]. Moreover, many approaches use formalism, such as OCL and Z, to enforce the consistency [2, 5, 7, 6].

Almost all approaches focus on all or some of the six types of UML diagrams. Namely, use case, class, object, sequence, collaboration, and statechart diagram. [8] studies the consistency between use case, class, sequence, and statechart diagram. [3] studies the consistency between class, object, sequence, collaboration, and statechart diagram. [5] studies use case, class, sequence, and
statechart diagram. [14] studies the consistency between three diagrams: class, sequence, and statechart diagram. [9, 2, 13, 16] study the consistencies between sequence and statechart diagrams. [11] studies the class diagram and statechart diagram.

The researchers pay attention to enforce consistency between only two diagrams (e.g. single sequence diagram vs. single statechart diagram). However, our approach is unique in that we are proposing a new technique to check the consistency between multiple state diagrams and one or more sequence diagrams. Moreover, the approach focuses on multiple state diagrams instead of a single state diagram.

7.1 Transformation

The consistency checking in the transformational approach is done in two steps. First, the UML diagrams are converted to interpreted diagrams. Second, the interpreted diagrams are compared to each other to detect the inconsistencies.

Alexander Egyed [3] presents a transformation-based approach to consistency checking. They define a set of model transformation rules to enable the conversion of one UML diagram into another. They also define a set of comparison rules to compare the transformed diagram with an existing one of the same type. For example, to check for inconsistencies between a sequence diagram and a class diagram, they first transform the sequence diagram into an interpreted class diagram. The interpreted class diagram is then compared with the existing class diagram. This approach needs two sets of rules: transformation rules and consistency rules. If one diagram can’t transform to another, then both diagrams transformed to an intermediate diagram to compare.

Hongyuan Wang et al. [16] propose an approach that checks the consistency between sequence diagrams and state diagrams. The approach converts statecharts using Finite State Processes and transforms sequence diagram to messages trace. They use an existing tool LTSA to support their approach. However, the approach considers only single sequence diagram and single statechart diagram.

Wuwel Shen et al. [13] propose to build a message graph from a state chart diagram and then go through the graph based on the sequence of messages retrieved from a sequence diagram to find any inconsistency between these two diagrams. Based on this method, a tool called ICER is developed to provide software developers with automatic consistency checking in the dynamic aspects of a model. However, the approach considers only single statechart vs. single sequence diagram.

Orest Pilskalns et al. [12] present an approach that combines structural and behavioral UML representations in order to derive and execute test cases to validate a UML model. They develop a method for encapsulating the behavioral aspects (i.e. message paths between objects) that exists in sequence diagrams into a directed acyclic graph. The objects in the graph are then associated with class attribute/parameter values which are used to generate and execute test cases. Their approach would require OCL object constraints to be written.

7.2 Consistency Rules

In consistency rules approach, the consistency is checked using the set of consistency rules. The diagrams are compared to each other directly without transformation or formalism.

Boris Litvak et al. [9] present an approach to consistency checking between UML sequence and state diagrams. They created the BVUML (Behavioral Validator of UML) tool which automates the behavioral validation process. Their approach associates states with only one object lifeline in the sequence diagram so a single run of the tool validates consistency for only one object. Therefore, the tool must be run multiple times in order to check the consistency of an entire sequence diagram.

Alexander Egyed [4] introduces an approach for quickly, correctly, and automatically deciding what consistency rules to evaluate when a model changes. The approach does not require consistency rules with special annotations. Instead, it treats consistency rules as black-box entities and observes their behavior during their evaluation to identify what model elements they access. The UML/Analyzer tool integrated with Rational Rose fully implements this approach. It was used to check 24 types of consistency rules. The author found that the approach provided design feedback correctly and required, in average, less than 9 ms evaluation time per model change with a worst case of less than 2 seconds at the expense of a linearly increasing memory need.

7.3 Formalism

Since UML is not precise enough, some researchers formalize the UML diagrams to some formal languages (e.g. Z). They then compare this formalism to detect the inconsistencies between the diagrams.

Yves Dumond et al. [2] show that it is possible to integrate semi-formal and formal methods for the dynamic behavior of the UML models. The objective is to favor the integration of formal techniques in the actual practice of software engineering. They introduce an approach to formalize sequence diagrams and verify coherence with the statechart diagrams. The approach translates the UML sequence diagrams into the pi-calculus, by preserving the object paradigms. To preserve the object notation, they name the pi-calculus processes with the name of the objects. The consistency between sequence diagrams and statechart diagrams can be checked by verifying that the messages in the sequence diagrams trigger states in statechart diagrams.

Krishnan [7] describes a framework in which UML diagrams can be formalized to perform consistency checking. UML diagrams are translated into specifications of the theorem proving tool PVS (Prototype Verification System). The PVS is a language that allows for the introduction of abstract data types, functions, etc. To check
for consistency between sequence and class diagrams, the class diagrams must first be annotated with OCL constraints. The PVS will check if the sequence of states described in the sequence diagram can be obtained from the class diagrams. Custom traces (i.e. sequence of states) can also be supplied by the user to check if other properties hold.

Soon-Kyeong Kim and David Carrington [6] describe how consistency checking between different UML models can be accomplished by using a formal object-oriented metamodeling approach. They formally define the abstract syntax and semantics of the UML model using Object-Z as a metalanguage. They then define consistency constraints that logically exist between semantically equivalent elements in the metamodel but are not defined in the current UML metamodel structure. Once the consistency constraints have been defined for each of the UML model elements, consistency checking between different model elements can be achieved by verifying that the combined models preserve all of the consistency constraints for the individual model elements. They use the formal language to ensure the consistency between two diagrams.

8 Conclusion and Future Work

To avoid errors in UML diagrams, we should check the consistency among the diagrams and make sure that the diagrams are consistent to each other. To accomplish this, we have proposed this work to identify the problem that may arise due to the fact that some aspects of the model will be described by more than one diagram. We proposed a solution that analyzes the multiple UML state diagrams and UML sequence diagrams to detect inconsistencies in states, single step transitions, and sequences. Our approach, The Super State Analysis (SSA), identifies five types of inconsistencies: impossible super states, unreachable super states, illegal transitions, missing transitions, and illegal sequences. The Super State Analysis (SSA) detects the inconsistencies by computer and human.

In the future, we are planning to expand the case study to a bigger one with more states and sequence diagrams. We will consider more interaction between the state diagrams with different number of instantiations. Furthermore, we will investigate the different type of inconsistencies using the five comparisons. Moreover, we will use the set notations to formalize the different sets that are involved in the comparisons. Also, we plan to build the SSA tool to perform all five comparisons automatically. We are also developing approaches to minimize the state explosion to allow the SSA to scale to larger systems.

9 References


P.A.D.I.C - Assessment & Measurement Based Framework to Improve Productivity and Predictability in Engineering Projects

Annie Abraham, Raghu Subramanian, and Rose Neena Tom
Engineering Excellence Group, HCL Technologies Ltd, Bangalore, Karnataka, India

Abstract

P.A.D.I.C framework has been conceptualized as a continuous improvement model to focus on organizational performance targeting enhanced productivity and predictability within engineering projects. Engineering practices need to be heavily focused on the ideas of practice maturity, an understanding of productivity, collaboration of process, tools, knowledge management and the search for improvement. Equally significant is the fact that in the global zeal to improve reliability of products developed, the practices that are internalized among the practitioners decrease rather than increase the predictability of outcomes. Consequently a universal need for the paradigm shift from 'Quality of Deliverables' to 'Quality of Outcomes' has been generated. This framework is a holistic approach marrying the proven methodologies of P.D.C.A cycle combined with statistical process control measures applied on the defined processes and practices. In this paper we attempt to describe the conception of the framework and application in projects along with performance models and measurement procedures defined at organizational level. It is deployed as a diagnostic tool for assessing the health of projects and it is expected that implementation results may assist the practitioners to understand the gaps and adopt approaches to bridge them.

Keywords: P.A.D.I.C Framework – Plan, Assess, Analyze, Define, Control

1 Introduction

1.1 Productivity

Current economic realities (liberalized and dynamic markets, constantly changing customer preferences, new structure of production and work, etc.) are leading to a rethinking of the notion/concept of productivity. Whereas traditionally, productivity is viewed mainly as an efficiency concept (amount of outputs in relation to efforts or resources used), it is now viewed increasingly as an efficiency and effectiveness concept [4], effectiveness being how the enterprise meets the dynamic needs and expectations of customers (buyers/users of products and services) i.e. how the enterprise creates and offers customer value. Productivity is now seen to depend on the value of the products and services (utility, uniqueness, quality, convenience, availability, etc) and the efficiency with which they are produced and delivered to the customers.

The structures of the production-distribution systems are also changing. Products and services and hence customer values are increasingly created through enterprise networks, supply-chains and value-chains that even extend beyond national boundaries. In these situations where an enterprise relies on network of suppliers, service providers, extended and disaggregated supply and delivery networks, its effectiveness and efficiency are very much dependent on the way it manages its value-chain [2].

To be relevant in the dynamic and changing environment, the productivity improvement effort must focus on:

- Doing the right things (know ‘what’ to produce and distribute) by continuously reviewing and identifying the changing customer needs and expectations and developing and designing products and services to best satisfy the needs and meet the expectations. Consequently create more customer values.

- Doing things right (know ‘how’) by constantly improving production and distribution processes to produce and deliver the goods and services in the most efficient way.

Productivity is not just an efficiency concept: First, there is increasing appreciation that productivity is not just an efficiency concept but that -equally important- it is also an effectiveness concept. Customer orientation is now a primal consideration and superior quality is up front an indicator of good productivity performance. Productivity is getting synonymous to quality [2, 4].
1.2 Predictability

The best way to achieve predictable software development outcomes is to start early, learn constantly, commit late, and deliver fast. This may seem to cut against the grain of conventional project management practice, which is supposed to give more managed, predictable results. But with predictability, the confidence on a shifting foundation cannot be built. The problem with conventional approaches is that they assume the foundation is firm: they have little tolerance for change.

The paradox is that trying too hard to create predictability creates opposite effect. Conventional practices are fragile in the face of change, and even in the face of learning. And yet, the more complex the system, the more necessary learning becomes. What is needed is an approach that encourages learning, and does not commit until learning is complete. The next section introduces a framework for addressing these changing demands both internal and external to the business organization. It should be obvious that decreasing the amount of speculation involved in making a decision increases predictability of the outcome. If you can make decisions based on facts rather than forecasts, you get results that are more predictable [6].

The remainder of this paper is organized as follows: Section 2 describes the newly instituted framework with its components, objectives, methodologies. Section 3 provides an overview of process performance model and the supporting models, the ‘effort’ & ‘defect’ prediction models that are defined at the organization level and deployed as part of improvement initiatives. Section 4 elaborates the functioning of the model with its constituent analysis and supporting documentation. Section 5 describes the case study of application of this framework.

2 P.A.D.I.C Framework

Fig. 1 P.A.D.I.C Framework

The P.A.D.I.C framework enables identifying gaps in engineering practices, initiating improvement programs to bridge the gap, instilling a better culture of adoption of tools, knowledge sharing, and thereby bringing in a higher maturity among engineering practice levels.

P.A.D.I.C framework has been formulated as a customized format from these models:

- EFQM (European Foundation for Quality Management)
- DMAIC principles of Six Sigma.
- CMMi (Capability Maturity Model – Integrated)
- Lean software development principles

These guidelines and models have been referred in the conceptualization of a total approach to evaluate and analyze engineering projects.

2.1 Framework Methodology

Having adopted a review & evaluation mechanism as the fundamental approach to target the productivity and predictability improvement goals for the organization the next step that was undertaken was to define the methodology for the same.

Given below is its graphical representation:

Fig. 2 P.A.D.I.C Methodology

2.2 Framework Objectives

The primary objectives that have been derived to fit this framework over the regulatory process compliance measures are:

1. Bring out strategic inputs to improve Productivity & Predictability in projects and engagements
2. Identify opportunities to strengthen key process areas
3. Identify practice gaps and suggest actions and provide support in:

Fig. 2 P.A.D.I.C Methodology

The P.A.D.I.C framework enables identifying gaps in engineering practices, initiating improvement programs to
a. Promoting reuse of good practices across projects
b. Encouraging greater use of tools
c. Facilitating knowledge sharing practices
d. Elevating engineering practices to higher maturity levels.

3 Process Performance Model

Process Performance model is an important aspect of any organization’s quality management system, which is used to estimate or predict the value of a process performance measure from the values of other processes and product measurements. Following are some of the basic constituents of Process Performance model:

1. Relationship Matrix: It is the mapping of organizational objectives to its processes, subprocesses, metrics and their baselines. The aim is to achieve business objectives by controlling baselines through effective process and sub process management.

2. Sub-process components: Sub-processes are defined components of a larger defined process. For e.g. a typical Requirements process may be defined in terms of requirements elicitation, analysis, questionnaire & brainstorming (for implicit requirements collection). However the level of sub process break up depends on the project life cycle and its corresponding requirements.

3. Prediction Models: Prediction Models are deployed to predict the achievement of overall capability of the identified processes through appropriate sub processes. Statistical methods are utilized to predict the performance based on data collated from process and sub-process capabilities. These prediction models are based on Linear Regression Statistical Model. The input for prediction would be the base line values for effort and defects.

4. Earned Value Management (EVM): EVM is a management methodology for integrating schedule, scope, resources, and objectively measuring project performance and progress.

Process Performance Models are to be used in the planning phase of projects to support the planning activity. If the project specific processes have been tailored, (e.g. the standard life cycle phases are not followed, use of different project execution methodologies), the performance model definition is revisited to confirm its applicability and if needed recalibrated.

3.1 Business Objective – Process Mapping

Business objectives are the driving force behind any organization’s process and policy planning. Some of the perspectives that are focused while defining the organizational objectives pertain to Financial, Customer Related, Process and Organizational capability management.

Various phases involved in navigating the organization towards competitive success [4] are defining business objectives, measuring the process/practice behavior and predictions about the performance. The following table identifies the relationship between the business objectives and various processes as defined in the process framework.

<table>
<thead>
<tr>
<th>Business Objectives to Process Mapping</th>
<th>Increase Customer Satisfaction</th>
<th>Increase Profitability</th>
<th>Improve Product Quality</th>
<th>Timely Delivery</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Planning</td>
<td>Indirect</td>
<td>Direct</td>
<td>Indirect</td>
<td>Direct</td>
</tr>
<tr>
<td>Project Tracking</td>
<td>Indirect</td>
<td>Direct</td>
<td>Indirect</td>
<td>Direct</td>
</tr>
<tr>
<td>Project Closure</td>
<td>Direct</td>
<td>Direct</td>
<td>Indirect</td>
<td>Indirect</td>
</tr>
<tr>
<td>Configuration Management</td>
<td>Indirect</td>
<td>Indirect</td>
<td>Indirect</td>
<td>Indirect</td>
</tr>
<tr>
<td>Software Quality Assurance</td>
<td>Direct</td>
<td>Direct</td>
<td>Indirect</td>
<td>Direct</td>
</tr>
</tbody>
</table>

Fig. 3 Process Mapping

3.2 Business Objective – Metrics Mapping

The following table identifies the relationship between some of the business objectives and the metrics that are identified for each of process / sub process.

<table>
<thead>
<tr>
<th>Business Objectives to MetricsMapping</th>
<th>Increase Customer Satisfaction</th>
<th>Increase Profitability</th>
<th>Improve Product Quality</th>
<th>Timely Delivery</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schedule Variation</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Effort Variation</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Customer Satisfaction Index</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Defect Removal Efficiency</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Productivity</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 4 Metrics Mapping

3.3 Prediction Models – Effort Prediction Model

Effort Prediction model, based on regression model, uses the effort variation data from life cycle phases to predict the projects overall effort variation from the latest baseline. The linear regression equations are derived based on this data.

Note: The figures and data appearing in the models are only for illustrative purposes. The organizational baselines have been applied during the actual implementation of these.
### 3.4 Prediction Models – Defect Prediction Model

This mechanism has been developed for the full cycle development or enhancement projects based on linear forecasting model. Defects for subsequent phases are forecasted with inputs from the baseline phase-wise derived figures and existing phase defect data. Baseline phase-wise weighted defect density figure is dependent on the size unit defined for a particular project.

![Defect Prediction Model](image)

**Table 1: Working Table**

<table>
<thead>
<tr>
<th>Phase</th>
<th>Defect Distribution across the phases - EUC (base level)</th>
<th>Estimated Defect for each phase</th>
<th>Annual Defect Found</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirement</td>
<td>7.42%</td>
<td></td>
<td>8.89%</td>
</tr>
<tr>
<td>Design</td>
<td>23.03%</td>
<td></td>
<td>28.83%</td>
</tr>
<tr>
<td>Coding</td>
<td>36.57%</td>
<td></td>
<td>40.73%</td>
</tr>
<tr>
<td>Testing</td>
<td>38.83%</td>
<td></td>
<td>44.89%</td>
</tr>
<tr>
<td>Overall</td>
<td>86.89%</td>
<td></td>
<td>100.00%</td>
</tr>
</tbody>
</table>

**Table 2: Predictions Table**

<table>
<thead>
<tr>
<th>Phase</th>
<th>Initial Estimate</th>
<th>Predicted Defect after Requirement</th>
<th>Predicted Defect after Design</th>
<th>Predicted Defect after Coding</th>
<th>Actual Defects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirement</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Design</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coding</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Testing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 3: Error Prediction Model**

![Fig. 5 Effort Prediction Model](image)

Any slippage between the predicted and actual data is analyzed to deliver a better quality product.

### 3.5 Earned Value Management (EVM)

EVM is a project management technique that provides a unique advantage in that it combines the measurement of technical performance (accomplishment of planned work), schedule performance (behind or ahead of schedule), and cost performance (under or over budget) using a single integrated methodology.

EVM aids in effectively measuring and controlling project risks by providing measures in monetary terms. The parameters that are analyzed as part of EVM are:

1. Planned Value: Planned cost of the total amount of work scheduled to be performed by the reporting date.
2. Earned Value: Planned cost for the completed work.
3. Actual Cost: Cost incurred to accomplish the completed work.
4. Budget at Completion: Total planned value at the end of the project.
5. Reporting Date: Date on which earned value and actual cost are measured for project activities that are either completed or in progress.

### 4 P.A.D.I.C - Functioning

In the current practice process compliance levels are periodically evaluated and reported by the process implementation group. This group defines process, implements them and imparts training session to update the practitioners in process changes and improvements. A detached evaluation by an internal group to gauge the practice levels of engineering processes and procedures in the system was lacking. Furthermore, the increasing demands from the market was forcing the business organization to measure how well the organization converts input resources (labor, materials, machines etc.) into goods and services and consequently to define the baselines for productivity with futuristic improvement goals.

### 4.1 Framework - Institutionalizing

The P.A.D.I.C framework has been instituted as a multi-dimensional engine to capture many parameters that are relevant to analyze, define and implement the improvement initiatives. The outputs are aimed at assisting the organization in internalizing applicable & mature practices, devising program to boost it to higher maturity levels.

![P.A.D.I.C Framework](image)
The assessment or evaluation of practices being the key instrument in this framework, application of the same in conjunction with data management process is very critical. This framework executed along with analysis of process performance measures helped in identifying the glaring gaps in practice maturity levels.

4.2 Improvement Framework

To address the gaps in current practices and to implement the recommendations that are generated as the outcome of P.A.D.I.C the following methodology has been adopted borrowing the PDCA principle.

5 Case Study - Applying P.A.D.I.C

5.1 Plan & Prioritize

A specific engagement that was geared towards improving productivity was identified for this model implementation. The process areas to be covered as part of the assessment were carefully chosen. Data from process implementation and compliance group was collected for this engagement. A questionnaire was formulated to conduct the assessments on the chosen practice areas like estimation, project management, requirements & change management, risk management and testing practices. An assessment scoring guideline was created to rate the practices appropriately.

5.2 Assess & Analyze

The assessments on selected projects within the engagement were conducted with respect to the practices as mentioned in earlier section. Each of the practice area with its sub-criteria was evaluated by a team of assessors through detailed discussions with project teams. Findings were collated and mapped on a radar chart for each of the practice area.

![Fig. 9 Assessment Scoring Guideline](image)

![Fig. 10 Case Study – Assessment rating for Testing Practices](image)

![Fig. 11 Case Study – Assessment rating for Requirements Practices](image)

![Fig. 12 Metrics Analysis – Case Study](image)
The collated results from assessments and metrics analysis were correlated for each of the practice area in the manner given below.

5.3 Correlation between Review & Defect Removal

![Graph showing correlation between Review Effectiveness and Defect Removal Efficiency](image)

**Fig. 17 Case Study – Correlation Defect Removal & Review Effectiveness**

5.4 Define & Describe - Case Study Results

Some of the salient inferences that were drawn from these outcomes are:

1. The lower performance levels of DRE (Defect Removal Efficiency) can be attributed to lower levels of RE (Review Effectiveness)
2. Testing practice area scores that average between 30-40% need closer scrutiny and improvements
3. Review mechanism for test cases, traceability to requirements, overall test strategy can be improved

5.5 Define & Describe-Recommendations

Key recommendations as given below were suggested for implementation in the engagement.

1. Review process to be tightened across project and deliverables.
2. Reusability of test cases to be implemented
3. Value stream analysis for identification of process wastages - Light weight process for small projects (less than 100 person-days) to reduce overhead efforts
4. EVM deployment in projects

5.6 Implement & Improve-Recommendations

A few key improvement initiatives were launched in the engagement. With adequate emphasis on productivity measurement, base-lining throughout life cycle, adhering to the recommendations that have been suggested and complying with organizational process and prediction models, the engagement has been able to bring in significant improvement in their practices.
5.7 Control & Conform – Measurement and Tracking

Some of the key improvement initiatives that were launched in the engagement:

1. Accurate collection and categorization of effort for engineering & non engineering activities, ad hoc activities, change requests, wait-time, support activities.

2. Monthly accurate reporting of key metrics (Effort/Schedule variance, RE, DRE, COQ (Cost of Quality), Defect Density) by every project.

3. Re-sizing the project (due to change requests).

4. Calculating actual productivity based on actual effort spent

5. Root Cause Analysis for deviations

6. Reporting key metrics for the project including productivity

6 Conclusions

There has been significant improvement in productivity levels within the engagement and its projects. Quantified measurement of the same is in progress. We are hopeful to bring in ~10 - 15% of improvement by enhancing the maturity level of their practices.

A few challenges and learning from this exercise have been the dire need for identifying and establishing contractual commitments towards productivity & measurement mechanisms early in the life cycle, driving of improvement initiatives as part of project execution, deployment of strong governance and reporting measures and last but not the least an open mindset by the relevant communities.

This framework has thereafter been successfully implemented in other engagements as well. The feedback from the improvement initiatives is continuously being incorporated into P.A.D.I.C framework and thereby bringing in higher maturity to engineering practices across the organization.

7 References


[7] Session on ‘Quality Vs Innovation’ at NASSCOM Summit, 6th & 7th Sep 2007, Bangalore, India


[16] Shewhart, W A. Statistical Method from the Viewpoint of Quality Control, Department of Agriculture, 1939
SESSION

TESTING AND VALIDATION

Chair(s)

TBA
An Integrated Test Environment Process Model to Control Software Failures

Mohan Bheemasenarao
Trans Atlantic Systems, Inc
333 North Ave, 23B, Secane, PA, 19018

M.H. Samadzadeh
Computer Science Department, Oklahoma State University, Stillwater, OK, 74078

Abstract - Software is important to virtually any business in today’s world. The most challenging issue that the software industry is facing today is how to produce quality software consistently. When there are multiple applications involved in a project in an integrated environment, controlling software quality becomes more complex. Even though many methodologies have been suggested to address the issue of software quality, there is always a demand for new and innovative software testing processes and methodologies in the face of changing technologies. The proposed model, an Integrated Test Environment Process Model (ITEPM) includes effective software practices that were developed based on practical industry data. This model was tested for efficacy, applicability, viability and practicability. A software development process that utilizes the proposed model will be able to control the software failures in an Integrated Test Environment and improve the software quality.

1. Introduction

1.1 Importance of Software and Quality

Production-quality software is an indispensable technology and has become a powerful business asset. It has zero reproduction cost, can be distributed worldwide in seconds, does not wear out or deteriorate, unless it undergoes modification and is the most economical and flexible way to implement almost any complex function [Humphrey 02].

The most challenging aspect that the software industry is facing today is how to deliver defect-free quality software. Software quality is defined as conformance to explicitly-stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software [Pressman 05].

1.2 Importance of Software Testing

Software testing is a critical part of the software development process and impacts the delivery of high quality software [Eickelmann and Richardson 96]. The Standish Group, an IT consulting firm, reported that 65 percent of software projects started in 2006 were failures, meaning they were either not completed on time, on budget, or met user requirements. This included 19 percent of the projects that were outright failures, meaning that they were abandoned without implementation [Rubinstein 07]. Failing to understand and manage risks can lead to project failure, a costly problem that has not been completely addressed in almost three decades since such outcomes were first described in the literature [Wallace and Keil 04].

2. Integrated Test Environment (ITE)

An Integrated Test Environment validates functionality across multiple systems. ITEs provide a means to integrate different applications from different software technologies and provides for the end-to-end flow of a product. When testing involves an Integrated Test Environment, it becomes more complex and challenging to control the quality of the resulting software. The input runs through the different applications in the ITE and undergoes various stages of validations in each application before coming out as the expected defect-free end-product. Since ITEs involve many major applications in the flow, failure of one application may impact the entire chain resulting in revenue losses, taking more time to analyze the failures, and hampering the growth of the business. So it is critical for businesses to address these failures.

A number of applications may work well individually but when they are put together in an ITE, the end-product might fail. Data can be lost across an interface, one module can have an inadvertent adverse affect on another, sub-functions when combined may not produce the desired major functions, individually acceptable imprecision may be magnified to unacceptable levels, global data structures can present problems etc. [Pressman05].
Studies strongly suggest that formal project management practices have the power to reduce software project risks. The value of such practices lies largely in the well-defined patterns and directives they create for coordinating interactions and integrating inputs from various project constituents. Formal milestones also help in monitoring progress and spotting discrepancies throughout the project trajectory [Tiwana and Keil 04].

3. Reasons for Software Projects Failures in an ITE

This section discusses the various reasons for software project failures in an ITE namely planning failures, requirement failures, process failures, technology failures, management failures, and uncontrolled factors which are described in the following sections.

3.1 Planning Failures

Planning is an important phase in software projects that defines various tasks involved in the project and outlines the project schedule. The project plan is developed at the beginning of a software development undertaking and is continually refined and improved as the project progresses. It can be useful to the management as a framework for rigorous review and it can play a significant role in the process of developing quality software. The following three items constitute the common planning failures in an ITE.

Unrealistic Schedules [Humphrey 02]: Unrealistic release schedules without any rational plan are bound to fail and will result in defective software. This may be due to factors such as poor project planning, underestimation of tasks, ineffective project head, and poor communication.

Out-of-Sync Release Schedules and Code Delivery Dates: Testing process will have to wait until all applications become available, resulting in missing the schedules.

No Dedicated ITE: Some applications, though required in an ITE, are not even in the ITE since they do not have a test environment. These applications are tested independently but not as a part of the ITE. This could be risky in the production phase.

3.2 Requirement Failures

To begin any software project, it is essential to have a set of Business Requirements. Business Requirements are transformed into Business Related Documents (BRD) based on which the System Related Documents (SRD) is developed. These documents must be finalized and accepted by all applications in an ITE. The following four items constitute the common requirement failures in an ITE.

Misunderstanding of Requirements [Wallace and Keil 04]: Requirements define what the product should be and how it should perform. Thus, a clear understanding of the requirements could be responsible for producing a good product outcome, even as problems with process outcome persist.

Incorrect Requirements: Giving incorrect requirements to the applications will result in defective products delivery to the market and can adversely impact business prospects.

Frequent Requirement Changes: Such changes impact the flow across an ITE since applications not impacted by requirement changes have to wait until applications impacted by these are modified to complete their implementation, resulting in delays in the final implementation across the ITE. While the requirements or objectives normally change during the early phases, there is a point beyond which changes will do more harm than good [Humphrey 02].

Requirements Conflict Among Applications: Requirements for one application may conflict with the functionality of some other application in an ITE, thus causing failures.

3.3 Process Failures

The end-to-end testing process has to be finalized and accepted by all applications in an ITE. Individual applications have to make sure that this will not conflict with their own testing process. If the end-to-end testing process is not followed, it may result in the following failures.

Poor Quality: The poor quality work of one application may result in failure across the entire ITE.

Data Configuration Mismatch across ITE: Given two communicating applications, data found in one application is not found in the other application, thus resulting in a failure.

Data Sharing Issues: Many testing groups share the data generated from one instance of an application in an ITE thus causing duplicate, redundant, and conflicting data issues across the ITE.

Lack of Progress Tracking System: This could be critical in any software project. When there is no tracking system to monitor the progress of each application and mitigation strategies to avoid failures, the projects will generally fail.

3.4 Technology Failures

Applications in an ITE may be operating on different software technologies due to various business needs. It is essential to have proper software interfaces clearly defined to avoid the following failures.

Software Technology Configuration Mismatch across ITE: Some applications running on the latest software technologies and some applications running on old software technologies may result in complex interface issues.

Instance Issues across ITE: Some applications in an ITE operate on multiple instances whereas other applications operate on a single instance in an ITE, thus causing failures.
3.5 Management Failures

Top management controls all applications in an ITE. Top management should clearly define the objectives, the scope of operation, and the goals of all applications in an ITE. What follows are some of the management failures in an ITE.

Inappropriate Staffing [Humphrey 02]: When management fails to provide timely, adequate, and properly trained resources for a project, the projects will generally fail.

Lack of Coordination among Applications in an ITE: This may be due to different applications in an ITE coming under different departments in the organization, different regions with varying working culture, etc.

Lack of Top Management Support for the Project [Wallace and Keil 04]: Management support for a project is essential in terms of providing the required budget, resources, motivation, and encouragement.

3.6 Uncontrolled Factors

Software projects may fail due to many uncontrolled factors including factors such as turn-over of the skilled people, organizational restructuring resulting in the shifting of people, a product becoming obsolete, the software technology utilized becoming obsolete, organizational changes due to acquisitions resulting in business changes, changes in business due to governmental regulations, and so on. Since these factors are difficult to predict, top management must have mitigation strategies in place to counter these factors in the event they occur.

4. Design and Development of an Integrated Test Environment Process Model (ITEPM)

An Integrated Test Environment Process Model (ITEPM) consists of models such as planning model, requirements model, process model, technology model, management model, and team model to address various failures in an ITE. These models establish a technical and management framework for applying methods, tools, and people to the software development task. Also, these models identify roles, specify tasks, establish measures, and provide input and output criteria for the major failures impacted. Well-defined process models help in identifying the problem areas and suggest various methods for improvement [Humphrey 05a]. When new or improved processes are developed, these models can be reused or modified and incorporated into the process models. These models provide a solid foundation for software process improvement in an ITE and help to improve software quality.

<table>
<thead>
<tr>
<th>Planning Model</th>
<th>TEMPLATE FOR PLANNING MODEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Model to prevent planning failures in an ITE</td>
</tr>
<tr>
<td>Input criteria</td>
<td></td>
</tr>
<tr>
<td>Input source</td>
<td>Planning failures in all applications in an Integrated Test Environment</td>
</tr>
<tr>
<td>Factors</td>
<td></td>
</tr>
<tr>
<td>Release schedule</td>
<td></td>
</tr>
<tr>
<td>Capacity planning and tracking</td>
<td></td>
</tr>
<tr>
<td>Level of effort</td>
<td></td>
</tr>
<tr>
<td>Release listing report</td>
<td></td>
</tr>
<tr>
<td>Release development plan</td>
<td></td>
</tr>
<tr>
<td>Status reporting</td>
<td></td>
</tr>
<tr>
<td>High level risk assessment</td>
<td></td>
</tr>
<tr>
<td>Environment setup plan</td>
<td></td>
</tr>
<tr>
<td>Output criteria</td>
<td>Release schedules of all applications in an ITE is documented.</td>
</tr>
<tr>
<td></td>
<td>Capacity model completed.</td>
</tr>
<tr>
<td></td>
<td>Level of effort estimation completed.</td>
</tr>
<tr>
<td></td>
<td>Release listing report ready for distribution.</td>
</tr>
<tr>
<td></td>
<td>Release development plan ready for distribution.</td>
</tr>
<tr>
<td></td>
<td>Milestone dates reported.</td>
</tr>
<tr>
<td></td>
<td>Risk assessment review completed.</td>
</tr>
<tr>
<td></td>
<td>Environment setup plan completed.</td>
</tr>
</tbody>
</table>

Planning Model Factors: The factors considered in the planning model are described in this subsection.

Release Schedule: Release management develops a schedule for an ITE testing project. The schedule is discussed with all impacted applications in an ITE to ensure alignment with the delivery of the release to production in support of the end-to-end testing process.

Capacity Planning and Tracking: Capacity of resources is tracked throughout an ITE testing lifecycle. Capacity planning includes manpower planning and any software and hardware tools required for the project.

Level of Effort (LOE): Level of effort is an estimation of number of hours required to complete a software testing lifecycle in an ITE. All applications must use their Level of Effort model process to estimate the time required for testing applications in an ITE. To estimate the number of hours required to test an application, the complexity of the testing must be determined.

Release Listing Report: This report lists the number of releases of all applications in an ITE throughout the year.

Release Development Plan (RDP): This plan is developed for each release by all applications in an ITE. It includes application-specific plans, schedules, resources, status, risks, and the software quality assurance functions [Humphrey 05b].

Status Reporting: Status reports help in periodically reviewing the status of an ITE project and assess progress. Information reviewed at the meetings is captured in the various status reports, ranging from the application level to the ITE level tracked by release management.
High Level Risk Assessment: The purpose of risk management during the planning phase is to identify any possible events that would have adverse effect on the successful completion of the software project [Tiwana and Keil 04]. Risk assessment has to be done at the application level as well as at the ITE level.

Environment Setup Plan: This plan includes all applications impacted by the release including hardware, software, servers, and network configuration. The plan may also contain the availability of all applications impacted in an ITE.

### Requirements Model

**TEMPLATE FOR REQUIREMENTS MODEL**

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Input criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model to prevent requirement failures in an ITE</td>
<td>List of common failures in an ITE.  Misunderstanding of requirements. Incorrect requirements. Frequent requirement changes. Requirements conflict among applications.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input source</th>
<th>Output criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements completed.</td>
<td>Requirements of all applications in an ITE received. Requirements reviews completed. Candidate listing finalized. Requirements analysis completed. Requirements modifications documented. Matrix to trace the requirements completed.</td>
</tr>
</tbody>
</table>

**Factors**

- Receiving requirements
- Reviewing requirements
- Candidate listing
- Analyzing requirements
- Requirement modifications

**Process Model**

**TEMPLATE FOR PROCESS MODEL**

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Input criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model to prevent process failures in ITE</td>
<td>List of common process failures in an ITE.  Poor quality. Data configuration mismatch. Data sharing issues Lack of tracking system.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input source</th>
<th>Output criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process failures in all applications in an ITE</td>
<td>ITE testing procedure documented. Readiness of ITE is checked. Data setup for applications in ITE is setup. Tracking system established. Quality policy is setup.</td>
</tr>
</tbody>
</table>

**Factors**

- ITE test plan and test cases
- Application readiness
- Interface readiness
- Data setup
- Test execution
- Verification and validation
- Software quality assurance
- Reviews and audits
- Quality assurance reporting
- Quality assurance training

Requirements Model Factors: The factors considered in the requirements model are described in this subsection.

Receiving Requirements: A work request (WR) is a business client’s request for development of a new product or the client’s requested modifications to an existing product. A work request is composed of requirements that define the development and testing work to be done by the organization. Complete and accurate requirements are to be collected before starting the work.

Reviewing Requirements: The release management group reviews all work requests and assigns a date by which all applications in the ITE must perform an impact assessment. The assessment will determine whether or not the new work requests impact the different applications that exist in an ITE. After it is determined that the application is impacted by a work request, a separate project profile is created. The order of magnitude (OOM) [Murthy 07], which estimates the needed capacity to complete the work request based on the information available in the project profile, is recorded in the LOE.

Candidate List: A candidate list basically contains the number of ITE work requests approved by the release management group after prioritization. The candidate list may be distributed to the impacted ITE application groups.

If there are changes in the requirements, the candidate list may have to be re-evaluated. All the application groups impacted by the changes may have to be notified of the changes in the requirements and changes in the candidate list.

Requirement Analysis: The detailed client requirements based on the candidate list are captured and may be stored in a requirement database. The key business objectives, business function, client traceability, dependencies among application groups, and contact information may be recorded in the requirements database. A team meeting has to be conducted with all application groups in the ITE to analyze the requirements documents.

Requirement Modifications: Whenever a client submits changes to the requirements, the ITE testing team will be notified and the impact to the ITE testing schedule needs to be analyzed. The modifications and the impacted applications are reviewed by the ITE testing team. Once the applications concur with the modifications, the requirements documents are updated.

Requirement Traceability Matrix: Business requirements are transformed into Business Related Document (BRD), based on which the System Related Document (SRD) is developed. The application groups use the SRD and the general design document to create the detailed design in order to ensure traceability to the original work request. The traceability matrix is utilized by the test teams to map test cases and the test results to the requirements.

Process Model Factors: The process model factors are described in this subsection.
ITE test plans and test cases: The ITE test plan containing the details of the testing process, test approach, and test methodology has to be designed as a part of the ITE testing process.

Application Readiness: All applications in an ITE have been tested in conformance to the system requirements and kept ready before the integrated testing is started. Application tests ensure that the proper changes were made to the code.

Interface Readiness: Interface readiness can be checked by doing a shake-out test which ensures the connectivity of all applications in an ITE. This test will ensure that all applications are working properly before the actual integration testing begins.

Data Setup: During the test data setup phase, the test team, in coordination with the ITE applications, has to make sure that the required data to carry out the integrated testing is made available to all applications. Also, if multiple applications share a common data, the data configuration checks have to done to avoid data mismatch across ITE systems.

Test Execution: ITE test execution includes regression testing and progression testing. Regression test is conducted to ensure that the previous code works with the new changes, and that the new functionality does not break the procedures related to the existing functionality. Progression testing is the testing of new functionalities with the new software.

Verification and Validation [Pressman 05]: Verification in an ITE involves verifying the test results for each requirement with respect to the corresponding system related requirement to ensure that we are building the product right. Validation in an ITE involves verifying the test results for each requirement with respect to the corresponding business related requirement to ensure that we are building the right product.

Test Status Tracking: A test status tracking mechanism has to be established to track the test results and also for reporting purposes. A suitable software test reporting tool may be used for this purpose. Status review meetings may also be conducted as part of progress tracking with other applications in an ITE.

Software Quality Assurance (SQA): The purpose of software quality assurance is to provide visibility to practices and processes followed by all application teams in an ITE and to ensure that the key deliverables and design artifacts meet the defined standards [Hower 07]. The SQA team performs a role in establishing and improving a well-defined process for delivering high-quality software products. The SQA process is essential in planning and performing each activity associated with the software testing work request, application, or release. [Prasad 04].

Reviews and Audits: The SQA manager ensures that the SQA process reviews of different applications are scheduled and conducted. The audits are conducted to determine whether the procedures and standards set by the organization are being met in the development of a specific product.

Quality Assurance Reporting: The SQA manager collects and reports the data regarding process compliance and cost in hours, schedule, and status of the SQA activities.

Quality Assurance Training: The skills and knowledge needed by each member of the organization to perform their functions are identified and any skill or knowledge gaps are addressed through a training plan.

### Technology Model

| Technology Model |
|------------------|---------------------------------------------------------------|
| **TEMLPAE FOR TECHNOLOGY MODEL** |
| **Purpose**   | Model to prevent Technology Failures in an ITE |
| **Input criteria** | List of common failures in an ITE |
| | Software technology configuration mismatch across ITE. |
| | Instances issues across ITE. |
| **Input source** | Technology failures in all applications in an Integrated Test Environment |
| **Factors**        | Name and location |
|                     | Ownership and contact details |
|                     | Database server specifications |
|                     | Application server specifications |
|                     | Mainframe specifications |
|                     | Instance details |
|                     | Input type |
|                     | Source and destination |
|                     | Response times |
|                     | Access details |
| **Output Criteria** | All applications in an ITE configured. |
|                     | All testing instances documented. |
|                     | Technical specifications of all applications documented. |

Technology Model Factors: The factors considered in the technology model are described in this subsection.

Name and Location: The Technology model should have all application names and their address location.

Ownership and Contact Details: The technology model should have details such as application owner information and contact information.

Specifications: The technology model should have database server specifications, application server specifications, and mainframe specifications. The specifications should include details such as dbms type, operating system details, memory size, host name, and program size.

Instance Details: There may be several different testing instances for different user groups in an organization. The technology model should have details of all applications instances required in an ITE such as production instance, test instance, user acceptance testing instance (UAT), and staging test environment (STE) instance.

Input Type: The input file received by different applications may be in different formats depending upon the type of technology they use. The technology model should list the type of input files coming into the system.
Source and Destination: The technology model should list the source application sending the data and the destination application where data is being transmitted.

Response Times: The most common fall-out noticed in an ITE is the timing issue because of the different response times taken by different applications in an ITE. So, the technology model should have details about the typical response times of all applications for both best case and worst case scenarios.

Application Access Details: Details of how the application should be accessed with userid, password, privileges, and permissions, etc. should be included. Special software tools may be required in some cases to access the systems.

### Management Model

**TEMPLAE FOR MANAGEMENT MODEL**

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Model to prevent management failures in an ITE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input criteria</td>
<td>List of common failures in an ITE. Inappropriate staffing. Lack of coordination among applications in an ITE. Lack of top management support for the project.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input source</th>
<th>Management failures in all applications in an Integrated Test Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factors</td>
<td>Project management</td>
</tr>
<tr>
<td></td>
<td>Financial management</td>
</tr>
<tr>
<td></td>
<td>Development management</td>
</tr>
<tr>
<td></td>
<td>Test management</td>
</tr>
<tr>
<td></td>
<td>Team management</td>
</tr>
<tr>
<td></td>
<td>Configuration management</td>
</tr>
<tr>
<td></td>
<td>Implementation management</td>
</tr>
<tr>
<td></td>
<td>Release management</td>
</tr>
<tr>
<td></td>
<td>Support management</td>
</tr>
<tr>
<td></td>
<td>Environment</td>
</tr>
</tbody>
</table>

| Output criteria | Adequate staffing for the project is finalized. Coordinated testing process is documented. Top management support is ensured. |

**ITEPM Team Model**

**TEMPLAE FOR ITEPM TEAM MODEL**

<table>
<thead>
<tr>
<th>Purpose</th>
<th>To build an ITEPM team</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input criteria</td>
<td>Goals Strategy Roles Team Plans Reviews</td>
</tr>
<tr>
<td>Input source</td>
<td>ITEPM team failures in all application teams in an ITE</td>
</tr>
<tr>
<td>Factors</td>
<td></td>
</tr>
<tr>
<td>Goals</td>
<td></td>
</tr>
<tr>
<td>Define strategy and plans</td>
<td></td>
</tr>
<tr>
<td>Define roles</td>
<td></td>
</tr>
<tr>
<td>Team reviews and meetings</td>
<td></td>
</tr>
</tbody>
</table>

| Output criteria | Team roles, goals, processes, and responsibilities are defined. Well-defined ITEPM team is established. |

Management Model Factors: The factors considered in the management model are described in this subsection.

Project Management: Project management is responsible for launching a new project, decides the goals with the marketing department, and coordinates with the release department. They work with the management team to select team members for the project.

Financial Management: Financial management is responsible for budget planning for the organization. It includes hardware capacity planning, technical support, disaster recovery planning, and overall resource management.

Development Management: They are responsible for all development activities that are involved in the development of the project. They manage the development team.

Test Management: Test management is responsible for test-related activities that include preparation of test plans, test procedures, and test cases which meet the test standards.

Team Management: Team management is responsible for setting goals for the team. They are responsible for defining strategies, tasks, and plans for the team.

Configuration Management (CM) [Pressman 05]: They are responsible for the overall configuration management activities, and prepare the configuration management plan.

Implementation Management: The responsibilities of the implementation management are to produce a high-quality product, and to ensure that the implementation is fully confirmed to the design.

Support Management: They are responsible for helping the team to use proper tools and methods, and handle the team’s configuration management and change control functions.

Release Management: They are responsible for the overall management and coordination of the releases.

Environments: Environments are locations where the latest versions of the code are stored.

Team Model Factors: The team model factors are described in this subsection.

Goals: The members of the teams are committed to a common set of goals to maintain the team’s motivation and energy. The team agrees on the goals during project launch. All members of the team know their individual goals and understand where they stand against these goals. The team goals should be measured frequently.

Strategy and plans: The plans and strategies should be clearly defined for achieving the goals which are set. The team should have a strategy for reviewing every module before code inspections. The team should have set plans to test each and every module created.

Team Roles [Humphery 05b]: Team roles are tasks performed by members of the team. The tasks are assigned based on the individual skill sets. The roles of the team members are defined before the project launch.

Team Reviews and Meetings: The team progress will be reviewed at every stage of the software process. All documents and modules created are reviewed. If any
problems are found at any stage, recommendations should be made to rectify the errors found.


<table>
<thead>
<tr>
<th>Test Plan Template for Planning Model</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TEST PLAN FOR PLANNING MODEL</strong></td>
</tr>
<tr>
<td>Planning failures</td>
</tr>
<tr>
<td>Unrealistic schedules</td>
</tr>
<tr>
<td>TP2: Resource capacity verification</td>
</tr>
<tr>
<td>TP3: Compare capacity model with LOE</td>
</tr>
<tr>
<td>Out-of-sync release schedules</td>
</tr>
<tr>
<td>TP5: Risk analysis checking</td>
</tr>
<tr>
<td>No dedicated ITE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test Case Template for Planning Model</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TEST CASE FOR PLANNING MODEL</strong></td>
</tr>
<tr>
<td>Test case title</td>
</tr>
<tr>
<td>Test plan reference</td>
</tr>
<tr>
<td>Failure</td>
</tr>
<tr>
<td>Test environment</td>
</tr>
<tr>
<td>Application name</td>
</tr>
<tr>
<td>Test instance name</td>
</tr>
<tr>
<td>Test data source</td>
</tr>
<tr>
<td>Execution method</td>
</tr>
<tr>
<td>Execution steps</td>
</tr>
<tr>
<td>• Check the release schedules of all applications</td>
</tr>
<tr>
<td>• Check the ITE release schedule</td>
</tr>
<tr>
<td>• Check if the release date for an ITE are earlier than ITE release date</td>
</tr>
<tr>
<td>Expected results</td>
</tr>
<tr>
<td>Test results</td>
</tr>
</tbody>
</table>

Test plan and test case templates for other models can be similarly derived.

6. Summary and Future Work

6.1 Summary

It is widely known that creating good production-quality software is quite challenging and the software industry must use the best available methods to improve quality. A software development process that utilizes the proposed model should be able to manage if not control the occurrence of software failures in an Integrated Test Environment and help to produce production-quality software consistently.

6.2 Future Work

For future work, the factors listed in the planning model, requirements model, process model, technology model, management model, and ITPEM team model can be further broken down into individual models to cover more failure points than listed in this work. For each application in an Integrated Test Environment an Application Process Model can be developed similar to the Integrated Test Environment Process Model proposed in this work. More test cases and desk checks can be considered for a more thorough testing of the model.

REFERENCES


Empirical Efficiency Testing for OSS

Dae-Woo Kim1, Hyun-Min Lim1, Jae-Hyoung Yoo1, and Sang-Ha Kim2
1Network Technology Lab., Korea Telecom, Daejeon, Korea
2Department of Computer Science, Chungnam National University, Daejeon, Korea

Abstract - This paper describes efficiency testing for KT-OSS (Korea Telecom Operations Support System), which is called NeOSS (New Operations Support System). Since NeOSS is a large software and telecommunications operations support system, it is essential that efficiency tests, including tests for performance, load, stress, and stability, be performed before it is operated in the field. In addition to the functionality of the system, its performance is also important, because if the system has problems and suffers from degradation in performance, it will not be able to be operated normally, and may not run as a result. Therefore, before the developed system is released, there must be sufficient testing of its efficiency. In this paper, we describe our experience of the testing of the NeOSS, the tests related to the efficiency testing, test organization, test-bed, test procedures, and test case. Through this efficiency testing, we were able to guarantee the performance of the NeOSS.

Keywords: Operation Support System, Efficiency Testing, Test-Bed, Test Phases

1 Introduction

If the defects and problems of a system are only found after the developed system is released in the field, they could seriously affect the business related to the system. In particular, a poorly performing system may be incapable of being operated normally, or may shut down when many users are using it. Faults such as these could result in the failure of the OSS development project. As such, it is important to ensure performance quality in the development and maintenance of the KT-OSS, which is called NeOSS (New Operations Support System).

In general, the performance quality characteristic is shown in the ISO/IEC 9126 [1,2] as an efficiency characteristic. To ensure the efficiency in the development and maintenance of NeOSS, we performed efficiency testing. This included performance, load, stress, and stability tests of the NeOSS. This paper describes the empirical efficiency testing activities related to the performance of the NeOSS that were undertaken to ensure the successful development and maintenance of the NeOSS. Here, we show test phases, test-bed, and criteria for tests and experiences for the testing activities. Through this efficiency testing, we were able to successfully develop and maintain the NeOSS.

The rest of this paper is organized as follows: Section 2 gives a brief overview of the KT-OSS. Section 3 describes the testing activities for the KT-OSS. Section 4 describes the efficiency testing. Section 5 describes a case study of the efficiency testing of NeOSS. Finally, we conclude our work and outline some work to be undertaken in the future in Section 6.

2 Overview of the NeOSS

The NeOSS is KT’s integrated operations support system. Its main functions are service configuration, service assurance, service quality and operation information management, facility management, network management (IP (Internet Protocol), ATM (Asynchronous Transfer Mode) and etc.) and others. The services related to these include PSTN (Public Switched Telephone Network), ADSL (Asymmetric Digital Subscriber Line), VDSL (Very-high-bit-rate DSL), Wireless Internet service and others.

The core technologies of the NeOSS include the application architecture and the business process based on CBD (Component Based Development), multi-layered application architecture, common development platform (MS (Microsoft),Net), EAI (Enterprise Application Integration) for workflow and message bus, common data code, single sign-on and others.
Figure 1 shows the architecture of the NeOSS, depicting the manner in which the sub–systems are connected through the EAI information bus. The sub-systems are SO (Service Ordering), SA (Service Assurance), FM (Facility Management), ADM (Access Domain Management), WM (Workforce Management), SLA (Service Level Agreement), and NM (Network Management). In addition, according to the NeOSS development plan, the network quality information system and others are developed and new services are added to the NeOSS.

3 Testing Activities for KT-OSS

3.1 Phases of Testing for NeOSS Development

<table>
<thead>
<tr>
<th>Phases</th>
<th>Description</th>
<th>Type of testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code review</td>
<td>Testing for compliance with development standards, and of the grammar of the program and module</td>
<td>White box</td>
</tr>
<tr>
<td>Unit test</td>
<td>Testing of the respective sub-systems</td>
<td>Black box</td>
</tr>
<tr>
<td>Integration test</td>
<td>Testing of the integrated sub-systems by service</td>
<td>Black box</td>
</tr>
<tr>
<td>Acceptance test</td>
<td>Testing by the operations group and users for acceptance of the NeOSS</td>
<td>Black box</td>
</tr>
<tr>
<td>Operation test</td>
<td>Testing for NeOSS operation by users before applying it in the field (including monitoring of performance)</td>
<td>Black box</td>
</tr>
</tbody>
</table>

The code review in Figure 2 was conducted when the NeOSS was being developed. In particular, the operation testing was conducted to validate suitability, performance and stability from the perspective of the users who will be using the new system in practice.

The purposes of the operational testing were as follows:

- Checking the functionality of the NeOSS by using system functions such as handling orders from services, failure reports, and others.
- Performance monitoring by checking performance metrics such as server CPU utilization, memory usage, number of Web requests per second, and others during a specified period

- Analysis of the support activities of the NeOSS technical support team and the help desk for the NeOSS (this is in preparation for supporting the operation of the NeOSS).
- Obtaining the opinions of users regarding the NeOSS

3.2 Test-Bed

To simplify the development and maintenance of the NeOSS, we made a test-bed [3] for the NeOSS, which included a development environment and two test environments to manage and control the quality of the software system. The configuration of the test-bed is shown in Figure 3. The Development I environment enables the developers to freely develop and conduct unit testing of the various sub-systems of the NeOSS. Development II is for the next version of the NeOSS, which is substantially different from the operational version of the NeOSS. However, the Development II environment is optional, and is only made when it is needed in the course of development. Test I of the two test environments provides a test environment for the performance of unit testing and integration testing by the
developers, with the version controlled by a version manager. Test-Staging provides a test environment for the performance of unit testing, integration testing and efficiency testing by the testers from the test & evaluation department, the users, and the operators, under strict version control by the version manager. This environment is called Test-Staging, because the NeOSS version and the hardware configuration are similar to those in the operational environment, and we performed the final testing before the field release. Incidentally, Test-Staging represents 8% of the real operational environment.

### 3.3 Test Steps & Organization

![Figure 4. Test Steps & Organization](image)

The test steps and the related organization are shown in Figure 4. The organizations involved in the testing included the development department, the testing & evaluation department, the operations department, and the users. To rigorously manage and control the quality of the NeOSS, the testing & evaluation department was created to be independent from the development department. Testing was performed through the following sequential steps:

1. In the first step, the development department performs its own unit testing and integration testing in Test 1. If defects are found during these tests, the developer can fix the defects freely. In the second step, the testing & evaluation department performs the unit testing, integration testing, efficiency testing, and regression testing in the Test-Staging. If defects are found during the second step, the testers inform the developers. After these defects have been fixed by the developers, the testing for defects is conducted again. In the third step, the testers are the operator of the operations department and the representative users of the sub-systems. These testers perform the unit testing and integration testing to verify and validate the system, as the acceptance testing in the Test-Staging.

### 4 Efficiency Testing

#### 4.1 Efficiency Testing and its Criteria

Efficiency testing includes the performance test, load test, stress test, and stability test. The testing of the respective sub-systems was performed by the load-generating tool. The load of the efficiency testing represents about 8% of the load in the operating environment, because the scale of the Test-staging is about 8% of the operating environment. The types of efficiency testing performed in the development and maintenance of NeOSS are as follows;

- **Performance testing:** This test is used to check response time and processing time, and is performed at a load of 1% ~ 3% of all users as the number of simultaneous users. For reference, the average load in the operating environment is about 2% of all users as the number of simultaneous users.
- **Load testing:** This is to check whether or not the system is stable when it is being used by the maximum number of simultaneous users. This has the same criteria as the performance testing in overload, which is about 3% of all users as the number of simultaneous users.
- **Stress testing:** This is to check whether or not the system is stable under the specified overloading. The criterion is the system’s stability at more than twice the peak load, which it is increased to gradually over a specified period of time.
- **Stability testing:** This is to check whether or not the system is stable under specific overloading during a specified period of time, and to verify the system’s stability under peak load conditions for more than six hours.

Table 2 shows the criteria of the four types of efficiency testing.

<table>
<thead>
<tr>
<th>Efficiency Testing</th>
<th>Metrics</th>
<th>Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Performance test (at average load)</strong></td>
<td>Response Time for web and client operation (in query functions)</td>
<td>Within 3 sec.</td>
</tr>
<tr>
<td><strong>Note</strong> Average load: 1 ~ 2% of all users as the number of simultaneous users</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Load testing</strong></td>
<td>Number of occurrences of a delay greater than one second for query operation (in Database query)</td>
<td></td>
</tr>
<tr>
<td><strong>Stress testing</strong></td>
<td>Number of more than 10,000 page reads (in Database query)</td>
<td>0</td>
</tr>
</tbody>
</table>
### 4.2 Efficiency Testing Procedure

- **Selecting Test Cases**: The test cases for efficiency testing are determined by selecting the core test cases from the test cases of the functionality testing.
- **Making the scripts**: We create the scripts for the test cases, which are executed by the load generation tool.
- **Load calculation**: We calculate the load of the test for the respective sub-systems of the NeOSS. The load is the number of simultaneous users. The formula used is the number of simultaneous users of the respective sub-systems, multiplied by the scale of the Test-staging.
- **Performance testing**: This test is performed at a load of 1% ~ 2% of all users as the number of simultaneous users. For reference, the average load in the operating environment is about 2% of all users as the number of simultaneous users.
- **Load testing**: This test is performed in overload, which is about 3% of all users as the number of simultaneous users.
- **Stress testing**: This test is performed at more than twice the peak load, which is increased to gradually over a specified period of time.
- **Stability testing**: This is performed under conditions of peak load for more than six hours.
- **Result Analysis**: Through the efficiency testing, we measure the response time, CPU utilization, memory usage, and amount of memory leaks by using the tools, the performance log data obtained through Windows Server 2003, and the profiler’s log in the DB server. We then analyze the results of the respective tests.
- **Report feedback**: The results of the efficiency testing are then reported to the developers and development partners.

### 4.3 Efficiency Testing Environment

The efficiency testing environment is shown in Figure 6. In the testing environment, we used two kinds of test tools for the efficiency testing, because the NeOSS has two kinds of clients, which are Microsoft’s WebForm and WinForm. In the WebForm client, we used Microsoft’s VSTS (Visual Studio 2005 Team System) [4], which consists of a VSTS client, a controller, a DB server, and agents. The DB server has the function of saving the test results, while the agents play the role of load generation. The VSTS supports regression testing and the performance testing. However, as the VSTS does not support the WinForm client, we used the LoadCube tool, made by a Korean developer. This tool was not easy to use, and the re-usability of the script was also very low. Therefore, in addition, we are taking account of the QALoad [5] from the Compuware.

Also, in the event that the added and modified functions simply query the DB(Database), the stored procedures related to the functions were tested as a part of efficiency testing, and their performances were monitored through the DB server’s profiler.

### 5 Case Study for Efficiency Testing

Details of the efficiency testing case study, such as CPU utilization, response time, Memory leaks, and DB queries per AP server are shown in Figures 7, 8, 9, and in Table 3. This testing was performed whenever a new service was added to the NeOSS. The load range for the performance test, load test, and stress test was from 1% to 10% of all of users as the number of simultaneous users in practice.
Table 3. Number of Stored Procedures for DB Query over 1 Second & over 10,000 Reads Page

<table>
<thead>
<tr>
<th>Subsystems</th>
<th># of Blocking</th>
<th>Load: 1 User</th>
<th>Load: Simultaneous Users (2%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Duration&gt;1,000ms</td>
<td>Reads&gt; 10,000 pages</td>
</tr>
<tr>
<td>FM</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>SA</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SLA</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>WM</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SN</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>WES</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
That is, we performed the testing under overloaded conditions rather than the testing conditions of Table 2 to determine what amount of overload endangers the health of the system. The stability testing was performed at a 3% load for a period of six hours. For reference, in the operating NeOSS, the average load is 2%, and the overload is 3%. The scale of load is 8% of the practical simultaneous users, because the scale of the testing environment is 8% of the practical operation environment.

As shown in the figures and in the table of results, we measured and analyzed the response time per query, the response time and the number of page reads per DB query, memory leaks, and CPU utilization per server as well as memory utilization, CPU utilization per service, and the number of service downs (which are not shown in this paper). Figure 7 shows the CPU utilization of the AP server under loads ranging from 1% to 10%. In Figure 8, the graph shows whether or not the sub systems meet the criterion for response time. For example, if ten functions related to the DB query of a sub system are tested for the response time under a load ranging from 1% to 10%, and all of the ten functions have a response time of three seconds or less, the success rate is 100%. The number per sub system in Table 3 represents the number of Stored Procedures that do not meet the criteria of the testing, and the number of blocked DB queries according to the loads. From these results, we know that the NeOSS-SLA had poor performance in terms of the response time and DB blocking, as well as poor performance in DB queries. However, in the stability testing, there were no memory leaks. The CPU utilization per AP (Application) server was normal because the measured value was below 50% in conditions of average load, as well as at overload. These results were relayed to the developers, who worked to improve the poor aspects of performance, after which the efficiency testing was conducted again.

6 Conclusions

In this paper, we looked at the testing phases, the testing steps, the testing organization, and the test-bed for KT-OSS. In addition, we described the process for the empirical efficiency testing of the NeOSS. As the aim of the efficiency testing is to ensure the performance quality of the NeOSS, we showed the tests, the criteria, the test environment, and all related test procedures. Through this testing, we were able to ensure the performance quality of the NeOSS, and predict the performance quality under the operating NeOSS.

However, the efficiency testing was performed on a limited number of functions of the NeOSS, because the scale of the Test-Staging environment was relatively small. This means that there could be potential risks for the performance of the NeOSS. Therefore, in the near future, we need to study a means by which efficiency testing can be performed without limitation on the scale.

7 References


A Service Oriented Architecture Complexity Metric, Based on Statistical Hypothesis Testing

Michael J. Maynard    George Dimitoglou
Department of Computer Science
Hood College
Frederick, MD 21701

Abstract—Service Oriented Architectures (SOA) is a widely used paradigm for the development and deployment of scaleable, loosely coupled distributed environments. Along with the flexibility and relative ease of deployment, such architectures have inherent complexities. These complexities impact the maintainability and testability of SOA implementations but no metrics exist to identify and express these complexities. This paper presents a first attempt towards developing a preliminary complexity metric, the balance factor, which is derived via statistical hypothesis testing and indicates if computing resources are uniformly distributed throughout a SOA-based environment.

Keywords: software architecture, web services, metrics, complexity

I. INTRODUCTION

Centralized system architectures tend to establish static, specific communication and data interchange channels between computing resources. This approach works well in small, static environments where tracking messages interfaces and computations are simpler. Centralization provides better control and harmonization of interactions. However, centralized architectures don’t scale. Distributed environments scale better but their difficulty to manage, control and maintain increase proportionally to the number and heterogeneity of system, components and platforms.

Service Oriented Architectures (SOAs) are inherently distributed and scalable, yet they are amenable to platform, programming language, data type and processing paradigm heterogeneity. The SOA approach is a relatively new software architecture paradigm to deploy, organize and control distributed resources. SOA has already gained widespread acceptance. It is widely used and industry trend studies predict an even more dominant presence for the future [2].

The core component of a SOA – the services (pieces of self-contained access to business logic functionality) – can be designed, developed and deployed by multiple groups, organizations and providers. Technically a service is an interface for various messages so it is viewed as an abstraction that conceals implementation details from the consumers [9]. Another component is the enterprise service bus (ESB), an underlying infrastructure that provides point-to-point, mediation and proxy-based connection interoperability between services and those that consume them.

The integration of these services and an ESB at a high level is trivial, allowing the loose coupling of resources in a distributed system architecture that evolves, often organically, with little or no overall control over size, scope and boundaries. As a result, SOA implementations may be described as a single concrete architecture but determining their boundaries and complexity is arduous, if not impossible. Not knowing the boundaries of a distributed environment obstructs the ability to identify bottlenecks and determine if the SOA is balanced in terms of resources and their consumption. This has direct implications with the manageability, maintainability and cost of the environment.

In software engineering, metrics exist to determine software size and complexity such as
source lines of code (SLOC), the cyclomatic complexity [12] and Halstead measurements. In networking, graph theory provides the mathematical foundations to assess the complexities of network topologies. At a more abstract level in algorithm analysis, metrics do exist to measure algorithm complexity based on asymptotics. Unfortunately, in SOA, no such metrics exist.

The main objective of this work is to introduce and develop a simple model to quantify the notion of balance in a SOA, aiming for metrics similar to the notions of soundness and confidence used in software engineering [3]. A more ambitious objective is to bring to light aspects of the SOA paradigm which will allow other quantitative measurements and ultimately, to be able to establish a single metric that would describe overall architectural complexity.

II. METHODOLOGY

A. Assumptions
To examine and analyze the notion of balance in a SOA certain assumptions are made about the typical SOA architecture. To ensure the generality and pervasiveness of the model we use the fundamental SOA as described by Krafzig et al [8]. In Figure 1, a sample of the fundamental SOA is presented. The major components of the architecture are identified either under the enterprise or the basic layer. The enterprise layer exposes the services to the enterprise and provides the front-end interfaces to service consumers. The basic layer contains all the components required to perform the required computations for the services to be rendered. This layer can be logically divided in two parts, each part from either side of the Enterprise Service Bus (ESB). The ESB is a software architecture construct that acts as a infrastructure conduit between services and the components that implement them. Services may be discrete, such as a simple calculation or a composition of services. In either arrangement the services communicate via the ESB with a collection of back-end mechanisms and repositories.

The mechanisms can also be divided on basic logic services and basic data services. The former perform computations, while the latter facilitate interaction with data repositories and data retrievals.

![Diagram](image)

Figure 1: The fundamental SOA

Clearly, this fundamental SOA is a skeleton architecture containing only basic data and logic. For the analysis, the focus is on rudimentary, task-completing services. Therefore, a number of relaxing assumptions are made. Data services allow for read and write data operations while logic services provide computation and results.

Composite services are included for completeness but any cost for orchestration is obscured as being part of the ESB functionality. All services are considered “internal” to the SOA. Consequently full disclosure exists about both their exposed and non-exposed functionality.

Issues related to security, reliability and data mapping are excluded. Similarly, higher level issues such as business processes, policies and governance of the architecture are excluded as well.

B. Generalization
Consider any two implementation instances of the fundamental SOA. They could each have a similar profile as the instance illustrated in Figure 2. Both instances would be architecturally identical, offering a set of services, operations and messages, but the implementations and services of each instance would be different. The implemented services could vary, serving as interfaces for different messages and computations, causing the two implementations to vary significantly. This is to be expected, as the notion of service is an abstraction that conceals implementation details from the consumers.
Therefore, a general model of this implementation instance can be derived using web services and operations as the mechanisms to deliver the basic logic and data services. From Figure 3, we can determine initially that a SOA is made up of one or more web services (WS), each of which is comprised of zero or more web service operations (O) along with messages (M) being exchanged between services. Therefore,

\[ \text{SOA}_1 = \{ \text{WS}_A, \text{WS}_B, ..., \text{WS}_N \} \] (1)

Relation 1, states that SOA$_1$ consists of the set of N web services. A web service is similar to an API or an interface to some back-end computations or capabilities.

A web service is composed of both exposed and non-exposed operations. Web services can be traditional Simple Object Access Protocol (SOAP)-based services or they can also be non-SOAP based such as a Remote Procedure Call (RPC) invocation or a stand-alone module invocation. SOAP based web service operations can be defined, exposed and described using a web service description language (WSDL) file.

This definition describes the characteristics of the service operation without any reference made to the underlying technology which hosts or enables the service to exchange messages [5].

Non-exposed operations are operations that are unavailable as an interface to a client; however the web service uses these operations for internal purposes, similar to a program method within the objected oriented programming language paradigm. Further, it is assumed that an individual SOAP-based web service yields that

\[ \text{WS}_A = \{ O_1, O_2, ..., O_N \} \] (2)

thus describing each individual web services in terms of the N operations they consists of. For non-SOAP-based web services, rather than having distinct operations as they have been already described, they are often one-way invocations of back-end functionality.

The last factor to consider is the messages being transmitted through the SOA. The WSDL file describes the format and details of the messages that are being passed and received. The details include end points, ports, and the message syntax/format.

Additionally, for non SOAP-based web services the messages are often direct exchanges of parameters and data, for example, parameters to a servlet that are appended to the end of a URL.

C. Approach

The fundamental SOA, the generalized services model, and the working assumptions described earlier provide a constrained environment that can be analyzed to determine the balance of a SOA.

The focus of the approach is to dissect the web services of a SOA and attempt to quantify, analyze and attempt to identify meaningful patterns that can provide an indication of the degree of resource utilization and efficiency.

The analysis is based on a simple SOA with five web services working together to geo-rectify an image for a Geography Server (Figure 4).
The operations in this example are based on a workflow task sequence, beginning with the services consumer invoking the Image Service by providing an input image. The Image Service determines the latitude and longitude coordinates associated with the image corner points, via the Latitude and Longitude Services.

The coordinates are passed to a Tile Retrieval Service that will determine, correlate and retrieve all those tiles in the Geography Server associated with the image. The tiles are passed to a Map Overlay Service which makes the association of the image with the tiles and stores this input in the Geography Server.

The geo-rectify example shows the different interactions between web services in a workflow-like fashion to accomplish a specific task. It includes single invocations of web service operations and composition of web service operations to invoke another web service (e.g. the Latitude Service result must be combined with the Longitude Service result to invoke the tile retrieval service). It also demonstrates the interaction with back-end applications (databases) for data logic service calls.

It is apparent from this illustration that certain functions and their associated operations can be assigned to specific SOA components (levels). In Figure 5, the services provider side of the SOA is divided in three levels.

From the perspective of the services consumer, Level 1 is logically closer to the enterprise layer and provides service descriptions, entry points and APIs of exposed services. Level 2 provides the basic data and logic services and Level 3, the logically most distant to the enterprise layer, includes the back-end components such as a data repositories, database management systems (DBMS) or other functionality.

In detail, Level 1 exposes services via WSDL files, which act as an API and provide consumers with the means to interact with the SOA.

Level 2 contains the source code for the operations within each exposed and non-exposed web service. For non-SOAP web services such as RPC invocations, Level 2 may be bypassed and the service works using a direct connection to the back-end layer (Level 3).

Finally, Level 3 includes all the back-end services, applications, and custom code that accomplish the “work” that the SOA is designed for.

Level 3 is the best understood part of the SOA since it contains program code, which can be evaluated by existing methods for developing metrics. Using McCabe’s cyclomatic complexity [12] it is simple to derive many complexity metrics for all the back-end software and provide:

(a) An overall understanding of the flexibility inherent within the organizational structure of the program code, and
(b) The impact and level of difficulty re-factoring the code when changes are introduced.

These particular metrics can offer insight with respect to risk analysis, adaptability, test planning, and re-engineering [3]. Risk analysis provides the means to assess inherent risk during the development cycle. Adaptability measures how well the software may adapt to change over time,
and therefore understanding the expected required level of effort each time a portion of the code is modified or extended. The test planning aspect refers to the ability to determine the number of unique test cases to be generated in order to provide a satisfactory testing state.

Level 2 includes the web service operations for SOAP-based web services, along with their code implementations. This level provides insight on how the operations interact with each other, what services call other services, and what services invoke back-end code. Along with the revelation of the SOA inner workings and the flow between services, this is the level that service compositions may be uncovered and described.

Level 3 containing the WSDL files is the most promising level to determine a “balance” factor associated ($\beta$) with the SOA. This level provides valuable information such as the number of SOAP-based web services and exposed web service operations, while it also allows for the calculation of the average number of web service operations per web service for the SOA (Table 1).

<table>
<thead>
<tr>
<th>Number of Operations per Service</th>
<th>Total Operations</th>
<th>Average Operations ($\bar{x}$) per web service</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>WS 1</td>
<td>WS 2</td>
</tr>
<tr>
<td>SOA 1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SOA 2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SOA 3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SOA 4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SOA 5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SOA 6</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>SOA 7</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 1- Sample SOA Average Operations

In Table 1 there are seven different SOAs (SOA$_1$-SOA$_7$), each consisting of five web services. The options differ based on the number of operations per web service and this is reflected in the results of the rightmost columns (total and average number of operations). In this particular comparison, the total number of operations is ten and the average is two across all examined SOAs.

D. Statistical Hypothesis Testing

The presentation of the data in Table 1 revealed the parameters that would affect the balance factor in a SOA. For example, SOA$_7$ is completely balanced, having two operations per web service. The opposite of this is SOA$_1$ where one web service has the majority of operations, and the rest of the services have very few operations. This depicts a SOA that is greatly unbalanced. One operation per service is considered to be minimal since a web service with no operations indicates a web service without functionality. This distribution of operations per service can significantly skew the results when trying to identify if a SOA is balanced. SOA$_1$ and SOA$_7$ in Table 1 illustrate this example. Both of these SOAs have the same number of total operations and the same average number of operations per web service. Clearly, SOA$_1$ is “unbalanced” with the majority (60%) of all operations concentrated on one service (WS5). At the same time, SOA$_7$ is well-balanced with the number of operations being evenly distributed across all the web services. In the particular example the notion of balance is easily identified due to the small number of operations and web services. However, this may not be the case with larger environments that may contain hundreds of operations and services.
Table 2 - Sample SOA options with Balance Factor

<table>
<thead>
<tr>
<th>SOA</th>
<th>WS 1</th>
<th>WS 2</th>
<th>WS 3</th>
<th>WS 4</th>
<th>WS 5</th>
<th>Total Operations</th>
<th>Average Operations ((\bar{x})) (per web service)</th>
<th>Chi-Squared ((x^2))</th>
<th>Balance Factor ((\beta))</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOA 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>2</td>
<td>0.040</td>
<td>4.042</td>
</tr>
<tr>
<td>SOA 2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>10</td>
<td>2</td>
<td>0.199</td>
<td>19.914</td>
</tr>
<tr>
<td>SOA 3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>10</td>
<td>2</td>
<td>0.406</td>
<td>40.600</td>
</tr>
<tr>
<td>SOA 4</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>10</td>
<td>2</td>
<td>0.557</td>
<td>55.782</td>
</tr>
<tr>
<td>SOA 5</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>10</td>
<td>2</td>
<td>0.735</td>
<td>73.575</td>
</tr>
<tr>
<td>SOA 6</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>10</td>
<td>2</td>
<td>0.909</td>
<td>90.979</td>
</tr>
<tr>
<td>SOA 7</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>10</td>
<td>2</td>
<td>1</td>
<td>100</td>
</tr>
</tbody>
</table>

By using the chi-squared “goodness of fit” statistical technique, a numeric value can be computed that represents the balance of the SOA. This numeric value summarizes the discrepancy between expected and observed values and is expressed in Equation 1:

\[
\chi^2 = \frac{\sum (x_i - E_i)^2}{E_i}
\]

*Equation 1: Chi-squared test statistic*

where \(x_i\) is an observed frequency, \(E_i\) is an expected (theoretical) frequency and \(k\) is the number of possible outcomes of each event.

For instance, utilizing the chi-squared technique [4] on the web services for SOA1 against the perfectly balanced web services within SOA7, generates a chi-square \((x^2)\) distribution value. Multiplying this distribution value by 100, results in a “balance” factor as a percentage.

The closer the balance factor \((\beta)\) value is to 100, the more balanced the SOA is in terms of the number of operations spread amongst web services and the distribution and use of resources. Conversely, the closer this number is to zero, the more unbalanced the SOA. Table 2 includes the results of the balance analysis. From the balance factor results, it is apparent that SOA1 is the most unbalanced (balance factor 4.04). The same factor for SOA7 is close to zero, showing that the operations within this SOA are not distributed among the services. This also indicates that a small number of web services are performing the majority of the work hence increasing the possibility of complexity within those unbalanced parts. SOA2 is the most balanced (balance factor 100) which is also obvious by observing the distribution of operations. While in reality, attaining such an ideal balance factor in heterogeneous, distributed environments is not trivial, still, it provides an insight as to the ideal SOA distribution of operations and web services. Consequently, if a deployed SOA could approach the “optimal” balance factor, then this even distribution of operations over services could signify a less complex overall environment. In practice, SOAs tend to resemble the configuration profile of SOA4 and SOA3 (Table 2). Such SOAs are not completely unbalanced, however they contain a few services that handle the brunt of the business logic, and many other services that are used less often and provide very specific and limited operations.

The rationale behind this metric is based on performing a statistical hypothesis test. The broad approach is founded on a method of making statistical decisions from observations and experimental data. This technique is typically used to determine if experimental results contain enough information to indicate high fluctuations in the observed data distributions. In the case of SOA, the higher the fluctuations the more “uneven” and unbalanced the environment appears to be.

Adopting these tests to the evaluation of the SOA, it is seen that calculating the value of average operations per service for almost any environment is possible. This average value \((\bar{x})\) becomes the expected value \((E)\) while observed values are the actual numbers of operations per service. If the calculated test statistic is large, then the observed
and expected values are not close and this indicates a poor fit of the observed data against the expected value which in SOA terms indicates an unbalanced environment.

III. CONCLUSION

Considerable work has been done in understanding and representing software complexity. Complexity concepts that can be expressed by specific metrics provide better software understandability, modifiability, testability, maintainability and lead to more efficient software development.

Unfortunately, few such metrics exist in the area of software enterprise systems and particularly in service-oriented architectures. These loosely coupled distributed environments can scale and grow uncontrollably, resulting in environments that are difficult to modify, maintain and test. Being able to measure the degree of complexity within these large environments will help analyze bottlenecks and areas needed for improvement, as well as predict issues that may arise on future growth.

The focus of our work was to devise a metric that would be able to indicate, just by parsing WSDL files from a SOA, those areas in the architecture that are heavily used. The implication from the existence of this metric would be to identify resource-intensive web services and potential bottlenecks that may have direct impact on the maintainability, testability and performance of the SOA.

Devising such metrics required the use of a simplified, yet representative model of a SOA. The fundamental SOA that includes all the elements: an ESB, basic data and logic services and back-end facilities, was used to develop a generalized example that could exercise and test the validity of the devised metric. The result of this study was to determine a metric, the balance factor, indicating how skewed or evenly distributed the resources in a SOA were, regardless of the number of services and operations involved.

This metric is just a first attempt to better understand the notion of complexity in SOAs. Developing a single but comprehensive SOA complexity metric is the goal of future work, which would be similar to the McCabe cyclomatic complexity number for source code.

ACKNOWLEDGMENT

Special thanks to Stephen McCabe of McCabe & Associates and Jim Devilbiss for their insightful comments and discussions.

REFERENCES

Towards Model Driven Testing for Mission Critical Software Systems

Emanuel S. Grant¹, and Hassan Reza²
¹Department of Computer Science, University of North Dakota, Grand Forks, North Dakota, USA
²Department of Computer Science, University of North Dakota, Grand Forks, North Dakota, USA

Abstract - A fundamental problem faced in developing and maintaining safety critical software, is that of verification of the system that has just been completed or modified. This problem is most evident in large integrated systems, in which one component has been modified, and it is not known what impact the modification has on the entire system. This work presents an approach to checking the input/output relation of software systems that uses domain-specific modeling languages to model the expected inputs and outputs, and constraints between them. The technique provides a partial solution to the verification problem and is most applicable during the maintenance phase of the system. This approach is applied a program synthesizer for mathematical state estimation problems.

Keywords: Verification, Domain modeling language.

1 Introduction

Traditional approaches to developing safety critical systems, places heavy emphasize on verification, validation, and certification. This is done in order to demonstrate the functional correctness of a system, which is dictated by a set of requirement specifications, or the adherence to certain standards and guidelines [1]. The key verification techniques are classified as dynamic testing and static testing [2]. The essence of dynamic testing is to run the software using a set of test cases based on the source-code, and the specification of the software to show the functional correctness of the software product. Source-code testing is based on graph theory to select test cases.

Software verification is very costly, and is the ultimate hurdle to the release of software system. Verification also plays an important role in system-wide qualities such as reliability, usability, security, etc. The main objective of software verification is to validation of the users’ expectations. In this report, an approach to software system verification is taken that focuses on the input and output of the system, while treating the system as a black box. The approach is presented as case study of a program synthesizer system.

The goal of program synthesizers is the automatic generation of software for families of applications within a specified domain. Program synthesizers are one of the many approaches to delivering customized software products quickly and cost efficiently, from existing libraries of program components and/or parameterized templates and schemas [3, 4]. Program synthesis systems that can generate fully executable code from high-level behavioral specifications are rapidly maturing (see, for example, [5, 6, 7]).

The main components of a program synthesizer (see the un-shaded portion of Figure 1) are: (1) a domain-specific specification language, which is usually declarative, (2) a knowledge base, which encapsulates all of the domain information necessary for code generation, and which can be in the form of generic algorithms, libraries, etc., and (3) a synthesis engine, which will translate the specification from a declarative problem description to executable code by instantiating and/or composing domain-specific code fragments, derived from the knowledge base.

Typically, the domain knowledge encodes families of algorithms (e.g., as templates) that can be instantiated in many different ways into concrete code. These methods provide a way of representing classes of applications that can be specialized quickly and efficiently.

![Figure 1](image-url)

A fundamental issue faced when developing program synthesis technology is that of verifying the correctness of the
derived output code with respect to the input specification. Any software developer must be concerned with the correctness of his/her implementation, but the generic nature of a program synthesizer makes the correctness issue more difficult. Testing such systems is especially difficult because it can be hard to predict exactly how domain knowledge is instantiated/composed to produce concrete programs - by design, program synthesizers must react to a wide variety of possibly unanticipated inputs. Program synthesizers may incorporate advanced techniques that are difficult to test with traditional methods - for example, there may be search involved during the code generation process, which may lead to a large number of possible paths to verify.

To model the input/output relation, domain-specific modeling languages (DSMLs) are used. A large number of DSMLs are being developed and applied (Examples are [8, 9]). DSMLs offer a vocabulary of terms and concepts that are fundamental to the problem and solution domains, whereas the more general modeling language constructs are usually too generic to be directly applied in a solution for some problem domains [3].

This work proposes the use of rigorously specified domain-specific modeling languages [11] (DSMLs) that are based on the Unified Modeling Language (UML) [10] notation and its extension mechanisms and are represented as UML profiles, to model the input/output relation of a program synthesis system. Application of this technique will be demonstrated on a specific application - that of checking the input/output relation of a program synthesizer for mathematical state estimation problems.

2. General Description

The approach taken in this work for verifying program synthesizers is based on a structured set of activities that includes some activities from traditional software development methodologies. The validation process commences with the instantiation of a synthesizer's input equation specification and the corresponding output filter code models by use of the DSML. The instantiated models are the inputs to the verification tool (activity), which generates a verification report (result) as output.

Input/output verification can be carried out automatically, with the use of an analysis tools for UML. The input specification must first be translated from its concrete syntax to the syntax of the DSML. This involves “lifting” from the concrete syntax to the domain-specific concepts defined by the DSML. The translation is straightforward and amounts to parsing the concrete syntax into a UML object diagram. Similarly, the output from the synthesizer must be transformed to an output object diagram.

The USE tool is used in verifying the syntactic and semantic constraints expressed in the DSML. USE is a UML Object Constraint Language (OCL) [10] verifier, developed as a PhD research project and freely available at http://www.db.informatik.uni-bremen.de/projects/USE/. A USE specification is a description of a UML class diagram model with additional OCL constraints, and an object diagram. USE checks the object diagram for conformance to the class diagram and constraints. Since the DSML for the example contains only static concepts represented as class diagrams and the stereotypes for these concepts are defined by OCL constraints, by lifting the input/output concrete syntax to an object diagram, the USE tool can be used for input/output checking. See Section 4 for more details on USE.

3. The Case Study

In this section a description of an application of the verification technique to a synthesizer for state estimation problems, i.e., problems concerning the estimation of the state of an object (e.g., its position, attitude, or noise characteristics) based on noisy sensor measurements. This is an important problem from aerospace and geophysical applications.

The most common way of solving a state estimation problem is to use a recursive update algorithm known as a Kalman filter [12], which provides a statistically optimal estimate of a state based on a model of the dynamics of the problem under study and a model of how the sensor measurements relate to the state.

The program synthesizer takes as input a mathematical specification including equations and descriptions of the noise characteristics and filter parameters. From this specification, it generates code that implements (some variant of) a number of standard Kalman filter algorithms. There are many variations of the Kalman filter algorithm, each variation being chosen according to the problem specifics. For example, a nonlinear problem is usually solved by an extended Kalman filter, a problem with large initial uncertainty can be solved by an information filter, a time-invariant problem is solved with a steady-state Kalman filter etc.

The synthesizer can generate code in unpredictable ways. This is both strength, and a weakness, as it allows interesting solutions to be generated to deep problems but also makes it difficult to keep track of the correctness of the results of code generation. In order to address the latter issue, the input/output verification technique is applied. The main concern with the generated code is that code fragments will be composed that are inconsistent with each other. A schema can be thought of as having a number of slots, which can be instantiated by the schema itself or calls to other schemas. If two schemas instantiate different slots, however, there is a danger that the slots will be instantiated inconsistently due to a bug in the domain implementation. It is time-consuming to
check slot consistency by hand, but by developing independent models of how slots should be connected, it is possible to check slot consistency automatically.

A DSML was developed for the Kalman filtering domain using static models of state estimation problems and of Kalman filter implementations. The models created are UML class diagrams that capture the static components of the input and output of and their relationships, and represent the concrete syntax of the synthesizer’s DSML.

4. Verification Process

Verification of the synthesizer’s input equation specification and output filter intermediate code is made up of the following tasks: (1) verifying the input specification against the input model (syntactic verification); (2) verifying the output filter against the output model (syntactic verification); (3) mutual verification of the input and output semantic constraints (semantic verification).

4.1 Syntactic verification

Syntactic verification of the input equation specification and output filter intermediate may be conducted independently of each other. Verification of the input specification may be done before invoking the synthesizer, to ensure that any error in the output filter code is not because of an incorrect input specification. The syntactic verification includes: (1) Ensuring mandatory classes and associations are included in the input specification and output filter, (2) Ensuring that the types of input specification and output filter are of the types prescribed in the input specification model of and the output filter model, respectively.

4.2 Semantic verification

Semantic verification, like the syntactic verification is conducted partly as a manually and partly as an automatic process and is intended to verify aspects of the input specification and output filter that are other than structural. The semantic verification involves satisfying the constraints associated with the meaning of the input specification and output filter components. This process includes verifying that: (1) The input specification is a correct representation of a set of stochastic equations that models a time-controlled process whose state is to be estimated, (2) The output filter code contains the required components for a filter that is one of the types generated by the synthesizer.

4.3 Verification process

Information about a system state is presented as graphical views in USE. The models developed for verifying the input and output were manually converted from their graphical format into the USE textual format. The OCL constraints were converted to the USE-specific format.

The verification process of the input specification and output filter involves checking that: (1) 1 the mandatory model elements are present in the specification and filter, respectively, (2) the multiplicities of the domain class diagrams are not violated, and (3) all items given in the input specification and output filter appear in the domain class diagrams.

4.4 Results

The outcome of this work produced beneficial results in more than one area of the research. Firstly, it was realized that the development of the models and statements of formal constraints are best executed in a concurrent and iterative manner. Attempts to conduct these tasks as separate and independent task proved difficult and generated incorrectly stated products. Secondly, the process of formalizing the informally stated constraints led to a deeper understanding of the domain concepts, and evinced issues that were not apparent in the informally stated concepts.

A series of test of the verification process were conducted with the example equation specification and filter code. During some of these tests, a number of undetected and unintentional errors in the constraints and models were discovered. Other test involved embedding known errors in the specifications, which the verification process successfully uncovered. From the analysis of test conducted, it was determined that the process is able to detect errors in the specifications of the input and output. Further research will have to be conducted to determine whether this verification process can be successfully applied to other classes of program synthesizers.

5. Related Works

This approach to input/output verification is a form of lightweight verification that emphasizes the role of modeling to highlight what is to be verified. The approach is a form of product-oriented verification in that the result of the synthesizer is verified, rather than attempting to verify the synthesizer itself. There has been some other work on product-oriented certification. [13] checks the result of the program synthesizer Autobayes5 for the violation of simple safety properties, such as safe array bounds access and absence of division by zero. The approach is to encode the safety properties as a set of rules (a safety policy) that can be used to generate verification conditions to be proved by a theorem prover.

A similar approach is pursued in [14] in which term rewriting is used to check functional properties of a program synthesizer system. The product-oriented approach is derived
from the ideas of proof-carrying code [15] in which a compiler is augmented with certificates of partial correctness of the object code generated. A related approach is that of run-time result-checking [16] in which correctness of a particular run of a system is checked at run-time rather than checking the correctness of the software itself - e.g., for a sorting algorithm, it is easier to check that a given sorted list is indeed sorted rather than check the correctness of the algorithm.

None of the reviewed approaches incorporates domain modeling. A review of some of the most current DSMLs shows: that they are based on textual notations, lack a clear definition of the underlying syntax and semantics, and are designed to be used at the implementation phase of software development.

6. Conclusion

In this report, a formalized model-based technique has been present for partial verification of the input specification and output code of program synthesizers. The rational for this work lies in the need to be able to use automatic program generation with an acceptable degree of confidence. The technique was applied to a program synthesizer for state estimation problems from an aerospace domain. The experience gained in this exercise demonstrates that this verification technique can provide a high level of confidence in the use of program synthesizers. However, the processes used in the verification technique have to be nearly fully automated in order to derive the full benefits from the use of such techniques.

Acknowledgement

This work was partially funded under the University of North Dakota - UAS Risk Mitigation Strategy Project.

Reference


Validating Specifications for Model-Based Testing

Submitted to SERP 2008

Pieter Koopman
Software Technology
Radboud University Nijmegen
The Netherlands
Phone/Fax: +31 24 3652(483/525)
Email: pieter@cs.ru.nl

Peter Achten (contact)
Software Technology
Radboud University Nijmegen
The Netherlands
Phone/Fax: +31 24 3652(483/525)
Email: P.Achten@cs.ru.nl

Rinus Plasmeijer
Software Technology
Radboud University Nijmegen
The Netherlands
Phone/Fax: +31 24 3652(644/525)
Email: rinus@cs.ru.nl

Keywords: functional programming, model-based testing, validation tools, quality of specifications

Abstract—In model-based testing the behavior of a system under test is compared automatically with the behavior of a model. A significant fraction of issues found in testing appear to be caused by mistakes in the model. In order to ensure that it prescribes the desired behavior, it has to be validated by a human. In this work we introduce a tool, esmViz, to support this validation. Models are given in a pure, lazy functional programming language. esmViz provides an interactive simulation of the model, as well as diagrams of observed behavior. The tool is built on the iTask toolkit which results in an extremely concise GUI definition. Experiments show that esmViz helps to gain understanding of a model and to detect and remedy errors.

I. INTRODUCTION

In model-based testing the behavior of a system under test, sut, is compared automatically with the behavior of its specification. Examples of model-based test tools are G\texttt{\textasciitilde}st [10], QuickCheck [4], TorX [14], T-Uppaal [12]. The specification is a possibly non-deterministic state transition system used as model in the tests. The number of states, inputs and outputs can be infinite. The sut is assumed to be a state transition system with a hidden state. One can only apply inputs to the system and observe the corresponding output. Key advantages of model-based test tools are the significant reduction of the amount of manual testing: increase of test speed due to automation; and reuse of specifications for regression testing.

Model-based test systems execute a finite number of traces. For each trace the sut and the specification start in their initial state. An input is selected that is covered by the specification, it is applied to the sut, and the allowed states of the specification are computed. If, during this process, the test system discovers that no states are reachable for the specification, then the sut has shown behavior that is not covered by the specification. In test jargon it is said that an issue is found.

Ideally, each issue indicates an error in the sut. However, in practice a significant fraction of issues appear to be caused by problems with the specification: it does not correctly capture the intentions of the users and the sut does something different. Even though the fraction of issues depends on a lot of factors such as the kind of system and the effort spent in creating the model, we estimate that the specification has to be blamed for about 25% of the issues.

Incorrect specifications are a problem for several reasons. First, if an issue is found it is not clear whether we have to blame the specification or the sut. Finding and correcting errors in the specification takes time during the test phase of the project. Second, errors in the specification are only found during model based testing if the behavior of the sut differs from the specified behavior. Third, any change in the specification during the testing phase can cause major implementation changes to the sut. Finally, any change in model or sut invalidates in principle all previous test results. Hence, errors in the specification can be very expensive and it is worthwhile to invest effort to ensure its quality.

In the model-based test system G\texttt{\textasciitilde}st the pure, lazy functional language Clean serves as specification language. Due to its high abstraction level it is possible to write concise specifications which contributes to their quality. It allows the test engineer to model arbitrarily large state, input, and output domains exactly as desired. The advantages have been presented earlier ([8], [11]). The Clean compiler checks quality aspects like type correctness and consistent definition of used identifiers. Other quality aspects such as the reachability of states, determinism and completeness, and the preservation of constraints can be checked by systematic testing [9].

The use of a high level specification language does not rule out the possibility that the specification prescribes the wrong behavior in a consistent way. Hence, these kinds of errors can not be found by the above mentioned techniques. In order to ensure that the specification prescribes the desired behavior, it has to be validated by a human. In this work we introduce the tool esmViz to support validation of G\texttt{\textasciitilde}st models. This simulator enables the user to execute the specification. Such an interactive execution appears to be more illustrative than reviewing the specification. Second, it is possible to record the traces of the specification executed in the simulator. The states visited and their transitions can be visualized in an expanded state transition diagram. Since the type of states, inputs and outputs can be infinite and different in each and every specification, doing this conveniently is not straightforward. The key to the solution is to use generic definitions such that operations
on these types can be derived instead of defined manually.

The layout of the paper is as follows: in Sect. II we introduce the concepts and notation that will be used throughout this paper. In Sect. III we discuss the issues that arise when testing against a formal specification. In Sect. IV we describe esmViz. Its implementation is discussed in Sect. V. Related work is discussed in Sect. VI. We present user experiences in Sect. VII, and conclude in Sect. VIII.

II. Model-based Testing

In model-based testing the test tool compares the observed behavior of the system under test, sut, with the model in order to judge the correctness of the behavior. Any deviation of the observed behavior of the sut from the behavior allowed by the model is called an issue. In this section we review the models used by the model-based test tool G\textsuperscript{Vst}.

The models used by G\textsuperscript{Vst} for testing state based systems are extended state systems, ESMs. An ESM consists of some initial state \( s_0 \) and a set of transitions of the form \( s \xrightarrow{i/o} t \). In such a transition \( s \) is the source state, \( i \) is the input triggering this transition, \( o \) is the output of the system associated with this state and input, and \( t \) is the target state of the system. The sets of possible states \( S \), possible inputs \( I \), and possible outputs \( O \) of the ESM can all be infinite. The \( i/o \) combination is also called the label of the transition from \( s \) to \( t \).

A trace \( s \xrightarrow{\sigma} t \) is a sequence of labels. The empty trace contains no labels. If we have a trace \( s \xrightarrow{\sigma} t \) and a transition \( t \xrightarrow{i/o} u \) we construct the trace \( s \xrightarrow{\sigma;i/o} u \). If we are not interested in the target state, we will occasionally write \( s \xrightarrow{i/o} \equiv \exists t, s \xrightarrow{i/o} t \) and \( s \xrightarrow{\sigma} \equiv \exists t, s \xrightarrow{\sigma} t \). All traces from a given state are defined as: \( \text{traces}(s) \equiv \{ \sigma | s \xrightarrow{\sigma} \} \). The init of a state \( s \) is the set of inputs \( i \), such that there is an output \( o \) and target state \( t \) in the ESM such that there exists a transition \( s \xrightarrow{i/o} t \). The after of a state \( s \) is the set of possible target states \( t \), reachable after the given trace \( \sigma \); \( s \xrightarrow{\sigma} \equiv \{ t | s \xrightarrow{\sigma} t \} \). We overload traces, init, and after for sets of states instead of a single state by taking the union of the element sets.

A. Conformance

In model-based testing we try to determine conformance of the sut and the model called spec. The sut is assumed to be a transition system, but treated as a black box: one can observe its traces, but not its internal state. During tests, all observed traces of the sut have to be traces of the specification to say that the sut conforms to the specification. Formally, this relation is defined as:

\[
\text{spec} \xrightarrow{\sigma} \text{spec} \equiv \forall \sigma \in \text{traces}_{\text{spec}}(s_0), \\
\forall i \in \text{init}(s_0 \xrightarrow{\text{after}_{\text{spec}} \sigma}), \\
\forall o \in O. \\
(t_0 \xrightarrow{\text{after}_{\text{spec}} \sigma}) \xrightarrow{i/o} (s_0 \xrightarrow{\text{after}_{\text{spec}} \sigma}) \xrightarrow{i/o}
\]

Here \( s_0 \) is the initial state of spec, and \( t_0 \) the initial state of sut. Intuitively the conformance relation reads: if the specification allows input \( i \) after trace \( \sigma \), then the observed output of the sut should be allowed by the specification. If spec does not specify a transition for the current state and input, anything is allowed. Because the sut is a black box, its initial state \( t_0 \) is generally not know explicitly. We assume that the sut is in this abstract state when we switch it on, or we reset it.

Limiting the applied inputs to the init of the states of the current traces allows for partial specifications spec.

B. Testing Conformance

The conformance relation defined above covers all traces. Most interesting systems contain cycles, so traces can become infinitely long. Due to the possible infinite types for input and output, there can be even infinitely many traces of finite length. It is clear that in general a test system cannot prove conformance by executing tests. The test system G\textsuperscript{Vst} approximates the conformance of the sut to the model by executing a finite number of traces of finite length.

To increase efficiency the test system records the set of allowed states, \( s_0 \xrightarrow{\sigma} t \), rather than the trace \( \sigma \). If at some point in the test this set of states becomes empty we have found an issue: a trace that shows that there is no conformance between sut and the model. Clearly this way of testing is sound, each trace leading to an issue during testing shows that there is no conformance between the sut and the model. This way of model-based testing is also complete, if there is no conformance between sut and the model, there are one or more traces indicating this. Such a trace can be found by testing (if the allowed length during tests is sufficiently large).

C. Representation of the transitions

To represent the ESM in the model-based test tool G\textsuperscript{Vst} we need a finite (preferably small) and flexible representation, even if the set of transitions is infinite. Furthermore it should be easy to determine the init of the set of actual states, or to determine if an input is in this set, since this information is needed before we can apply an input during model based testing. The crucial step is to use a function to model the transitions rather than a data structure containing individual transitions. Each function alternative with variables in its patterns captures a family of related transitions. As usual lists represent sets. To define init easily we use specifications of type \( S \times I \rightarrow [\text{Trans} O S] \).

A basic assumption in G\textsuperscript{Vst} is that a transition always contains a sequence (list) of output symbols. This gives some additional flexibility as well as a suitable notation for no output (the empty list). Usually it is most convenient to specify the sequence of outputs and the target state in a transition. However, the number of allowed output sequences for one input can get huge, which makes it infeasible to state them explicitly. For instance in an authentication procedure a typical input can get huge, which makes it infeasible to state them explicitly. For instance in an authentication procedure a typical input can get huge, which makes it infeasible to state them explicitly. For instance in an authentication procedure a typical input can get huge, which makes it infeasible to state them explicitly. For instance in an authentication procedure a typical input can get huge, which makes it infeasible to state them explicitly. For instance in an authentication procedure a typical input can get huge, which makes it infeasible to state them explicitly. For instance in an authentication procedure a typical input can get huge, which makes it infeasible to state them explicitly. For instance in an authentication procedure a typical input can get huge, which makes it infeasible to state them explicitly. For instance in an authentication procedure a typical input can get huge, which makes it infeasible to state them explicitly.
output sequence. Again, a single function captures a family of related transitions. In Clean these types are:

```plaintext
: : Spec s i o := s i → [Trans o s]
: : Trans o s = Pt [o] s | Pt ([o] → [s])
```

Note that we use type parameters to allow any concrete type to be used for state (s), input (i), and output (c).

1) Example: As an example specification we show the model of a beverage vending machine that supplies coffee and tea (see Fig. 1). Initially the machine is in a state called Off.

![Fig. 1. The intended specification of the beverage vending machine](image-url)

After the input SwitchOn it enters state On 0 without producing any output. The integer in this state is used to record the amount of money inserted. Now the user can either insert a coin with a value given as parameter as long as the counter in the state remains less than Max, or press a button to receive a product. If there is enough money the user gets his product and the value of the counter is decreased accordingly. The types used in this model are:

```plaintext
: : Money := Int
: : State = Off | On Money
: : Input = SwitchOn | SwitchOff | Coin Money | Butt Product
: : Product = Coffee | Tea
: : Output = Cup Product | Return Money
```

A possible specification is given as the function `vSpec` below. We deliberately introduce some errors and strange transitions in this specification, later we return to it in an attempt to find these problems.

```plaintext
vSpec :: !State !Input → [Trans Output State]
vSpec Off SwitchOn = [Pt [] (On 0)]
vSpec s SwitchOff = [Pt [] Off]
vSpec (On s) (Coin c)
  // condition should be s+c<Max
  | s<Max  = [Pt [] (On (s+c))]
  // output should be Return c
  | = [Pt [] (On s)]
  // pattern should be (Butt Coffee)
vSpec (On s) (Butt coffee)
  | s>20  = [Pt [Cup Coffee] (On (s-20))],Pt [] (On s)]
vSpec (On s) (Butt Tea)
  // we get Coffee instead of Tea
  | s>10  = [Pt [Cup Coffee] (On (s-10))]
  // do nothing for other buttons
vSpec (On s) (Butt p) = [Pt [] (On s)]
  // otherwise: nothing defined
vSpec state input = []
```

This specification is partial (e.g. the effect of pressing a product button when the machine in the state Off is not defined), and nondeterministic (if there is enough money in the machine and the user asks for coffee, the machine either produces coffee, or does nothing at all). Non-determinism models limited knowledge of the state of the real machine: e.g. if there are coffee beans it will produce coffee, otherwise it cannot produce coffee and waits for a new command.

III. ISSUES FOUND IN MODEL BASED TESTING

Issues are traces that show that there is no conformance between the sut and the specification. Ideally each issue found indicates an error (bug) in the sut, but that is not always the case. Other sources of issues are inaccuracies in the model, problems in the interface between the test system and the sut, and internal faults in the test tool. One wishes to eliminate these other sources of issues before actual testing starts.

In ordinary automatic testing the test tool executes a manually specified or recorded trace. As a rule of thumb test engineers say that 40% of the issues found in this kind of tests indicates a real error in the sut. A tiny fraction of these issues is caused by the test tool itself, or the interface with the sut. Most issues are caused by the fact that the trace used does not correspond to the current version of the specification, or the specification itself is incorrect.

In model-based testing the traces are generated automatically and on-the-fly from the specification. This guarantees that the traces used during the tests always correspond to the current specification. As one expects this implies that a larger fraction of the issues found indicate errors in the sut. In our experience about 75% of the issues found during model-based testing indicate errors in the sut. The fraction of actual errors depends on the amount of effort spent on making a high quality specification, the quality of the informal specification and requirements used as basis, and the size and complexity of the system.

The specification is a Clean function, hence the compiler can readily check relevant properties: i) all used identifiers properly defined, ii) is the entire specification type correct, iii) are all alternatives (transitions) reachable. Still, well typed specifications can go wrong. The problems with specifications that cannot be detected by the compiler can be divided in the following classes.

1) Relevant behavior of the system is not covered in the specification. Since the test system is carefully designed to handle partial specifications, this cannot be detected. Missing parts of the behavior are not covered in the tests.

2) The specification contains design errors. Typically a family of transitions is too large or too small, or leads to the wrong target state. If the sut does a better (or at least different) job, the test system will notice the difference if an appropriate trace occurs and hence reports an issue. Consider the alternative for `vSpec (On s) (Coin c)` in the example of the previous section. The wrong condition and forgotten return of money if the state becomes too large are probably design errors.

3) The transitions are designed correctly, but the implementation is incorrect. A typical example is the use of lowercase identifiers (variables) where an uppercase
identifier (constructor) is intended, or vice versa. Another source of problems is copy-paste programming used to define similar transitions, where not all necessary changes are made. In our example this occurs in the transitions for vSpec (On s) (Butt coffee), and vSpec (On s) (Butt Tea).

All these problems result in well typed models. If the implementation is based on such a model, it is not possible to detect the problems by testing. Nevertheless, they must be found and preferably before model-based testing starts.

Various approaches to find these kind of problems are:

**Inspection or reviews of the specification.** Problems can be found by manual inspection of the specification. As the model tells the whole story, there is nothing that prevents these errors from being detected by reviewing the code. However, due to their subtle nature, they might be missed.

**Model checking.** If we have the right properties and the specification is available in a form suited for model checking, the problems can be found by model checking. Limiting factors are the availability of the model in a form suited for a particular model checker, and the availability of properties to check. If the problems are known we can often find such a set of properties quite easily, but that is too late. In our example we can require: p1) every transition preserves the amount of money, p2) the amount of money in the machine is always less then Max, and p3) if we receive a product, it must be equal to the requested product. Finding a complete set of properties that reveals all problems is in general quite tricky.

**Testing properties of the specification.** Properties on transitions can be tested by the logical branch of Gvst. The advantage is that everything can be done within the same framework, especially the Clean specification function can be used as subject of tests. The drawback is that testing gives less certainty for large systems (although for small specifications the logical test system provides a proof by exhaustive testing).

**Validation by simulation.** The specification can be used as basis for an interactive simulation. With some effort the simulator not only displays the current transition, but also depicts the state space that is covered in the current simulation. Such a simulation can reveal that (important) parts of the behavior are missing, as well as problems with individual transitions. This requires a thorough observation of the shown behavior. Since the state space is discovered step-by-step by the user, the chances of finding the problems are quite good.

Each of the above methods can in principle find problems in the specification, but none of them can guaranteed this. Each method either requires human spotting of problems, or human formulation of properties revealing the problems.

In the remainder we describe esmViz. It combines model-checking of properties on transitions with validation by stepwise simulation. Together with Gvst, this covers a broad range of tools to investigate the quality of models.

**IV. Validation of Specifications with esmViz**

In this section we describe the web browser-based simulation tool, esmViz, that we have created to determine the quality of specifications. The tool also gives an impression of the behavior specified by the model, and checks user defined predicates on the transitions encountered. Simulation is useful to give non-experts a good impression of the specified behavior. The GUI of esmViz is a screen with the following elements (Fig. 2(a)): 1. A list of found issues. The list is empty in Fig. 2(a). 2. The explored model as an Extended State Diagram (ESD). 3. Within the ESD set of possible active states determines the inputs that can be given. These are enumerated as buttons that the user can press to advance the system one step. In Fig. 2(a) the active states are $S = \{0, 20\}$, and init $(S) = \{\text{ButtCoffee}, \text{ButtTea}, \text{Coin10}, \text{Coin20}, \text{SwitchOff}\}$. 4. Commands for navigation purposes, resetting the exploration, and so on. 5. The current trace, as explained in Sect. II. Here the trace has length 4. 6. Finally, a legend that tells what the elements of the rendering are.

The tool esmViz creates an ESD of the behavior encountered during simulation which is rendered as a directed graph. In ESM diagrams a parameterized state is drawn as one state, in the ESD a state is created for each value of the parameters encountered during simulation. In the beverage vending machine example the states $(On 10)$ and $(On 20)$ are different in the ESD, but they are one state in the ESM (Fig. 1). A transition $s \xrightarrow{i/o} t$ is rendered as an arrow between state $s$ and state $t$, and has label $i/o$ at its edge.

**A. The ESM description**

The ESD is created by esmViz based on an ESM and instances of generic functions used for instance to display and compare values of the data types used for states $S$, input $I$ and output O. The ESM as described in Sect. II is a Clean value of type $(ESM S I O)$:

```
:: ESM s i o = { s_0 :: s ,
    d_F :: Spec s i o ,
    out :: s i → [[o]] ,
    pred :: (SeenTrans s i o) → [[String]]}
:: SeenTrans s i o ::= (s,i,[o],s)
```

Field $s_0$ is $s_0$, and $d_F$ is $δ_F$. The function out is needed to generate the output sequences to be used when esmViz encounters a transition of type $[o] → [s]$. If such transitions can not occur in the used specification, this field can be undefined. Field pred is a predicate over the transitions seen during simulation as discussed in Sect. III. Each problem detected is reported as a nonempty list of strings.

While exploring esm, the tool collects all visited states, transitions and issues. This results in a partially known automaton, and is captured concisely with the following type:

```
:: KnownAutomaton s i o
    = { trans :: [ SeenTrans s i o ] ,
        issues :: [(SeenTrans s i o,[String])] }
```

Encountered states can be extracted easily from the seen transitions and are not recorded separately. Transitions that correspond to an issue are drawn in red.

The tool esmViz also indicates the transitions that are part of the current traces. For a nondeterministic specification there
can be multiple traces active. We record this as a list of transitions that is possible in each step of the trace.

:: Trace s i o ::= [[SeenTrans s i o]]

Trace transitions are drawn in blue with larger arrowheads.

For implementation reasons it is convenient to record the set of active states. For a nonempty trace these are exactly the states in the after set of the current traces. Let this set after \( k \) steps be \( S_k \). Each state in \( S_k \) is rendered with a red interior. States are displayed as circles, where \( \text{esm}\_s_0 \) has a double border. Initially, \( S_0 = \{ \text{esm}\_s_0 \} \).

The user can choose one input of \( \text{init}(S_k) \), which is the set of all possible inputs. This set of inputs is empty for a final state. The number of possible inputs is limited (by default 50). Given a concrete choice \( i \in \text{init}(S_k) \), \( \text{esmViz} \) adds all transitions from the current states that correspond to this input.

For transition specified by \( \text{Pt o t} \) in spec, the output and target state to be used are immediately clear. For transitions specified by a function \( \text{Pt f} \) of type \([s] \rightarrow [t] \), the function \( \text{esm.out} \) is used to determine the outputs and target states of transitions.

If the target states of these transitions exist the arrows go to the existing states, otherwise the states are added to the ESD. The new transitions are also added to the traces, and the existing part of the trace is pruned to reflect the new extensions. The set of new states \( S_{k+1} \) is computed with \( S_{k+1} = \{ t \mid s \in S_k \land s \rightarrow o t \in \delta_F \} \).

The system determines for each known state whether the user has ‘discovered’ all outgoing edges, i.e., all edges with \( i \) in the \( \text{init} \) of that state. In that case, the state is rendered with a blue interior instead of a light grey default one. This provides a strong clue which part of \( \text{esm} \) has been fully explored.

Pressing the button labeled Back removes the last transition from each trace. The known automaton is not affected by going back in the trace. The browser’s back button acts as undo action. With the Add all button all transitions leaving from the current states are added. These transitions are not added to the trace, nor effect \( S_k \). Using an integer edit field, adding transitions can be done recursively \( n \) steps deep. Pressing Prune removes all transitions and associated issues that do not belong to the current trace. The Reset button brings the \( \text{esmViz} \) tool in its initial state, only the state \( s_0 \) is displayed. The trace can be removed by the button Clear trace, the states and transitions in the ESD are not effected by this action.

The current state can be changed by clicking on a state in the diagram. If this state is part of the trace or reachable from an active state in one step the trace will be adapted accordingly, otherwise a new trace starts at that node.

B. Example

Here is the beverage vending machine \( \text{esm} \) specification:

\[
\text{vendingESM} :: \text{ESM State Input Output}
\]

\[
\text{vendingESM} = \{ s_0 = \text{Off}, d_F = \text{vSpec}, \text{out} = \text{undef}, \text{pred} = \text{healthy} \}
\]

where healthy checks \( p1 - p3 \) (Sect. III). An ESD showing all issues discribed by healthy is depicted in Fig. 2(b).

\[
\text{healthy} ::= (\text{SeenTrans State Input Output}) \rightarrow [[\text{String}]]
\]

\[
\text{healthy} (s,i,o,t) = \begin{cases} \text{true} & \text{if } (\text{vs}+\text{vi} \neq \text{vt}+\text{vo}) \text{ // value preservation in transition? (p1)} \end{cases}
\]

\[
\begin{cases} \text{\"value is not preserved in this transition, "} \\ \"\text{value s+value i, toString Max}\\text{ vs+vi, toString vt}\\text{ vt>Max}\" \text{ to the delivered product!}\} & \text{if } (\text{vt} \geq \text{Max}) \text{ // value of target state within bound? (p2)} \end{cases}
\]

\[
\begin{cases} \text{\"Value of target state \\"toString vt}\\\" larger than Max (\"toString Max.\")\} \text{.\"} \} & \text{case } (i,o) \text{ of } // obtained the ordered product? (p3) \end{cases}
\]

\[
\begin{cases} \text{\"The required product is unequal\"} \\ \"to the delivered product!\"\} \end{cases}
\]

\[
\text{where } vs = \text{value s}; \text{vi} = \text{value i} \\
\text{vo = value 0; vt = value t}
\]

V. IMPLEMENTATION

The \( \text{esmViz} \) tool has been written in Clean, using the iTask toolkit [13]. It offers a fair amount of functionality (see also other tools in Sect. VI). In this section we present the most interesting parts of the implementation. These are the main structure of the GUI (Sect. V-A) and the integration of the ESD rendering tool Graphviz [5] that we used in the application (Sect. V-B).
A. The Main GUI Structure: Iterating iTasks

The main GUI structure of esmViz is an iteration of the main tool task function DiGraphFlow. As discussed in Sect. IV, it provides the user with a number of elements, expressed as a list of choices (the arguments of orTaskL below which folds the basic iTask –|– choice operator over the list):

\[
\text{DiGraphFlow } (\text{ka.as.trace}, \text{n})
\]

orTaskL

\[\text{issuesToHtml ka.issues !! state}
\]

\[\text{chooseTaskV} (\text{sortBy } \lambda (\text{a}, \text{b}) \rightarrow \text{a} < \text{b})
\]

\[\text{[(render i,step i) } \\triangleright \langle i \rightarrow \text{possibleInputs esm as}? \rangle
\]

\[\text{chooseTask}
\]

\[\langle \text{"Back" , back}
\]

\[\langle \text{"Prune" , prune}
\]

\[\langle \text{"Reset" , return_V (newKA,[esm.s_0],[]),n))
\]

\[\langle \text{"Clear trace" , return_V (ka.as,[]),n))
\]

\[\text{.stepN <<= ! traceHtml trace NOTIFY!!! legend ]}
\]

The correspondence between this definition and the GUI as displayed in Fig. 2(a). The list of found issues are displayed before the ESD editor (line 3); the possible inputs init S0 are defined in lines 4–5; the navigation commands are summarized in lines 6–11; and finally, the trace and legend are displayed in line 11. The state task is given below:

\[
\text{state}
\]

\[\text{= editTask } \text{"OK"}
\]

\[\text{mkDiGraph ThisExe}
\]

\[\text{let}
\]

\[\text{as.'trace'} = \text{findSelectedStates dig ka as trace}?
\]

\[\text{in return_V (ka.as,.'trace',n)}
\]

The iTask editTask l v combinator creates an editor with initial value v with which users can create new values of the same type as v's type. When the button labeled with l has been pressed, then the new value is returned by this editor and the task is done. As discussed in Sect. IV, the user can select a new state. For reasons of space, we do not show the code of the other functions. The entire GUI part of the tool is only 130loc.

B. The Rendering of the Explored Automaton

By far the most intricate component of the GUI is the ESD editor. Creating attractive renderings of directed graphs is known to be a hard problem. Fortunately, we can rely on other tools to solve this problem. Here we have used the Graphviz tool set [5]. Directed graphs are described using the DOT language. Given a DOT text file, the dot tool can be invoked to create a rendering in various formats (we will use the gif output). Note that this interface is text-based, whereas editors in the iTask toolkit are type based. We can embed the text based tools of Graphviz in the type based iTask toolkit in a compositional way by defining a suitable collection of data types that describe an ESD as a directed graph. This collection of data types captures the DOT language. The relevant top level type definitions are:

\[
\text{:: Digraph } \Rightarrow \text{ Digraph String [GraphAttribute]}
\]

\[\text{[NodeDef]}
\]

A (Digraph name attrs nodes item) value represents a directed graph. A directed graph has nodes, each of which is identified by a number, and is connected with other nodes by means of edges. Graphs, nodes, and edges have attributes. Graphviz supports an extensive set of attributes (almost 150) that can be used to alter and tweak the output. In DOT, attributes are specified as name = value pairs. Some attributes are shared by graphs, nodes, and edges. We have represented attributes separately for graphs, nodes, and edges, each as a list of unary data constructors. For instance, for graph attributes we have GAtt_name value pairs. A single generic function prints these values as correct DOT expressions. The result is that we have both a typed representation of DOT expressions (Digraph values) as well as a textual representation (printing such a value with toString). The function mkDigraph yields the Digraph value that represents the currently explored ESD. There are 326loc of type definitions and access functions.

The iTask editor for Digraph values performs the following actions for a d :: DiGraph value identified by name. First, compute e = toString d and save e in file name.dot. Second, invoke dot on name.dot, which yields a rendering as name.gif. Third, invoke dot to create a name.map file to allow the user to select states. Fourth, alter the names in name.map to invoke a script that sends the label of the selected state to the server application. Finally, generate the proper HTML to be included in the application page. The server application, when receiving the label of a selected state, updates the corresponding Digraph value to reflect the change. Now the application continues with the new Digraph value. This part of the implementation is 214loc.

VI. Related Work

The mCRL2 tool set [6], [1] uses a process algebraic specification language, mCRL2 [7], to describe distributed, communicating systems. It has a functional style data language with recursive types, data constructors, functions, lambda-abstraction, and structured data. It comes with an extensive number of tools (15) for analysis purposes. Five are relevant to our work: with xsim a user can explore a linearized mCRL2 specification in a similar way as with our tool, using a GUI (the simulation tool sim has a command line interface): the user can select actions, after which the tool shows the resulting state. Besides interactively exploring the mCRL2 specification, the tool set also allows to render the complete state space: NoodleView (for 2D rendering) and FSMView (for 3D rendering). Before this is possible, the state space needs to be generated with lps2lts.

The TorX tool set [14], [2] is a model based test tool to check conformance of real suts, based on the ioco theory of testing. The specification is a Labeled Transition System (LTS), or one that is derived from a higher level specification language that converts to LTS (e.g. mCRL2 described above). The tool uses
the specification to automatically determine inputs, observe outputs from the SUT, and make a final verdict. In this sense, it is not useful for exploring a specification. However, once a test run has been created, the user can explore the actual trace which is depicted as a message sequence chart.

The Uppaal tool set [3], [12] can be used for both validation and verification (using model checking) of time-based systems. Validation is done by means of a graphical simulator of a time-based automaton specification. The automaton specification is basically a labeled transition system with timing constraints. Uppaal allows for simple data types, clocks, and constraints on these clocks. The user can create specifications in an intuitive, graphical way. The user can stepwise direct the system’s behavior, or generate a random trace.

The esmViz tool differs with the mCRL2 approach in that we use a single modeling formalism. Except for the 3D rendering all of the functionality of the mCRL2 tool set is available in esmViz. The TorX tool set is really a model based testing harness, and is less suited for exploration purposes. Specifications within Uppaal can be created graphically. In esmViz specifications are given as a function, out of which a graphical approximation is ‘discovered’ by the user or by the system. In our opinion this combines the best of both worlds: the succinctness of functional programming with the intuitive appeal of a graphical rendering.

VII. EXPERIENCES

In order to judge the quality of esmViz 10 master students in computer science studied some test cases with and without esmViz. These students are literate Clean programmers, have a basic understanding of model-based testing with G\textsuperscript{\prime}Vis and the specifications needed (but no hands-on experience). After an introduction to esmViz and playing with an example similar to the beverage vending machine in this paper the students were asked to locate problems in two other case studies. The examples were heavily parameterized specifications of a number guessing game and a telephone number database that contains potentially over one million states. Drawing all these states makes finding the problems only harder. The errors in the specification can however all be found by traces of about ten to twenty transitions.

The students found esmViz very handy to get a feeling for the behavior of the specified system. Everybody found it much easier to understand a specified system with the tool than without. Finding errors in the specification by simulation remains hard, but the tool makes it easier. The same holds for finding the source of issues found by G\textsuperscript{\prime}Vis. This is consistent with the general observation in all kinds of testing: finding issues is one thing, but finding their cause is another.

VIII. CONCLUSIONS

There are two kinds of conclusions from the work described in this paper. First, the specification simulator esmViz described in this paper really helps a lot to understand the behavior of the extended state machines used as specification in model-based testing. Although the compiler of the statically typed functional programming language used as carrier of these specifications checks the models, the models can still contain errors. Finding these semantical errors is hard. The simulator helps in locating these problems, especially if an appropriate constraint on transitions or states is known. Second, implementing such a tool with iTasks is a real pleasure. Integrating Graphviz with iTasks turned out to be smooth. Implementing a browser interface for esmViz using the iTask system imposes some restrictions on the layout of the GUI, but works well. The different possible user actions are modeled each by an iTask. The iTask system is well suited to compose these tasks flexibly.

REFERENCES

A Constraint-Based Approach to Verification of Programs with Floating-Point Numbers

Carlos Acosta, Martine Ceberio, and Christian Servin

Computer Science Department
University of Texas at El Paso
El Paso, Texas 79968-0518
Email: ceacosta@gmail.com, mceberio@utep.edu, christians@miners.utep.edu
Telephone: (915) 747–6950
Fax: (915) 747–5030
SERP’08

Abstract—Software plays an important role in our daily lives. It can even be of extreme importance, e.g., software included in our cars such as the anti-lock brake systems, software applications controlling airport traffic, applications used in hospitals to deliver radioactive treatment to patients, programs used in nuclear stations. Ideally, we want programs to always work right and as specified. That is, we want reliable software. One of the things we can do to achieve reliable software is to verify it and validate it. Validation and Verification (V&V) are two terms that are similar and both terms have been used to refer to all the activities we perform to check that software does what it is supposed to do.

Constraint Programming has been successfully used in solving scheduling problems and circuit design, among others. Its use in verification of software is still being researched and it is being applied to automatically generating test cases and to showing the conformity of software to its specification.

In this work, we propose a different approach, based on constraints, to translate code constructs where existing approaches made use of guarded constraints. We discuss the challenges and advantages of our approach, and we describe a process for solving constraints of the form \( \neg A \land B \), which arise in our approach.

Keywords. Software verification, constraint solving, floating-point programs, inner/outer approximations.

Contact author: Martine Ceberio

I. INTRODUCTION

Software plays an important role in our daily lives. There is software in our cell phones, in our workplaces and in our homes just to mention a few examples. There is also software that is even more important; for example, software included in our cars such as the anti-lock brake systems, software applications controlling airport traffic and software in the airplane itself, applications used in hospitals to deliver radioactive treatment to patients, programs used in nuclear stations. These kinds of applications are critical: human-lives depend on their functionalities. Ideally, we want programs to always work right and as specified [12]. Programmers and users want specially the latter kind of software to work right. That is, we want reliable software. One of the things we can do to achieve reliable software is to verify it and validate it. Validation and Verification (V&V) are two terms that are similar and both terms have been used to refer to all the activities we perform to check that software does what it is supposed to do. We adopt the following meaning for validation and verification. Validation refers to checking that a system satisfies its specification (usually checking that the design specification satisfies the users requirements) whereas verification refers to proving that a system satisfies its specifications (usually proving that the code satisfies the design specifications) [12], [13].

Constraint Programming has been successfully used in solving scheduling problems and circuit design, among others. Its use in verification of software is still being researched and it is being applied to automatically generating test cases and to showing the conformity of software to its specification. In this work, we survey constrained-based verification techniques [3], [4], [11] and we propose a different approach, translating code construct where existing approaches made use of guarded constraints. We describe the challenges and advantages of our approach. In particular, we describe a process to solve constraints of the form \( \neg A \lor B \), which arise in our approach.

II. COMMON VERIFICATION TECHNIQUES

In this section, we present common verification techniques as listed by Balci in [1], [2], and shown on Figure 1. Balci classifies these techniques as follows:

- **Informal verification techniques**
  These techniques usually involve human participation and rely mainly on human reasoning. Some of the techniques that fit in this category are the following: audits, inspections, walkthroughs, and reviews.

- **Static verification**
  These techniques do not require machine execution of the model. Some of the techniques that fit in this category are the following: data flow analysis, syntax analysis, graph-based analysis, and structural analysis.

- **Dynamic verification**
  These techniques require model execution. Some of the techniques that fit in this category are the following: testing, debugging, and execution tracing.
Validation, Verification, and Testing Techniques

- **Symbolic verification**
  On these techniques symbolic inputs are given to a model and the output consists of expressions that result from the transformation of the symbolic inputs after going through execution paths in the model. Some of the techniques that fit in this category are path analysis, symbolic execution, cause-effect graphing, and partition analysis.

- **Constraint verification**
  Constraint verification techniques use assertion checking, boundary analysis, and inductive assertions. Some of the techniques that fit in this category are assertion checking, boundary analysis, and inductive assertions.

- **Formal verification**
  These techniques are based on formal mathematical proof of correctness. Some of the techniques that fit in this category are lambda calculus, logical deduction, and proof of correctness.

### III. RELATED WORK ON CONSTRAINT-BASED VERIFICATION TECHNIQUES

#### A. Automatic test case generation

In this section, we briefly describe the work of Gotlieb et al. [9]. In their approach, the authors mainly discuss their work for computations with integers.

The approach taken by Gotlieb et al. [9] consists of automatically generating test data that will execute a selected point in the code. In this approach, they transform the code into Static Single Assignment (SSA) form and analyze control-dependencies. They build a constraint system with this information and then they solve this system. When the constraint system is solved, test data is generated such that the selected point executes (if there is a feasible path that leads to the selected point).

The main steps of Gotliebs et al. can be outlined as follows:

1) Translate the program into a constraint system from its SSA form and control-dependencies.
2) The result of this step is a set of constraints (Kset). This set consists of the constraints generated for the program and the constraints that are generated for the selected point.
3) Solve Kset to generate test data for the selected point if there is at least one feasible path to the selected point.

#### B. Conformity of specifications and code

In this section, we briefly describe the work of Rueher et al. [3], [5] Their approach handles only operations with integers, i.e., they work on discrete domains. In this section, we first present an overview of the work in [5], which is the base of our work. Then, we describe the steps followed by Rueher et al.

The approach taken by Collavizza and Rueher consists in transforming the program and its specification into a constraint system. In this work, a program is verified if the union of the constraints derived from the program and the negation of the constraints derived from its specification is inconsistent (i.e., it does not have a solution). Consider that we have specification \( B \) and its implementation \( A \): we would then try to solve \( A \land \neg B \). In this sense, Rueher’s and Collavizza’s approach is similar to the process of resolution in logic. That is, we want to show that the implementation models the specification. That is,

\[
A \models B \quad \text{which is equivalent to} \quad A \land \neg B \models \bot
\]

The main steps of Collavizza’s and Rueher’s approach can be outlined as follows:

1) Translate the program into a constraint system. \((A)\)
2) Translate the negation of the specifications into a constraint system. \((\neg B)\)
3) Consider the conjunction of these two constraint systems as a CSP (possibly involving guarded constraints): \((A \land \neg B)\)

   a) If a solution is found, it means that the program does not meet its specification and the solutions to the CSP are the test cases that would fail to meet the specifications.
   b) If a solution is not found, it means that the program meets its specification.

### IV. OUR CONTRIBUTION

Our approach is based on the work of Rueher and Collavizza [5]. In this work, the CSPs generated from the code and the specifications may contain guarded constraints. Collavizza
and Rueher point out that standard CSP solvers may not be able to prune the system, and only after a costly enumeration process, the CSP solver is able to detect an inconsistency on the CSP. To deal with this disadvantage of standard CSP solvers, they propose using a SAT solver first. They introduce a Boolean variable for modeling conditional statements such as \( i < j \). Once the transformation is done, standard CSP solvers are able to detect the inconsistency.

In our work, we want to be able to handle programs involving floating points, i.e., representing real values. Since SAT solvers are not efficient for such problems, we want to consider an alternative approach so that there is no need for a SAT solver and it can be extended to handle domains of real numbers. The alternative we consider is that in the case of guarded constraints, we translate them by using the equivalence of logical implication with a disjunction. In the following subsection, we describe the main steps of our proposed approach.

### A. Algorithm of our proposed approach

Our proposed approach can be outlined as follows:

1. Translate if-then-else statements as specified by Collavizza and Rueher [5]. This step generates guarded constraints of the form \( A \rightarrow B \).
2. Transform guarded constrained of the form \( A \rightarrow B \) into their equivalent \( \neg A \lor B \). After Step 1, the CSP is in conjunctive normal form (CNF).
3. Transform the CSP in the form of a CNF into a disjunctive normal form (DNF), which is a disjunction of CSPs.
4. Solve the CSPs and consider the final solution to be:

\[
\bigcup_i \text{Solution}(C_i)
\]

In the following subsection, we present an example describing the steps outlined above.

### B. Example: Our proposed approach

Consider the example shown in Table 1 (also summarized in Figure 2. After translating the code and the negation of the specification we get the constraints shown in Table 2.

```java
// @ensures \text{\#results} \geq 0
public int absolute(int i, int j) {
    if (i < j)
        return (j - i)
    else return (i - j)
}
```

**TABLE I**
**EXAMPLE OF AN IF-THEN-ELSE STATEMENT**

If the code is correct with respect to its specifications, then we are aiming at the following CSP:

\[
((c_1 \rightarrow c_2) \land (c_3 \rightarrow c_4) \land c_5), D_i = D_j = D_r = \{0, ..., 65635\}
\]

which is expected not to have solutions.

However, we want to translate the guarded constraints into their equivalent (plain) constraints. After the transformation, we obtain the following new (equivalent) constraint system:

\[
\{ (\neg c_1 \lor c_2) \land (\neg c_3 \lor c_4) \lor c_5, D_i = D_j = D_r = \{0, ..., 65635\}\}
\]

(2)

Unfortunately, disjunctions of CSPs are difficult to deal with, thus we need further transformation.

Currently, we have a CSP in CNF, which involves disjunctions of constraints. We want to translate the CSP into a DNF because dealing with disjunction of CSPs is easier than dealing with disjunctions of constraints within a CSP. After this transformation on the CSP shown in Equation 1, we obtain the disjunction of the following CSPs:

- **CSP 1**: \( \{ (\neg c_1 \land c_3 \land c_5), D_i = D_j = D_r = \{0, ..., 65635\}\} \)
- **CSP 2**: \( \{ (\neg c_1 \land c_4 \land c_5), D_i = D_j = D_r = \{0, ..., 65635\}\} \)
- **CSP 3**: \( \{ (c_2 \land \neg c_3 \land c_5), D_i = D_j = D_r = \{0, ..., 65635\}\} \)
- **CSP 4**: \( \{ (c_2 \land c_4 \land c_5), D_i = D_j = D_r = \{0, ..., 65635\}\} \)

Now, we would have to solve these 4 CSPs and what we are aiming at is:

\[
\bigcup_i \text{solution}(CSP_i) = \emptyset
\]

However, in order to limit the computational complexity of our approach, we apply rules to reduce the number of CSPs to be solved. We now present these elimination rules.

**Challenges of our proposed approach** In our approach we face the following two challenges:

- Solving a union of CSPs.
- Solving conjunction of CSPs of the type \( \neg A \land B \).

In the following, we describe how to address both challenges.

### C. Elimination rules

Solving a union of CSPs means solving each of them one by one. Therefore our objective is to control the number of CSPs to be solved as much as possible. In our approach for each CSP involving guarded constraints, we generate 4 CSPs. However, we can eliminate two CSPs by applying the following rules:

- **Rule 1**: Eliminate CSPs containing \( C \land \neg C \). We apply this rule such CSPs have no solution. Moreover, if we were dealing with real domains, considering solving this
kind of CSPs could become a source of false positives and we want to reduce the number of false positives.

- **Rule 2:** Eliminate CSPs containing $C_{if} \land C_{else}$. We can apply this rule because of the semantics of the if-then-else statement.

By doing so, we reduce the explosion of CSPs to be solved.

**D. Solving constraints of the form $A \land \neg B$**

Consider that we have two constraints, $A$ and $B$, defining a relation on $x$ and $y$. $A$ and $B$ are represented as shown in Figure 3.

In the following, we propose two approaches to solving $\neg A \land B$.

**First approach**

We consider solving $A$ by an outer approximation. When performing such an outer approximation of $A$, we keep the discarded parts. The discarded parts constitute an inner approximation of $\neg A$. However, by doing so, there are two risks that we may run into:

1) we may miss some solutions; and
2) we may get false positives.

In the following, we identify and describe these risks.

**Risk of missing solutions**

Let us examine the risk of missing solutions by focusing on the region where this problem may arise, as indicated in Figure 5. After we solved $A$ by an outer approximation to get an inner approximation of $\neg A$, we solve $B$ on the inner approximation of $\neg A$. When solving $B$, we get an outer approximation of $B$, which is delimited in part by the inner approximation of $\neg A$. Note that we initially considered an outer approximation of $A$. Therefore, our approximation for $\neg A$ may not contain all solutions that in fact belong to $\neg A$. Since we may lose some
solutions of $\neg A$, and $B$ will be delimited by the approximation of $\neg A$, we have the risk of missing some solutions of $\neg A \land B$, colored in dark gray, as shown in Figure 6.

![Fig. 6. Risk in the first approach](image)

**Risk of false positives**

Now, let us examine the risk of having false positives. Note that when solving $B$, we computed an outer approximation of $B$. Therefore, we may get some solutions that originally did not belong to $B$. When solving for $\neg A \land B$, since we considered an outer approximation of $B$, we may have solutions that are not solutions, i.e., we may get false positives. These false positives are shown on Figure 6 colored in a black and white check board pattern.

**Conclusion**

One of our goals is to achieve completeness, i.e., not to miss any solution. Indeed, missing solutions may yield to the conclusion that the specification and the code are conform, while they are not, and could generate catastrophic situations. In this approach, correctness (i.e., no false positives) was not achieved, nor was completeness (i.e., no missed solutions).

A second approach, to address this problem, is presented hereafter.

**Second approach**

Let us recall that we want to solve $\neg A \land B$. In our second approach, instead of solving $A$ using an outer approximation and considering the discarded parts as solutions for $\neg A$, making it an inner approximation of $\neg A$, we now consider solving $A$ using an inner approximation and considering the discarded parts as solutions for $\neg A$, making it an outer approximation of $\neg A$.

**Risk of false positives**

Let us examine the risk of having false positives. Note that, in this approach, we only changed the way we solve $\neg A$. Therefore, we still have the same risk of false positives as in the first approach because when solving $B$, we still compute its outer approximation.

Furthermore, in the second approach, we may introduce more false positives. When solving $\neg A \land B$, we now consider an outer approximation of $\neg A$. As a result, we may get solutions that do not satisfy $\neg A$, and therefore, not $\neg A \land B$ either. The false positives that we may get on this example are shown on Figure 7.

![Fig. 7. Risk in the second approach](image)

**Discussion of the two approaches**

The two approaches we just described can be considered at different levels of risk: 1. The first approach involves the risk of missing solutions, and therefore to draw inappropriate / dangerous conclusions from the solving process. 2. The second approach is rather conservative, and there will not be any dangerous conclusion that the specifications and code conform together. On the other hand, there may be too many times when the code is rejected / not verified (at least in a first attempt), while it should not be.

Depending on the kind of targeted applications we are dealing with, one or the other approach may make more sense that the other one. In any case, the risk should be further quantified, in order to point out the gain / drawback and risk of using one approach or the other.

**V. Conclusion**

In this work, we considered the problem of verification of software. In particular, we were interested in proving that a program conforms to its specifications, using constraint programming techniques.

Our work extends and generalizes that of Rueher and Collavizza on the conformity of specifications and code. While their approach was handling programs dealing with computations on integers, our work aimed at also handling programs with floating-points computations. Besides, while they were making use of guarded constraints and of a SAT solver to detect inconsistencies early in the solving process, to avoid finding these inconsistencies after the costly enumeration process in normal CSP solvers, the use of SAT solvers is not efficient for handling programs dealing with computations on
floating-point numbers. In our work, we proposed an approach that does not include guarded constraints nor use a SAT solver. We pointed out the main two challenges related to our approach, and have extensively described algorithms to address one of them. In particular, two strategies to solve CSPs of the form $\neg A \land B$ were proposed and their properties pointed out. We also defined rules that prevent the computational explosion of our approach, hence addressing the second challenge as well.

As part of our future work, we will generalize our elimination rules to more complex combinations of statements, and show how we can control the growth of the problem size. We also plan to carry out a study of the gain observed in the number of false positives / false negatives between using our method with and without the elimination rules, and with the first approach or the second one. We anticipate that this study will not only allow us to draw conclusions regarding the level of risks, but will also help determine appropriate post-processing techniques to filter out false-positive results.

REFERENCES

A Comparison of Languages for Specifying Abstract Interfaces

I. H. Shah, Dept. of CS & IT,
Forman Christian College,
Lahore, Pakistan.

M. D. Tedd, Department of
Computer Science, University of
Wales, Aberystwyth, U.K.

Akram Naul, Dept. of CS & IT,
Forman Christian College,
Lahore, Pakistan.

Abstract: There is a dramatic increase in the extent to which software systems interact with other software systems. The need to create interface specifications of software systems so that they can be used from more than one programming language is frequently met. We have developed a technology for creating interface specifications of software systems in different programming languages. Our approach consists of writing specifications of interfaces at an abstract level and then automating the generation of their counterparts in different programming languages. These counterparts are referred to as bindings. This paper compares different languages for writing abstract specifications of interfaces to large software systems and justifies our selection of CDL (Component Description Language).

Key words: interfaces, bindings, CDL

1. Introduction

In order to conquer their complexity, the large systems are decomposed into interacting modules which can be designed and implemented separately. A module can be viewed as a set of objects and a set of operations on them; it serves as "a work assignment for a programmer or programmer team" [1]. A large system is developed by many people (perhaps at different sites); therefore, its development is not merely a programming problem but also a management problem. Parnas and Clements [2] report that "efficient and rapid production of software requires that the programmers be able to work independently ... [therefore,] a module interface specification must be written for each module". Shah and Tedd [3] compared different techniques for writing interface specifications and concluded that Abstract Specification with Programming Language Bindings was quite promising for specifying software systems. This paper compares different languages for writing Abstract Specifications of interfaces to large software systems and justifies the selection of CDL for this purpose.

2. Writing Abstract Specifications

The foremost requirement for this purpose is a suitable specification language. A language is suitable for writing an abstract specification of an interface only if it is capable of representing all facets of that interface. This section develops a model of an abstract interface specification, uses it to assess the suitability of Ada, ML, Z, the Larch Family of Languages, C++, Java and CDL to represent this model and finally selects a language most suitable for this purpose. The most critical facets of an abstract interface specification are the following; details about these facets can be found in [4, 5, 6].

2.1 Modularity

Modularity implies decomposition of a large system into modules so that each of them can be designed and implemented independently. A module usually realizes a single and simple conceptual function of the system. It therefore encapsulates data types related to that function and operations on them. In this way changes related to the definition and implementation of these types and to the operations on them are localized in this module. Each module is provided with a well defined interface for communication with the other modules. The interface to the system is structured from these interacting modular interfaces. Modularity directly supports modifiability, reliability, and readability.

2.2 Abstract Data Types

Liskov and Ziles [7] define an abstract type as "a set of objects capable only of a particular kind of behavior, which corresponds to a finite set of allowable operations on objects of the type". It is independent of its representation in the sense that details of how it is implemented are to be hidden from the user. The user is provided with certain operations and only needs to know what the operations are supposed to do, not how they do it. This allows modularization of programs to such an extent that implementations of abstraction can be changed underneath without the user knowing. It also localizes the effect of a bug at one place allowing relatively easy error correction because none of the places where the implementation is used makes any assumption how it is implemented. The use of ADT's also improves the portability of programs.

2.3 Semantics

The semantic specification of a software entity describes the meaning associated with that entity. It is desirable that the semantics of entities such as data types and operations and of error conditions be specified clearly so that the correctness of the system developed on the basis of this specification can be proved and controversial issues can be resolved during all the phases of subsequent development. The semantic description can be classified as:

(a) Formal description

In this case, the semantics of an entity are described in terms of logical expressions such as predicates, pre-conditions and post-conditions on the components of the entity. These expressions must be satisfied at the appropriate times.

(b) Informal description

The semantics are described in a natural language (please see documentation below in 2.13).
2.4 Behavioral Abstraction

Behavioral Abstraction is related to an operation and implies "making visible what an operation does and hiding how it does it". An example is a subprogram which exchanges information with its environment through formal parameters. Specification of an operation consists of the following two parts.

(a) Syntax specification consists of the name of the operation and the names and or types of formal parameters; the names of the formal parameters are required only in the implementation. In languages such as ML and C++, only the types of the formal parameters are required while in languages like Ada, names, types and even the modes of the formal parameters are required.

(b) Semantic specification describes the visible effects produced by an operation. It is better to capture the possible causes of failure and their recovery mechanisms in the behavioral abstraction of that operation rather than leaving them to the discretion of the implementer.

2.5 Formality

Formality implies description of both syntax and semantics of a specification entirely in a mathematically precise notation. Correctness of a program developed on the basis of a formal specification can particularly be proved by reasoning and this is perhaps the foremost requirement in safety critical systems.

2.6 Modifiability

Large software systems need to be modified during verification, validation, testing and maintenance phases. The modifications involving changes in the modular interfaces can affect some other modules and can be very expensive. For an interface to be modifiable the changes in one modular interface should affect a minimal number of modules of the system; this is possible only if the modules are as independent as possible.

It is also desirable to modify interfaces to some modules of the system and reuse them in specifying similar modules of the same or another system. In the words of Biggerstaff, et al. [8], "The modifying process is the life blood of reusability". The modifications may include renaming and deleting some entities.

2.7 Composability

Composability implies incremental construction of a module interface from the other primitive module interfaces, preferably the existing ones. Guttag et al. [9], emphasize that "it is essential that large specifications be composed from small ones that can be understood separately, and that the task of understanding the ramification of their combination be manageable" and Bardin and Thompson [10] stress the need for "Transitive Composability"; i.e. the capability of composite components to be further composable. It is cheaper and less error-prone as compared to starting from scratch.

2.8 Reusability

Reusability implies "the capability of a software component to be used again or used repeatedly in applications other than the one for which it was originally developed. In order to be effectively reused, the component may have to be adopted to the requirements of the new application" [11]. The two main advantages of reusing software are improved productivity and better quality. The benefits of reusing code are widely appreciated but Lens et al. [12] report that "reuse on the specification or design level offers benefits that complement those at code level". From their experience with the reuse they conclude that "reusable parts should be developed for reuse from the outset".

2.9 Reliability

A system is said to be reliable if it does not fail in exceptional circumstances. Things may go wrong due to many reasons such as violation of some limit or unexpected data. In some real time systems such as nuclear plants or systems controlling aircrafts, the results of errors can be disastrous. Identifying the exceptional circumstances and the desired behavior of each module in these circumstances should be part of the specification and design and must be done from the beginning.

2.10 Readability

It implies understandability. The specification of an interface is very often read to make corrections to it, implement it, test and maintain implementation, use and reuse the interface and to design a similar interface. Therefore, it is essential that the specification be understandable at least by a person familiar with the language used; it is most essential when a team of people is involved in the development of software. Ross et al. [13] point out that "understandability is not merely a property of legibility. Much more importantly, the entire conceptual structure is involved".

2.11 Overloading

Overloading; i.e. giving the same name to different entities contributes to simplicity and frees the specifier from the worries of inventing new unique names to avoid name clashes so that he/she can concentrate on the real problems of the interface.

2.12 Parallelism

"It should be possible to model the coexistence of the separate elements each obeying its own behavioral laws concurrently with others in the system" [6]; this can reduce the complexity of a system.

2.13 Documentation

Documentation describes in a natural language those aspects of an interface which cannot be captured by the formal notation provided by the specification language used. If a specification language does not support formal description of semantics the burden of defining semantics falls wholly on documentation. However, even if the specification language supports formal description of semantics, documentation is still required to
complement the formal notation. The PIMB Task Force on Abstract specification [14] also recommends "natural language description for the operations" in the PCTE abstract specification. Under these circumstances documentation becomes a vital part of an abstract specification. In the words of Hoare [15]: "The view that documentation is something that is added to the program after it has been commissioned seems to be wrong in principle and counter-productive in practice. Instead documentation must be regarded as an integral part of the process of design and coding". A simple textual comment mechanism is not enough when an abstract specification is to be mapped onto more than one programming language. In order to keep the documentation of an entity consistent with its syntax within all the bindings, there is a need to process the documentation along with the syntax of that entity.

2.14 Processing by Computer
It should be possible to process a specification by computer to:
• detect consistency and completeness errors,
• generate bindings in different programming languages,
• generate other objects such as intermediate representations.
Moreover, processing by computer is efficient and reliable.

3. Assessing Specification Languages
The following sections assess the suitability of a few specification languages on the basis of the model described above; for each language a brief introduction is followed by its assessment. In the case of programming languages that can also be used as specification languages such as Ada and ML and C++, we look at them from the specification point of view.

3.1 Ada
Named after Augusta Ada Byron, it is a high level programming language designed by a team led by Jean D. Ichbiah on behalf of the US Department of Defense (DoD) for developing software for embedded computer systems; i.e. process control systems, missile guidance systems, etc. Ada brings together the best features of the languages of that time (the late 70s) and embodies and enforces the principles of software engineering.

Assessment
3.1.1 Modularity. The packages support modularity. The packages are not instances of types and cannot be used as parameters, components of structures or values of pointer variables.
3.1.2 Abstract Data Types. Ada has a rich set of abstract data types; indeed, in the words of Brooks [16] "Ada is over-rich".
3.1.3 Semantics: Ada supports informal description of semantics.

3.1.4 Behavioral Abstraction. Ada has functions and procedures to capture the behavior of an operation.
3.1.5 Formality. Ada supports formal description of syntax only and does not support the formal description of semantics.
3.1.6 Modifiability. Packages and abstract data types support modifiability.
3.1.7 Composability. Packages and subprograms can be composed from existing ones; the nesting of package helps structuring a specification and keeping it in one textual unit.
3.1.8 Reusability. Generic packages support reusability. But there are "problems with importing through generic parameters"; the problems include "loss of relationship between subtypes and derived types". "Inability to import exceptions, record types, etc." and are discussed by Bardin et al [10].
3.1.9 Reliability: Ada provides partial support for reliability by providing a facility to define exceptions in specifications. But there is nothing to indicate which procedure raises which exception and when. There is also no formal mechanism to express recovery actions. Also, "exceptions cannot be specified as generic parameters" [17].
3.1.10 Readability. This was an explicit objective in the design of Ada. Properly indented Ada specifications using meaningful names for identifiers and comments where necessary are quite readable.
3.1.11 Overloading. Ada supports overloading.
3.1.12 Sequentiality and Parallelism. Ada supports both sequentiality and parallelism. "In Ada, parallel activities are described by tasks" [18].
3.1.13 Documentation. Ada has no mechanism apart from textual comments to handle documentation.
3.1.14 Processing by Computer. Ada specifications can be independently processed (compiled) by computer. But if a spec. B depends on another spec. A then A must be compiled before compiling B; this does not require the compilation of the body of A.

3.2 ML
ML[19] is a strongly typed impure functional language. It is impure because it allows side effects and its input / output is also of imperative nature. Being functional it is easier to prove the correctness of programs.

Assessment
3.2.1 Modularity. ML provides modules as abstract objects. But the modules cannot be nested; this leads to the specification consisting of many textual units in which all the operations are at top level.
3.2.2 Abstract Data Types. ML provides abstract data types. Definitions and representations of data types are in separate textual units; this provides another level of abstraction and supports top-down design. But
representation of data types is implemented using functions; this introduces an overhead.

3.2.3 Semantics. ML supports informal description of semantics.

3.2.4 Behavioral Abstraction. ML provides functions only (functional language) to capture the behavior of an operation.

3.2.5 Formality. ML supports formal description of syntax only and does not support the formal description of semantics.

3.2.6 Modifiability. Modules support modifiability.

3.2.7 Composability. Modules can be composed from existing modules using import mechanism. Standard ML provides a compose operator that composes two functions into a single function.

3.2.8 Reusability. Polymorphism supports reusability.

3.2.9 Reliability. ML provides mechanisms to raise and handle exceptions.

3.2.10 Readability. An ML signature is concise and being a programming language expression is readable for a programmer.

3.2.11 Overloading. ML does not support overloading.

3.2.12 Sequentiality and Parallelism. At present ML supports only sequential programs.

3.2.13 Documentation. ML has also no mechanism apart from textual comments to handle documentation.

3.2.14 Processing by Computer. The specifications written in ML can be animated.

3.2.15 Additional Feature.

3.3 Z

The Z [20] has been developed by the Programming Research Group (PRG) at Oxford University for writing formal specifications of computer systems. Z is based on typed set theory and predicate calculus. A Schema is a basic unit of specification in Z. "A schema groups together some declarations of variables and a predicate relating these variables". A schema has two parts:

- The signature part is a sequence of declarations of variables and functions. It should be noted that a function can be used as a data structure in Z.
- The predicate part describes properties of variables and the effects of operations on them and on the state of the system. It also describes the domains and ranges of the functions.

Assessment

3.3.1 Modularity. Schemas support modularity.

3.3.2 Abstract Data Types. Schemas are used to define abstractions including abstract data types.

3.3.3 Semantics. Z supports formal description of semantics.

3.3.4 Behavioral Abstraction. Operations are also defined using schemas.

3.3.5 Formality. Z is a formal specification language; both syntax and semantics can be expressed formally by schemas.

3.3.6 Modifiability. Schemas are usually loosely coupled; this supports modifiability of the entire specification.

3.3.7 Composability. Schemas can be combined with the help of logical operators such as conjunction and disjunction; in these operations the signature parts of the schemas are merged (variables common to all must have the same types) and corresponding logical operators are applied to their predicate parts. This facilitates construction of specifications of complex entities from those of their components. Z also provides a very good treatment for nesting; e.g. nested blocks in the symbol table for block structured languages like Ada, Pascal can be handled by sequences and specially defined operators.

3.3.8 Reusability. Generic schemas supports reusability. At a lower level schemas can be made partially available by filtering operations like schema projection and hiding.

3.3.9 Reliability. Schemas can express undesired events and the corresponding actions.

3.3.10 Readability. Z is not very readable; it can be understood only by a trained reader.

3.3.11 Overloading. Schemas having different contents can have the same names.

3.3.12 Sequentiality and Parallelism. Z supports sequentiality only.

3.3.13 Documentation. Z uses textual comments that can only be put between schemas but not in the schemas.

3.3.14 Processing by Computer. Specifications can be processed by computer to verify design by detecting errors at an earlier stage and to prove the correctness of implementations. It has also been attempted to generate Prolog programs from the Z specifications but this is slow.

3.4 Larch Family of Specification Languages

The Larch Project [9] at MIT’s Laboratory for Computer Science and DEC’S System Research Centre developed a family of specification languages and a set of tools, including language-sensitive editors and semantic checkers based on a powerful theorem prover to aid the productive application of formal specifications. The Larch technology is based upon a "two-tiered approach to specification". The Larch specification of each module has two components:

- specification of underlying abstractions,
- language-oriented behavioral specification (interface specification).
The abstractions are expressed in a language that is free from the programming language-dependent details; this language is called the Larch Shared Language. And the language-oriented behavioral specification is expressed in languages designed for specific programming languages, called the Larch Interface Languages.

The Larch Shared Language is common to all the programming languages and "is primarily algebraic". The shared language is used to define terms used in interface specifications.

"A trait is a basic unit of specification in the Larch Shared Language. A trait introduces operators and specifies their properties. Sometimes the collection of operators will correspond to an abstract data type".

The language dependent features of a component such as how this component communicates with its environment, side effects, exception handling are expressed in the corresponding Larch Interface Language. Each Larch Interface Language is designed for a specific programming language.

Most of the complexity is placed in the traits so that the interface specification is simple. The "operations specified in the interface specification are intended to be implemented by procedures but the operations of traits are not". Traits define theories that give meaning to the operators that appear in interface specifications.

Assessment

3.4.1 Modularity. The support for the modularity is usually available but it depends upon the corresponding Interface Language.

3.4.2 Abstract Data Types. Interface Language and Shared Language both support data abstraction; implementation language independent abstractions are expressed in the former and the rest in the latter.

3.4.3 Semantics. Shared language and the interface languages support formal description of the semantics.

3.4.4 Behavioral Abstraction. The facilities provided by the corresponding interface languages are used to capture the behavior of operations.

3.4.5 Formality. Formal description of both syntax and semantics is supported by the shared language as well as the interface languages.

3.4.6 Modifiability. Modules and traits support modifiability.

3.4.7 Composability. "The Larch languages are designed for incremental construction of specifications from other specifications".

3.4.8 Reusability. "The Shared Language traits are the principal reusable units in Larch".

3.4.9 Reliability. Exception handling is the responsibility of the interface languages which are modeled on the corresponding programming languages. This does not sound satisfactory.

3.4.10 Readability. "The Larch languages are designed to be readable" and the compositions mechanisms, defined as operations on specifications, are easy to understand.

3.4.11 Overloading. The Shared Languages do not provide this facility and Interface Languages depend on the underlying programming languages.

3.4.12 Sequentiality and Parallelism. Larch languages do not support parallelism.

3.4.13 Documentation. The Larch languages have no formal mechanism to handle documentation.

3.4.14 Processing by Computer. Larch specifications can processed by computer; tools aid different phases of developing a specification.

3.5 C++

C++ is an object oriented high level programming language designed by Stroustrup Bjarne[21]. C++ is a general purpose programming language but it has gained popularity for system programming.

Assessment

3.5.1 Modularity. C++ support modularity. The classes can be instantiated and instances of classes can be passed as parameters.

3.5.2 Abstract Data Types. C++ provides a rich set of abstract data types to handle various kinds of data.

3.5.3 Semantics: C++ support informal description of semantics.

3.5.4 Behavioral Abstraction. C++ has classes to capture the behavior of an operation / object.

3.5.5 Formality. C++ supports formal description of syntax only and does not support the formal description of semantics.

3.5.6 Modifiability. Classes, templates and abstract data types support modifiability.

3.5.7 Composability. Classes can be derived from existing ones; the nesting of a class helps structuring a specification.

3.5.8 Reusability. Classes and functions support reusability. The reusability is further enhanced by using templates.

3.5.9 Reliability: C++ provide support for reliability by providing a facility to define exceptions in specifications.

3.5.10 Readability. Properly indented and commented C++ specifications using good naming conventions for identifiers are quite readable.

3.5.11 Overloading. C++ supports overloading of functions and operators.

3.5.12 Sequentiality and Parallelism. C++ supports both sequentiality and parallelism.

3.5.13 Documentation. C++ have no mechanism apart from textual comments to handle documentation.
3.5.14 Processing by Computer. The specifications written using C++ can be independently processed (compiled) by computer. But if a specification B depends on another specification A, then A must be compiled before compiling B.

3.6 Java

The high level programming language Java [22] developed by the SUN Microsystems provides platform independence, supports object oriented programming and is used for internet applications.

Assessment

3.6.1 Modularity. Java classes support modularity. Classes can be instantiated, and passed as parameters.

3.6.2 Abstract Data Types. Java provides a rich set of abstract data types.

3.6.3 Semantics: Java supports informal description of semantics.

3.6.4 Behavioral Abstraction. Java has classes and methods to capture the behavior of an operation.

3.6.5 Formality. It supports formal description of syntax.

3.6.6 Modifiability. Classes and abstract data types support modifiability.

3.6.7 Composability. Classes and methods can be composed from existing ones; nesting of a class helps structuring the specification.

3.6.8 Reusability. Classes and methods support reusability.

3.6.9 Reliability: The facility to define exceptions in specifications provides support for reliability.

3.6.10 Readability. Properly indented Java specifications using good naming conventions for identifiers and comments are quite readable.

3.6.11 Overloading. Java supports overloading of methods but not operators.

3.6.12 Sequentiality and Parallelism. Java threads support both sequentially and parallelism.

3.6.13 Documentation. Java can record and generate documentation apart from comments.

3.6.14 Processing by Computer. The specifications can be independently processed (compiled) by computer.

3.7 CDL

The CDL (Component Description Language) [23] developed at Department of Computer science, University of Wales, Aberystwyth, “allows a component to be described in terms of interfaces which it provides and the interfaces which it requires; the language also functions as a module interconnection language by specifying how higher level components are to be constructed by connecting lower level components”[17].

Assessment

3.7.1 Modularity. Modules can be developed independently as unions or products.

3.7.2 Abstract Data Types. CDL provides abstract data types as interfaces which can be the components of other interfaces or parameters of operations.

3.7.3 Semantics. CDL describes semantics informally but treats the text describing the semantics specially.

3.7.4 Behavioral Abstraction. CDL’s functions and procedures capture the behavior of an operation.

3.7.5 Formality. CDL facilitates formal description of the syntax but not that of the semantics.

3.7.6 Modifiability. In addition to the typical support provided by modularity and data abstraction, CDL provides substitution and deletion mechanisms to modify interfaces. The side-effects of these modifications can be avoided by renaming the modified modules. An interface can also be enriched without redefining it; a new interface is defined which enriches the existing one.

3.7.7 Composability. New interfaces can be composed from the existing ones. CDL provides very simple composition mechanisms. An interface can also be modified during the composition.

3.7.8 Reusability. Reusability is one of the main objectives of CDL. Each CDL interface is reusable; substitutions can be made to adopt an interface to the requirements of the new application.

3.7.9 Reliability. CDL provides mechanisms to define exceptions; later an operation can be granted permissions to raise exceptions from these by using the raise qualifier. This version of CDL has no mechanism to capture the recovery actions.

3.7.10 Readability. Notation is easy and simple composition mechanisms make even the specifications of complex entities understandable.

3.7.11 Overloading. CDL supports overloading.

3.7.12 Sequentiality and Parallelism. This version of CDL supports the specification of sequential programs and not the concurrent ones.

3.7.13 Documentation. CDL did not provide any mechanism apart from textual comments to handle documentation.

3.7.14 Processing by Computer. A CDL specification can be processed by computer to detect errors, to generate bindings in Ada and C and to view the intermediate representation.

3.7.15 Additional Features. The information in an abstract specification is in general not sufficient and additional information is required to develop a working system. For example, in order to generate a binding of the specification in a programming language, more (language specific) information may be required; e.g. an operation or an exception can be meaningless in a
binding and should not appear in it. The CDL substitution mechanism can be used to substitute or delete an interface or a part of it. We, Shah and Tedd [24] made some extensions to the CDL. The most important extension is addition of a new feature to the language so that it can handle documentation of CDL expressions along with their syntax; others enrich the existing features of CDL.

4. Conclusion

In the light of above assessments and comparisons, CDL emerges as a marvelous choice for specifying abstract interfaces to large software systems. The extensions to CDL have further added to the proficiency of the language. We envisage CDL to be an ideal choice for specifying interfaces to the Web based applications.

5. References


[21] Stroustrup Bjarne, An Overview of the C++ Programming Language AT&T Laboratories, USA.


6. The authors

Ifitkhar Hussain Shah obtained M.Sc and Ph.D in Computer Science from the University of Wales, U.K. He is presently working as a Professor and Head of the Department of Computer Science & IT, Forman Christian College (A Chartered University) Lahore, Pakistan. Mike Tedd obtained his MA (Mathematics) at Cambridge University, England. After an early career in the computing industry, he joined the Department of Computer Science, University of Wales, Aberystwyth, where he became Professor and Head of Department. Muhammad Akram Naul obtained M.Sc in Computer Science from the University of Engineering & Technology, Lahore, Pakistan. He is presently working as an Assistant Professor of Computer Science, at Forman Christian College (A Chartered University), Lahore, Pakistan.
A Defect Typology to Support Software Process Verification and Validation

Trent Kroeger¹, Fengdi Shu², and Shaowen Qin³

¹ Defence and Systems Institute, University of South Australia, Adelaide, Australia
² Laboratory for Internet Software Technologies, Institute of Software, Chinese Academy of Sciences, Beijing, China
³ School of Computer Science, Engineering and Mathematics, Flinders University, Adelaide, Australia

Abstract - Approaches to software process development that are based on the principles and practices of software engineering have been advocated by many researchers as a means to produce software processes that meet the needs of process users. Such approaches require methodical and repeatable verification and validation techniques that enable effective and efficient identification, analysis and correction of software process defects. Defect classification schemes have proven to be effective for characterizing and improving the quality of software products. It is reasonable, therefore, to suggest that a similar scheme for the classification of process defects may provide comparable support to those performing software process engineering activities. This paper proposes a defect typology that can be used to classify software process defects as definition, asset, context or documentation problems. The paper then discusses potential applications of the typology to software process verification and validation and concludes by suggesting areas for further research.

Keywords: software process, process quality, defect classification, defect typology, software process verification and validation

1 Introduction

In recent years, a growing body of evidence has accumulated suggesting that software engineering organisations may derive benefits from defining, implementing and continuously improving their development processes [1-3]. To this end, many companies have invested significant resources in process improvement programs with the objective of implementing software processes that are appropriate to their business [4-7]. For these companies, software processes play a critical role in effectively supporting, guiding and, where necessary, controlling the execution of software engineering activities.

Given that the term ‘software process’ is defined and used in many different ways in the relevant literature and across industry, it is important to provide a clear definition as it relates to the research presented herein. For the purposes of this paper, therefore, a software process is:

A definition of partially ordered activities, with associated roles and input and output artifacts, and supporting process assets, intended to produce or enhance software products within a specified context

The definition above implies that the software process is a static entity that represents desired software engineering activities. This view can be contrasted with a common dynamic usage of the word ‘process’ that refers to the actual steps performed to achieve an outcome. It is also important to clearly distinguish between the terms of ‘artifacts’ and ‘process assets’. Process assets are typically documents such as procedures, guidelines and templates that provide support for those performing software process activities. By contrast, artifacts are those work products – for example software requirements, software design and source code - that are consumed, transformed or produced as a result of performing software process activities. Figure 1 further illustrates the nature of software processes by showing them in context with other elements of the software engineering system.

![Figure 1: A generic model of a human-centric software engineering system.](image-url)
A fundamental premise in any process-based software engineering approach is that the quality of the software process will influence the quality, cost and time-to-release of the software artifacts produced. Some researchers – most notably in [9] - have advocated process development approaches based on the principles and practices of software engineering. These approaches are referred to collectively as ‘software process engineering’[10]. The objective of software process engineering is to produce processes that meet the specific needs of users, largely software engineers and managers. To ensure that these needs are met, methodical and repeatable verification and validation techniques are required that can identify, analyze and correct software process defects effectively and efficiently.

Defect classification schemes have been applied successfully to software engineering as an effective means for characterizing and improving the quality of software products. It seems reasonable to suggest, therefore, that a similar scheme for the classification of process defects may provide useful support to verification and validation activities within the software process engineering life-cycle. Hence, this paper proposes a defect typology that may be used for classifying software process defects and discusses potential applications of the typology to support software process verification and validation.

The paper is structured as follows. Section 2 presents and discusses three software defect classification schemes that have been successfully applied to improve software product quality. Section 3 proposes a defect typology for software processes that is based on a classification scheme for software defects. Section 4 discusses how the proposed defect typology may be used to support software process verification and validation activities. Finally, sections 5 and 6 suggest areas for further research and summarize the key contributions of the paper.

2 Software Defect Classification Schemes

Systematic analyses and comparisons rest upon principles of classification. Within the software engineering community, defect classification schemes are used to quickly and consistently characterize the nature of software defects. This information can then be used to assess the quality of software products and processes, with the aim of identifying problems and potential improvements. Software defect classification and analysis has become critical to improving productivity and software quality with a number of defect classification schemes applied successfully in practice. Three such schemes include: IBM’s Orthogonal Defect Classification[11], Hewlett Packard’s Defect Origins, Types and Modes[12], and the IEEE Standard Classification for Software Anomalies[13].

The goal of Orthogonal Defect Classification (ODC) is to provide an in-process measurement paradigm to extract key information from defects and to enable the metering of cause-effect relationships. The attributes of ODC are organized according to two process steps: open and close. The former is performed when a defect has been detected and a new defect report is opened in the defect tracking system; the latter is performed when the defect has been corrected and the defect report is closed. Activity, Trigger, and Impact are attributes of the opener section. Target, Defect Type, Qualifier, Source, and Age are attributes of closer section. Among these attributes, Defect Type and Trigger are specific to ODC and the defect types provided by ODC include function, assignment, interface, checking, timing/serialization, build/package/merge, documentation, and algorithm errors. Although ODC was originally developed for software design and code phases, there have been a number of extensions[14].

The purpose of Hewlett Packard’s (HP) Scheme is to improve the development process by reducing the number of defects over time. A primary focus of the model is that all defects are not created equally. The developers use this scheme by selecting three descriptors for each defect: the origin of the defect (where the defect was injected in the system), the type of defect, and the mode of the defect (why the defect was injected in the system). The choice of a value for the attribute Origin, defines the possible set of values available (and reasonable) for the Type attribute. Within the HP scheme, the following types of defect are recognized: requirements/specifications, functionality, hardware interface, software interface, user interface, and functionality description. The HP process puts emphasis on defect prevention and early defect detection in the front end of development[15].

The IEEE Standard Classification for Software Anomalies (IEEE 1044:1993) provides a uniform approach to the classification of anomalies found in software and its documentation. The processing of anomalies discovered during any software life-cycle phase are described, and comprehensive lists of software anomaly classifications and related data are provided. The different attributes of the scheme are organized according to a general defect classification process consisting of four steps: recognition, investigation, action, and disposition. The types of anomaly identified in the scheme are logic problem, computation problem, interface/timing problem, data handling problem, data problem, documentation problem, document quality problem, and enhancement. This standard is aimed for audiences who want to implement a defect classification scheme compliant to a standard or who want to expand a defect tracking or defect classification scheme and are looking for proven methodologies supporting that effort.

Comparisons made between these schemes, such as in [16], show that each has strengths and weaknesses relative to the others. For example, the attributes of the HP scheme are not independent and the selection of values in one attribute influences the selection of values in a second attribute, while ODC and IEEE 1044:1993 describe a classification scheme, in which values are assigned independently for each attribute; the advantage of the latter approach is that new attributes are able to be added more easily [17]. Furthermore, comparative analysis of these schemes yields the following set of generally desirable properties that apply to any defect classification scheme: orthogonal attributes and orthogonal attribute values, complete attribute values, small
number of attribute values, description of attribute values[17], consistency across phases, and uniformity across products[11].

Osterweil[9] suggests that software processes are themselves a form of software and that there are considerable benefits that will derive from basing a discipline of software process development on the more traditional discipline of application software development. Therefore, it seems reasonable to suggest that a similar scheme to software defect classification be developed to support software process verification and validation. Analogy will be applied. However, since there are distinct differences between the defects in software and processes, it is unlikely that existing software defect classification schemes can be applied to software processes in their current form. For example, many existing software defect types are concerned primarily with defects in software source code, which does not necessarily have a direct analogy in software process. Also, there are important differences between activities and phases relating to defect insertion, detection and correction for software and software processes. Given these differences, research into software process defect classification is needed to adapt software defect classification schemes to the specific needs of software process engineering.

3 Classification of Software Process Defects

The position taken in this paper is that the concepts and methods for software defect classification may similarly be applied to software processes. Software classification schemes specify several defect attributes that may be used to characterize defects, for example activity, trigger, type and impact. Typically, defects are then further classified within each attribute according to a set of predefined values. The type attribute, is central to all classification schemes and most directly characterizes the nature and cause of identified defects. For these reasons, this research is focused initially on developing a scheme to classify software process defects based on type. It is envisaged that future research activities will seek to expand this typology into a comprehensive defect classification scheme with a range of attributes.

An important first step in this research is to carefully define the term ‘defect’ as it applies to software processes. For the purposes of this research, a software process defect is defined as:

*Any condition that causes or is likely to cause deviations from expectations based on software process requirements specifications, architecture descriptions, design documents, standards, etc. or from someone’s perceptions or experiences.*

This definition is derived from [13] and reflects a broad interpretation of the term defect to include both problems inherent in the process itself and problems with the process as perceived within the context of its use. It is important to note that since processes themselves are static entities, a process defect is characterized only by its potential to cause or contribute to anomalies in the software engineering system and is in this way more similar in nature to a software design defect than a defect in operating software.

The approach taken to create the process defect typology involved analyzing each type classification within the IEEE standard 1044-1993 and, where possible, identifying analogous concepts relating to software process. The IEEE standard was chosen as the basis for this work because it provided the most general representation of software defects out of the three schemes considered. The development of the typology was guided by six general principles: (1) the typology should be consistent with the software engineering system model shown in Figure 1, (2) defect types should be orthogonal, (3) the set of defect types should be as complete as possible, (4) as few defect types as possible should be defined, (5) defect types should be defined clearly and consistently, and (6) defect types should allow for consistent interpretation across different phases and processes.

Figure 2 provides an overview of the developed typology for software process defects. The typology identifies four distinct types of software process defects: definition problems, asset problems, context problems and documentation problems.

These defect types are grouped into two categories: *product problems* and *engineering problems*. Product problems are those defects found in the process itself that, if not addressed, will impact process users. Engineering problems are defects that are found in software process engineering artifacts, such as process requirements or design. Engineering problems may or may not lead to defects in the process.

A software process definition problem is a defect relating to the existence, content or ordering of defined process activities. For example, definition problems are often uncovered during software process assessment, where it is found that a defined process does not include activities that are widely considered to be best-practice. Definition problems can be further classified according to the following six subtypes:

- **Missing activities**: the process does not define one or more required activities
- **Incomplete activities**: the definition of one or more activities is not complete
- **Incorrect activities**: the process defines one or more erroneous activities
- **Incorrect flow**: the defined ordering of the process activities is incorrect
- **Ambiguous activities**: the purpose of one or more process activities is unclear
- **Extraneous activities**: the process defines one or more unnecessary activities

A software process asset problem is a defect in the information provided to support the performance of a given process activity. For example, consider the situation of a team performing a standard peer review process where review criteria were unavailable. In this situation all of the activities for the peer review may be performed in the correct order, but the lack of review criteria may lead to an ineffective result.
Asset problems can be further classified according to the following six sub-types:

- **Missing guidance**: one or more expected process assets do not exist
- **Incomplete guidance**: one or more process assets contain incomplete information
- **Incorrect guidance**: one or more process assets contain erroneous information
- **Ambiguous guidance**: one or more process assets contain unclear information
- **Conflicting guidance**: two or more process assets are inconsistent
- **Irrelevant guidance**: one or more process assets contain unnecessary information

A software process context problem occurs when there is a mismatch between the process and other elements of the overall software engineering system. An example of this type of defect would be if a very detailed and prescriptive process was introduced to an organization that had intentionally developed a working culture based on the principles of flexibility and autonomy. Context problems can be further classified according to the following six sub-types:

- **Input mismatch**: the process defines inputs inconsistent with those available
- **Output mismatch**: the process defines outputs inconsistent with those required
- **User mismatch**: the process is inconsistent with the skills or culture of its users
- **Technology mismatch**: the process is inconsistent with one or more technologies
- **Process mismatch**: the process is inconsistent with one or more other processes
- **Business mismatch**: the process is inconsistent with the business environment

A documentation problem is a defect in one or more software process engineering documents that will likely lead to the introduction of a defect to the process. Examples of this type of defect are any errors that are found during the review of the software process requirements, architecture or design. Documentation problems can be further classified according to the following six sub-types:

- **Missing items**: one or more expected documents do not exist
- **Incomplete items**: one or more documents are unfinished
- **Incorrect items**: one or more documents contain erroneous information
- **Ambiguous items**: one or more documents contain information that is unclear
- **Conflicting items**: two or more documents contain inconsistent information
- **Redundant items**: one or more documents contain unnecessary information

While the defect typology has been developed using sound principles and is derived from an existing classification scheme, it is difficult at this stage to comment on its validity. Ultimately, a determination of whether the typology is valid can be made only after it has been used to good effect in an industrial setting. The following section discusses potential applications of the typology to software process engineering that, if implemented, will allow for such a determination to be made.
4 Applications to Software Process Verification and Validation

We begin this section by making a brief comparison between the impact of software defects and process defects. Taking the example of software design defects, it is well understood that problems of correctness and completeness with respect to requirements, internal consistency, and redundancy, etc. can directly affect the quality and effort associated with the design implementation. Significant amounts of verification and validation is therefore planned and conducted to find and remove these design defects. Similar defects can and do occur in software processes, but organisations typically apply much less formal verification and validation approaches. As a result, process defects often persist throughout the life-cycle and are discovered only after deployment. Considering that the main objective of software process is to improve the efficiency and quality of software production, it seems natural that significant attention should be paid to discover and remove defects in processes before they are deployed.

Similar to the role software defect classification plays in software engineering, it is believed that the process defect typology proposed in this paper will facilitate the verification and validation of a software process throughout its life-cycle, thereby improving its quality in a more methodical and consistent manner.

Figure 3 depicts a simplified software process engineering life-cycle. Although shown as a linear sequence, it is understood that a process may proceed iteratively and incrementally through such a cycle a number of times from its inception to retirement. Although there has been some discussion of the software process engineering life-cycle [18, 19] in the literature, there does not as yet seem to be a widely accepted description. Hence, the life-cycle depicted has been developed based on the authors’ experience as well as using software engineering life-cycles as an analogy.

A comparison between Figure 3 and a typical waterfall software development life-cycle shows that testing phases are noticeably absent in the process life-cycle, and verification and validation, which typically involve a combination of peer-review during development and process assessment after deployment, are put in place as the primary means to achieve better process quality. This reiterates the intrinsic difficulty associated with the fact that processes are interpreted and executed by humans instead of computers, meaning that intensive and thorough testing prior to deployment is usually impractical. Advances in software process simulation provide a promising solution to this problem, particularly for relatively low-level processes.

It is envisaged that the software process defect typology proposed in section 3 can be used in the following ways to directly improve the effectiveness of software process verification and validation activities:

1. The typology can be used as a peer-review checklist to diagnose defects in process assets and supporting engineering documentation. The typology provides specific and practical guidance for defect identification, with each type focusing the process engineer on a certain aspect of the process. To exploit this, roles may be established in multi-person peer-reviews to cover all aspects of the process.

2. The typology can be used as reference criteria for internal and external process assessments. Current assessment approaches focus primarily on confirming the existence of certain process assets and evidence that indicates the performance of a set of recognized best-practices. The additional reference criteria provided by the typology may help to identify a wider range of process defects than would otherwise have been discovered.

3. The typology can be used to characterize and sort process defects for the purposes of facilitating causal analysis. Causal analysis of this defect data may lead to the identification of potential improvements to both the software processes and software process engineering activities.

Apart from the direct applications listed above, the proposed process defect typology also provides a much needed common terminological framework that can be used by process engineers, process users, process assessors and managers to communicate process related issues in a precise and consistent way.

5 Further Work

Given that software process defect classification research is at a preliminary stage, many opportunities for further work exist. The following three areas of work have been identified to verify, validate and extend the proposed classification scheme:

1. Further verification of the proposed typology is required to ensure that the scheme is complete and able to be applied in a repeatable way. One way to conduct this verification would be to have different people classify a large sample of software process defects identified within an industrial setting. Statistical analysis may then be performed to show whether the proposed attribute values are
sufficient and whether the classification scheme can be applied consistently.

2. It is also necessary to validate the use of the proposed defect typology in an industrial setting to show that it provides the benefits expected. This may be achieved by conducting a number of controlled trials within software engineering organisations and collecting a range of quantitative and qualitative information to determine the costs and benefits of applying the classification scheme.

3. Finally, to develop the typology into a complete defect classification scheme, it will be necessary to extend the scheme to include attributes other than defect type. One way to achieve this would be to adapt other attributes found in software defect classification schemes using an approach similar to that described in this paper for developing the defect type attribute.

6 Conclusions
Defect classification schemes provide an effective means for characterizing and improving the quality of software products. The position taken in this paper is that a similar scheme for the classification of process defects may provide comparable benefit for the evaluation and improvement of software processes. To this end, a defect typology has been presented that can be used to classify software process defects as definition, asset, context or documentation problems. Guidelines have also been presented for the potential application of this typology to support software process verification and validation activities, such as process asset peer-review, process assessment and causal analysis of process issues. In conclusion, this paper establishes that it is both possible and desirable to construct and apply a defect type classification scheme for software processes. Furthermore, this preliminary work indicates that software process defect classification represents a promising line of research worthy of further investigation.

ACKNOWLEDGMENT
The work of Fengdi Shu is partially supported by the National Natural Science Foundation of China under grant No. 60573082, the National Hi-Tech Research and Development Plan of China under Grant No. 2007AA010303. And the work was conducted during her 6 months visit to Flinders University, Australia in 2007/08.

REFERENCES
SESSION

SOFTWARE TECHNIQUES

Chair(s)

TBA
A GARBAGE COLLECTOR PROTOTYPE FOR C++ APPLICATIONS BASED ON MARK AND COMPACT USING ASPECT ORIENTED PROGRAMMING

Hamid Mccheick, Aymen Sioud, Abdenour Bouzouane
Department of Computer Science and Mathematics, University of Québec at Chicoutimi, 555 Boul Université G7H 2B1, Chicoutimi, Canada
{Hamid_mccheick, Aymen_sioud, abdenour_bouzouane}@uqac.ca

Rakan Mcheik
Department of Computer Science, Institut des sciences appliquées, Beirut, Lebanon
riko_81@hotmail.com

Keywords: Garbage collector, object-oriented applications, aspect oriented programming.

Abstract: We are investigating one of the claims of C++ object applications and its lack of garbage collectors. Therefore, an object memory management in C++ legacy systems can be automatically developed and integrated more easily than other kinds of object management techniques: explicit (manual). For instance, research effort has been done to improve object memory management technique, such as incremental garbage collector, reference counter, conservative garbage collector, smart pointer, and so on. Theses techniques have some limitations, such as amalgamated functional and technical aspects in the same object oriented programs and its implementation has to be done manually. Generally speaking, we need a mechanism and tool to i) separate object lifecycle management from functional aspects and ii) implement this task automatically and implicitly. We propose a method and develop a tool to manage object lifecycle management in C++ systems using the generational mark and compact technique and therefore to eliminate implicitly storage management bugs. Our approach is developed based on aspect-oriented programming and AspectC++™, in which we define an aspect as a software artefact that addresses a concern such as memory management in C++ applications.

1 INTRODUCTION

In wide-enterprise information system, an object lifecycle management of programs can be handled using one of the following strategies: implicit or explicit (Alexandrescu, 2001), (Bohem, 2002), (Detlefs, 1992), etc. In some programming languages such as Java, Lisp, and Eiffel, the garbage collection handles implicitly and transparently the object lifecycle. Automatic (implicit) garbage collection (GC) provides two advantages (Bohem, 2002): i) the programmer does not provide code to explicitly allocate memory and ii) so there is no danger of accidentally reusing an object’s memory while the object can still be accessed. However, in C++ and Ada languages, object lifecycle management has been done explicitly and manually under the direction of the programmer using new and delete operators (Jones, and Lins, 1996). The explicit management of object lifecycle requires a considerable effort on the behalf of the programmer. Consequently it leads to mysterious errors and bugs: dangling pointers, allocation failures, bugs, etc. (Horstmann and Budd, 2005). Nevertheless, its advantageous for allowing direct manipulation of pointers for optimality, efficiency, and performance reasons.

Meyer has classified automatic memory management at the third position in his seven commandments for object-oriented software development (Meyer, 1988). Ellis and Detlefs have proposed a collection algorithm and code-generator to safely integrate explicit GC in C++ application. This approach needs a change to code generators to ensure correct operation (Ellis, and Stroustrup, 1990). They introduced some properties that we
have to consider in GC such as Minimal changes, Coexistence, Safety, Portability, and Efficiency. Detlefs also has specified the following constraints to integrate a useful GC in a C++ application (Detlefs, 1992): i) GC requires no compiler support, ii) it requires no information from the programmer about the format of GC objects, iii) it has strategies to cope with bugs caused by aggressive optimizing compilers, iv) it allows the use of both automatically and explicit managed storage in the same program, and v) it invokes destructors on called heap objects. We take into account these constraints without any changes of existing class libraries and C++ implementations. In addition, an implicit and automatic GC has been integrated. These constraints will be discussed in section 4.

Several researches such as smart pointer and reference counter have been done to transform explicit memory object lifecycle management to a dynamic implicit memory object lifecycle management. Most of the object lifecycle management approaches in C++ have two major limitations: i) a code client has to inherit from a specific lifecycle management class, and ii) the developer has to implement this task manually at the same time of developing the application. Smart pointer technique uses the reference counter technique: one of the used strategies to object memory lifecycle management (Alexandrescu, 2001). However, reference counter wastes many times to manage new and delete operators.

Also, it is interesting to be able to handle object lifecycle management as a concern. In this way, the object lifecycle management will be decoupled from functional aspects (Kiczales et al., 1997), (Mcheick, 2006). To improve the modularity in object-oriented programs, there have been a number of approaches providing language-level support for separation of concerns in the OO research community, including aspect-oriented programming (AOP) (Kiczales et al., 1997), view-oriented programming (Mili, et al., 1999), and many others. We use AOP and its descendent AspectC++™ tool to provide an implementation of generational mark and compact technique for C++ applications.

Actually, C++ is one among the mostly used programming languages for its efficiency. It is an attractive language for several project types such as, real time systems, compiler optimizers, and performance intensive projects (i.e. computer game projects). Also, the optimizations of actual compilers make C++ as an interesting language especially to increase application performance. The C++ language uses explicit memory operators (delete, new) and the operations (free and malloc), for managing memory. For Ellis and Stroustrup (1990), implicit memory management via garbage collectors is not part of the C++ language.

In order to clearly understand the concepts of an explicit garbage collector, theses concepts will be explained through the example in Figure 1, in which construction and destruction of objects have been used. The main function uses new and delete operations to manage explicitly memory. It is clear that the programmer might make some mistakes resulting in bugs and forget to call a delete operation: the programmer does not make delete(p2) at the end of main function. The part of the code which is responsible for memory management can be implemented as an aspect to delete unused object.

```cpp
class Point {
    int X; int Y;
    ...

    Point (int NewX=0, int NewY=0){
        X=NewX; Y=NewY;
    }

    void showInfo(){
        cout << X << '
' << Y;
    }
};

int main(){
    Point *p1 = new Point(10,10);
    p1->showInfo();
    delete (p1);
    Point *p2 = new Point(200,200);
    p2->showInfo();
}
```

Figure 1 : Example of class Point using new and delete operations.

In this paper, we design and present a tool which implicitly manages the memory allocation and release of C++ programs. This tool makes use of AspectC++™ (AspectC++, 2005), an aspect oriented programming pre-processor, to implicitly implement memory management. The implementation of implicit memory management is realized based on generational mark and compact algorithm via AspectC++™. The implementation marks accessed objects in the first step, then compacts them at the second step and finally updates (also verifying promotion threshold of objects) the reference pointers. The inaccessible objects are destroyed. Furthermore, each iteration, the objects can be promoted.

The advantage of this approach is that it relieves the developer from explicitly coding memory management operators and operations. Another advantage is that using that tool, memory leaks can
be prevented and handled by the implicit garbage collector generated by the GMCAspect tool and appended to the application. Another advantage of this approach consists of solving cycle management and removing fragmentation. The GMCAspect tool uses AspectC++™ to generate and append implicit garbage collection code to any C++ application.

The next section includes a brief overview of garbage collection techniques. Section 3 describes the GMCAspect tool and its implementation as well as an example. Section 4 describes the result and the limitations of this tool. We conclude in section 5.

2 BACKGROUNG

We describe the various garbage collector techniques and separation of concerns.

2.1 Garbage Collector Techniques

Reference Counter and other techniques. Several techniques have been used to manage object lifecycle. We distinguish three approaches: object reference counter, leasing technique, and evictor technique (Henning, and Vinoski, 2004). The object reference counter is used by COM/DCOM (Frank, 1997). This technique consists of maintaining a counter of the number of objects referring to a shared object. This counter is incremented every time a new reference to the shared object is created (Jones, and Lins, 1996). Also, this counter is decremented each time a reference to the shared object is destroyed. The implementation of the reference counter technique is relatively easy, but has several problems such as the reference counter incoherence resulting from poor management of client failure: if we associate reference counter to a client and the client failed, his reference counter still not empty and the objects, referred by that clients, will never be collected by the garbage collector. Leasing technique consists of handling a contracts between client and server on which the objects reside. Leasing is used by RMI (Pitt et McNiff, 2001), JINI (Tanenbaum and Van Steen, 2002), DCOM de Microsoft (Frank, 1997) and CORBA (OMG, 1995). Its problem consists of manage the leases of objects. Evictor technique is based on a heuristics (i.e Least Recently Used and Least Frequently Used).

Generational Mark and Compact. This algorithm is based on the principal of the temporal locality which consists of: a) if an object is present in memory since a long time, there is many chances to still alive, b) however, if an object is just created, there many chances to be deleted soon.

This algorithm is one of the most used garbage collection because its fast, avoiding memory fragmentation and corruption. Its idea is based on mark and compact the objects: we mark all objects starting from the stack until every alive objects are marked. Once it is done, we copy all the marked objects. In summary, this algorithm has three phases: i) mark the alive objects, ii) compact all the marked objects, and iii) update the pointers of these objects. There are three methods to compact the objects : a) compacting arbitrary, where the objects are copied without carrying about the order, b) a linear compacting where we keep adjacent objects in the same structure, c) slide compacting where the objects are copied in a lower level and in the same order. To update the pointers, we distinguish four methods : a) Two fingers algorithm (Bartlett, 1989) where two pointers are used, one for the next free space and the second for the next object to be copied, b) Table based algorithm (Knuth, 1973) where we use a table to calculate the new addresses, c) Threaded method (Fisher, 1975) where we use a list for the objects in the original order, and d) Forward addressing algorithm (Jonkers, 1975) where we associate for each object the next object address. It is been proved that most recently created objects are most likely to quickly become unreachable: infant mortality (Jones, and Lins, 1996), (Wilson, 1994). Therefore, if an object is alive for many cycles, we assume that it is less likely to be deleted and so, we move it to an older generation heap, which is less frequently scanned for unreferenced memory. This algorithm is used, at least, in two platforms J2EE and .Net. In particular, Java virtual machine Hotspot for servers uses generational mark and compact algorithm (Sun Microsystems, 2006). This algorithm will be used to develop our tool.

2.2 Aspect-oriented techniques

Aspect-oriented programming (AOP) recognizes that the programming languages that we use do not support all of the abstraction boundaries in our domain models and design processes. Underlying AOP is the observation that what starts out as fairly distinct concerns at the requirements level, or at the design requirements level (non-functional requirements) end us tangled in the final program code because of the lack of support, both at the design process level, and at the programming language level, for keeping these concerns separate. With aspect-oriented programming, these concerns may be packaged as aspects, which can be woven into “any” application that has those concerns.
AspectC++™ is a general-purpose aspect-oriented programming extension to the C++ language (Kiczales et al., 1997) and (AspectC++, 2005). It is a pre-processor for a regular C++ compiler. After performing aspect weaving, AspectC++™ outputs standard C++ code.

In the example of Figure 2, an aspect which calculates the number of created object instances. In order to calculate the number of occurrence of “new”, we use the pointcut: call("new").

AspectC++™ adds new constructs to integrate new codes, such as joint point, pointcut and advice. Joint points are points in source code where aspect code should interfere. A joint point is a pattern recognized by aspect weaver, such as new operator. A point cut is used to identify a collection of joint points.

Advice code can be thought of as an action activated by an aspect when a corresponding join point in a program is reached. The activation of the advice code happens before or after the code join point is reached. The AspectC++™ language element to specify advice code is the advice declaration. It is introduced by the keyword advice followed by a pointcut expression defining where and under which conditions the advice code shall be activated.

iii. around(). is a function through which the advice code can explicitly trigger the execution of the program so that the advice code can be executed before and after the join point. The execution of the program code at the join point is achieved by calling the proceed() (AspectC++, 2005) function within the body of the around advice code.

In the following code chunk, an example of advice, is given to introduce two attributes in any class (flag and count). The following aspect wrapping_object declares these attributes:

```cpp
aspect wrapping_object {
private:
    advice classes("%")bool flag=false;
    advice classes("%")int count=0 ;
}
```

We can see how the aspect wrapping_object can be used to mark all visited objects in the mark step with flag attribute. The count attribute represents the number of the garbage collection visitor. These two variables are used in the section 3.

The figure 2 shows the integration of aspect “Print_name_function” with class Course. AspectC++™ weaver integrates these components (aspect and class) in class Course. It injects two lines of code in two functions of class Course: assign_teach(...) and assign_student(...). This aspect writes the signature of these functions using jointpoint construct.

### 3 ASPECT GARBAGE COLLECTION TOOL FOR C++ APPLICATIONS: GMCASPECT

To manage object lifecycle in C++ legacy applications, we propose a tool (GMCAspect) based on AspectC++™ and generational mark and compact algorithm. Figure 3 shows a C++ legacy application and GMCAspect tool as inputs. AspectC++™ uses a weaver to integrate C++ applications and object lifecycle management into a C++ standard applications. In this section, a concept, implementation and structures of GMCAspect is presented.
3.1 Concept of GMCAspect

The idea of generational mark and compact Garbage Collector (GMCAspect) consists of developing an automatic implicit technique to manage object lifecycle in C++ applications. This management is important to support performance criteria which are needed in different domains. To manage object lifecycle in C++ applications, GMCAspect implements the generational mark and compact Garbage Collector approach, which consists of applying mark and compact starting by the first generation until we get the space needed. When an object reaches the promotion threshold, it promotes to the next generation (see background explanations). While this algorithm gets rid of the fragmentation, it implies that you double the memory you need for a short moment which may be a problem in some case. It has been proved that most recently created objects are most likely to quickly become unreachable. As a conclusion, we use this algorithm to develop a generational mark and compact garbage collection (GMCAspect) for C++ applications. So to avoid unnecessary movement in the memory, we use separated memory for each generation (see background explanations). While this algorithm gets rid of the fragmentation, it implies that you double the memory you need for a short moment which may be a problem in some case. It has been proved that most recently created objects are most likely to quickly become unreachable. As a conclusion, we use this algorithm to develop a generational mark and compact garbage collection (GMCAspect) for C++ applications.

Traditionally, garbage collector traverses a graph of objects using registers and classes as roots. We traverse the code using classes as roots. We use DFS (depth first search) to go through the graph of the objects to mark reached objects and to update the number of the garbage collector visitor. Then we use the slide compact method and we copy the object in the same generation or in the next generation depending on that number. Finally, to update the object pointers, we use the Threaded method (Fisher, 1975). This method needs a list of objects to keep their orders.

In next section, we explain the implementation of GMCAspect.

3.2 Implementation of GMCAspect

The generational mark and compact Garbage Collector (GMCAspect) idea consists of developing an automatic implicit technique.
int *i = new int();
printf("Creation: Simple1\n");
Simple* simple1 = new Simple();

printf("Creation: Heritante\n");
Heritante* heritante = new Heritante();

printf("Creation: Simple2\n");
Simple* simple2 = new Simple();

printf("Creation: Composed (contient un Simple)\n");
Composed* composed = new Composed();

printf("Delete: simple2\n");
delete simple2;

return 0;
}

Figure 4: Example to be used by GMCAspect to illustrate this tool.

Figure 5 shows the result of the first call of the GMCAspect to parser the code of figure 4. It shows in particular the different objects.

4 RESULTS AND LIMITATIONS

Aspect Garbage Collector (GMCASPECT) has three advantages. First, GMCAspect does not require any intervention of the user on the objects handled during development. Also, this tool does not require any modification of C++ legacy applications. However, smart pointer technique and others techniques require changing C++ code that must inherit from SmartPointable class, which is developed in smart pointer library. Secondly, GMCAspect allows an implicit object lifecycle management. In addition, it prevents memory leaks. Thirdly, an implicit memory management tool for C++ has to handle the memory management of any object types. Smart pointer does not handle constant table and pointers. However, GMCAspect handles most objects except templates classes, since AspectC++™ pre-processor cannot handle template classes.

GMCAspect has the following limitations which are due to AspectC++™ limitations such as i) the inability to load some header files (AspectC++, 2005) because of PUMA configuration, which is an integrated tool in AspectC++™, ii) the inability to handle template classes (such as cin, cout, etc.), and to manage the creation of int, char, etc. Briefly speaking, our tool does not take into account header files seen the incapacity of AspectC++™ to load certain header files and it manages only objects created by the user or the legacy applications.

Our tool GMCAspect verifies the majority of the constraints emitted by Detlef, (1992). However, GMCAspect as well as that based on Generational Mark and Compact is not tested yet using a large number of C++ applications to show how much it improves their performance. On the other hand, GMCAspect manages only objects created by the user. So, the tool does not take care of system objects created by the compiler, as those created for namespaces. In fact, it is important to improve this tool to take care about all objects and to make a comparison between both tools from the view point of temporal and spatial consumption.

5 CONCLUSION

In this paper, we presented the problem of memory lifecycle management in C++ based applications. Traditionally, memory lifecycle management has been done explicitly and manually in C++ applications. Several research studies have been done to improve object lifecycle management, such as smart pointer and more others, which have to be integrated manually.

We proposed GMCAspect (generational mark and compact Garbage Collector), a tool to manage object lifecycle management in C++ systems based
on the generational Mark and Compact technique. This method and tool are developed based on aspect-oriented programming and AspectC++, in which we can define an aspect as a software artefact that addresses a concern. In this paper, we have described the concept of GMCAspect, its implementation, which handles C++ programs implicitly and transparently without user intervention. This tool relieves the developer from explicitly coding memory management operators and operations. Also, memory leaks can be prevented and handled by the GMCAspect tool and appended to the application. Our approach solves cycle management and removes fragmentation. This tool shows also that it is possible to use simultaneously an implicit and explicit garbage collector for C++ applications. In future work, we investigate the problem of memory management lifecycle where the object can be created by systems and for any type of pointers and templates classes. This task is obviously a more complicated issue in C++ applications.

ACKNOWLEDGEMENTS

REFERENCES

Cay Horstmann and Timothy Budd, 2005. Big C++, John Wiley and Sons, USA.
Bohem H.J., Bounding Space Usage of Conservative Garbage Collectors. POPL'02 (ACM), Jan 16-18, 2002 Portland, OR, USA.

APPENDIX
Implementing Readers/Writers Problem
Using Aspect-Oriented Framework

P. Netinant\(^1,2\)
\(^1\)Computer Science Department, Bangkok University, Bangkok, Thailand
\(^2\)Concurrent Programming Research Group, Illinois Institute of Technology, Chicago, IL, USA

Abstract – Programming in a distributed environment is a complex activity. Programmers need to be aware of issues unrelated to their domain of problem, and are often unprepared for the challenges the concurrent programming brings. The interaction of their components becomes more complex, and makes it difficult to validate the design and correctness of the system. Supporting separation of concerns in the design and implementation of operating systems can provide a number of benefits such as comprehension, reusability, extensibility and adaptability in both design and implementation. We have tackled this problem by adopting the technique of separation of concerns in concurrent programming. In this paper we demonstrate an Aspect-Oriented Framework (ACL) that can be used for system software such as operating systems. We also show how the separation of system aspectual properties from components. Readers/Writers problem is demonstrated using our framework. Our framework, which is based on aspect-oriented technology as well as language and architecture independence, is a component based model.

Keywords: Adaptability, Aspect-Oriented Programming, Framework, Operating Systems, Reusability.

1 Introduction

The principle of separation of concerns lies at the heart of software development as it introduces a number of benefits, originally addressed by [8, 2]. These include better understanding, extensibility, adaptability [3] of the system, and better reuse of the concerns. Although these benefits have been well established, there is still no universally accepted methodology in order to guide a programmer to best achieve separation of concerns. Concerns are divided into system and application level. Operating systems consists of separating multiple concerns crosscutting many components of the system. Systems are notorious of many crosscutting concerns such as synchronization, scheduling, fault tolerance, logging, and etc. We refer to these crosscutting concerns as system aspectual properties. Supporting separation of concerns in the system can provide a number of benefits such as comprehension, reusability, extensibility and adaptability for system and application software. In both the design and implementation of the operating system, the system designer has to consider how a number of aspects can be captured, and how a separation of concerns [8] will be addressed. Functional decomposition has so far been used as well as achieved along two dimensions - based on the components and layering paradigm. In OOP, these dimensions are layers and components; included methods, objects and classes. Current programming languages and techniques have been supportive to functional decomposition. However, languages are specific domain. Moreover, operating system design has also been aligned with traditional functional decomposition techniques. No functional decomposition technique has yet managed to address a complete separation of concerns. Object-Oriented Programming (OOP) seems to work well only if the problem can be described with relatively simple interfaces among objects. Unfortunately, this is not the case when we move from sequential programming to concurrent and distributed programming. As distributed systems become larger, the interaction of their components is becoming more complex. This interaction may limit reuse, make it difficult to validate the design and correctness of operating systems, and thus force reengineering of these systems either to meet new requirements or to improve the system. Certain system aspectual properties of the system do not localize well. They tend to crosscut groups of components or services (functions or methods) in the system. System aspectual properties tangle in components or services making the system difficult to understand and adapt. Changing needs to understand and correctly identify both system aspectual properties and core service implementation of the component or service. It is tightly couple design and implementation between components and system aspectual properties.

2 System Aspectual Properties

System aspectual properties are, for instances, mutual exclusion, scheduling, synchronization, fault tolerance, security, load balancing, performance measurement, testing, verifications and etc. They are all expressed in such a way that tends to crosscut groups of components or services. A tangling code of system aspectual properties results increasing of code dependencies between components and properties of the system. It makes their source code difficult
to understand, reuse, adapt, and maintain. One current attempt to resolve this issue is the Aspect-Oriented System (AOS). AOS aims at language and architecture independence, where components and system aspectual properties are separately decomposed in both design and implementation. These properties can be reused and adapted in the application later. Finally, components and system aspectual properties are combined together at run-time. We distinguish between components and aspects in the design of systems. System aspectual properties are defined as properties of the system that do not necessarily align with the system's components or services but tend to cut across groups of functional components, increasing either inter-dependency or intra-dependency, and thus affecting the quality of the software. Intra-dependency defines as a system aspectual property that crosscuts between many services (functionalities or methods) in the same components. Inter-dependency defines as a system aspectual property that crosscuts between many components or services. Although not bound to OOP, Aspect-Oriented Programming (AOP) [4, 5] is a paradigm proposal that retains the advantages of OOP and aims at achieving a better In this paper we have shown system design and implementation based on system aspectual decomposition in the context of the aspectual decomposition in the design of operating systems. Our approach is an aspect-oriented framework [6, 7]. Compared with what has so far been able to be supported by traditional approaches, our goals are to provide a better design and implementation for operating systems, better flexibility, higher reusability and adaptability, as well as to provide a technique that would be practical.

3 A Framework for Concurrent Systems

Our observation suggests that an Aspect-Oriented Systems (AOS) that uses Aspect-Oriented Framework could support designers and programmers in cleanly separating components and system aspectual properties from each other. Our framework is based on Aspect-Oriented techniques and layered approach [1]. We argue that system aspectual properties of the operating system should be excluded from the system components or services if there is a possibility to often change it, and it should not be treated as a single monolithic aspect. Our proposed framework (ACL) is based on system aspectual decomposition of crosscutting concerns in operating system design and implementation. Our framework consists of two frameworks: Based Layer and Application Layer Framework. CAL framework consists of two frameworks: Based Layer and Application Layer Framework.

The aspect-oriented framework supports both vertical and horizontal compositions. Functional and aspectual property components in the framework can be composed vertically or horizontally. In vertical composition, the upper layer can use the lower functional or aspectual property components from the lower layer. In horizontal composition, functional and aspectual property components in the particular layer only use to be composed.

The framework is based on system aspectual decomposition of crosscutting concerns in operating system design and implementation. The framework consists of two frameworks: The Based Layer and The Application Layer Framework. A system aspectual property is implemented in the SystemAspect class, while a component of the system is implemented as a Component class. Alike AspectJ [9], our framework uses PointCut, Precondition, and Advice. The AspectModerator class, where the point cut is defined, combines both system aspectual properties and components together at runtime. Pointcuts are defined collections of join points, where system aspectual properties will be altered and executed in the program flow. Every aspectual property could join points. Advice could be either before or after. Before advice could be implemented as blocking or non-blocking. Before advice executes when join point is reached, before the component executed, and if the precondition is hold. After advice executes after the component at the join point executes.

4 Architecture of the Framework

Our observation suggests that an Aspect-Oriented Systems (AOS) that uses Aspect-Oriented Framework could support designers and programmers in cleanly separating components and system aspectual properties from each other. Our framework is based on Aspect-Oriented techniques and layered approach [1]. We argue that system aspectual properties of the operating system should be excluded from the system components or services if there is a possibility to often change it, and it should not be treated as a single monolithic aspect. Our proposed framework (CAL) is based on system aspectual decomposition of crosscutting concerns in operating system design and implementation. CAL framework consists of two frameworks: Based Layer and Application Layer Framework.

The aspect-oriented framework supports both vertical and horizontal compositions. Functional and aspectual property components in the framework can be composed vertically or horizontally. In vertical composition, the upper layer can use the lower functional or aspectual property components from the lower layer. In horizontal composition, functional and aspectual property components in the particular layer only use to be composed.
component at the join point is executed. Every aspectual property will define advice methods. The execution model of a pointcut in the framework is based on inter-dependency and intra-dependency. In this paper, we show how readers/writers problem can be implemented using the framework. A system aspectual property is implemented in SystemAspect class, while a component of the system is implemented as Component class. Alike AspectJ [9], our framework uses PointCut, Precondition, and Advice. AspectModerator object, where the point cut is defined, combines both system aspectual properties and components together at run-time.

Pointcut is defined collections of join points, where system aspectual properties will be altered and executed in the program flow. Every aspectual property could identify and implement precondition. Precondition is defined a set of conditions or requirements that must be hold in order to be executed an aspect. Advice is defined collections of methods for each aspectual property that should be executed at join points. Advice could be either before or after. Before advice could be implemented as blocking or non-blocking. Before advice executes when join point is reached, before the component executed, and if the precondition is hold. After advice executes after the component at the join point executes. Every aspectual property will define advice methods. Our proposed framework (CAL) is based on system aspectual decomposition of crosscutting concerns in operating system design and implementation. CAL framework consists of two frameworks: Based Layer and Application Layer Framework. In this paper, we show how producers/consumers problem can be implemented in the based layer framework. A system aspectual property is implemented in SystemAspect class, while a component of the system is implemented as Component class. AspectModerator object, where the point cut is defined, combines both system aspectual properties and components together at run-time. A Pointcut is defined collections of join points, where system aspectual properties will be altered and executed in the program flow. Every aspectual property could identify and implement precondition. Precondition is defined a set of conditions or requirements that must be hold in order to be executed an aspect. Advice is defined collections of methods for each aspectual property that should be executed at join points. Advice could be either before or after. Before advice could be implemented as blocking or non-blocking. Before advice executes when join point is reached, before the component executed, and if the precondition is hold. After advice executes after the component at the join point executes.

5 Implement of the Framework

The framework consists of four components comprising the architecture of the framework.

- Each functional object (component) provides its services (methods) stripped of any aspectual properties (for example, no synchronization is included in Buffer objects).
- A proxy object intercepts called methods and transfers the calls to the AspectModerator.
- An AspectModerator object consists of the rules and strategies needed to bind aspects at runtime. Aspects are selected from the AspectBank. The AspectModerator orders the execution of aspects. The order of execution can be static or dynamic. Then, each precondition will be checked whether it is satisfied or not.
- An AspectBank object consists of aspect objects that implement different policies of a variety of aspects.

This section presents the design and development of aspect-oriented framework. The model is presented to demonstrate horizontal composition of the framework. The system service must be implemented as a Component class. The system aspectual property (SystemAspect class) must be derived from the SystemAbstractAspectFactory interface to implement the required behavior of a system aspectual property. A SystemAspectFactory consists of many system aspectual properties such as synchronization, tracing, logging, and reliability. The SystemAspectFactory, derived from the SystemAbstractAspectFactory interface, is known as an aspect bank.

During runtime, each SystemAspectFactory will be associated with one SystemAspect. The AspectModerator class must be derived from the AspectModerator interface to implement the required behavior. The following points are important about the aspect-oriented framework:

- A base layer framework is an implementation of an underlying system.
- An application layer framework is an implementation of application software over the system software represented by a base layer framework.
- A client object requests a service through a ProxyObject object of a framework.
- A functional component is implemented as a Component class without any aspectual property.
- A SystemAspectFactory object consists of various SystemAspect objects. A SystemAspect object is controlled by a SystemAspectFactory object.
- Each system aspectual property must be implemented as a SystemAspect object.
- Each crosscutting between Component object and a SystemAspect object must be defined in AspectModerator object as joinpoints in a Pointcut method.
- A client requests a service by sending a message to a ProxyObject object. The ProxyObject object changes the request to a specific pointcut method, and forwards it to the AspectModerator object.
The Proxy class is responsible for intercepting and forwarding the message sent from Client object to request a service. The Proxy class must implement the behavior of intercepting a service request. A client object of an aspect-oriented framework must request a service by calling the call() method. A call() method consists of at least two parameters: object name provided a service and a service requested to serve. The first parameter is of type string, and the second is type of string as well. The ProxyObject class will forward a request to the AspectModerator object by calling a PointCut() method. A PointCut() method must have the same number parameters and the same parameter type as the call() method.

The SystemAspectFactory class must be derived from the SystemAspectFactoryAbstract interface to implement the required behavior. The AspectModerator class is responsible for composing the functional components and the system aspectual property into a service request. The AspectModerator class acts like a coordinator between functional components and system aspectual properties, when and where system aspectual properties will be composed into a functional component. The composition of system aspectual properties and functional components must be guided and defined as PointCut() method. Each PointCut() method must have at least two parameters: component name and service name (methods of the component) that will be composed. The first parameter is of type string, and the second is type of string as well.

The AspectModerator class will create the SystemAspectFactory object. The SystemAspectFactory object can support either static or dynamic aspects at runtime. The attachImpl() method is used to associate a system aspectual property of a SystemAspectFactory object. The AspectModerator class will be associated with functional components and system aspectual properties that will be composed. The PointCut() method will define join points between functional components and system aspectual property. Currently, the PointCut() method uses if...then...else... statements to define joinpoints. The synchronization aspect property crosscuts both read and write services. It crosscuts the before and after execution of read and write services. A tracing property crosscuts both read and write services. It only crosscuts the after execution of read and write services.

The SystemAspectFactor class must be derived from the SystemAspectFactoryAbstract interface to implement the required behavior. The SystemAspectFactory class provides a dynamic binding of variety system aspectual properties. It focuses on the interface of the system aspectual property. Each system aspectual property must be derived from the SystemAspectAbstract interface to implement the required behavior. Implementation of a system aspectual property is implemented in the SystemAspect class. Each system aspectual property can define before(), after(), and precondition() methods depending on its needs.

The AspectModerator class operates composition between system aspectual properties and functional components using a composition rule defined by join points of a pointcut. The AspectModerator class performs composition rules by sending AspectFactory messages. Messages sending causes polymorphism. The implementation of AspectFactory uses bridge patterns. A message finds the correct member object of the AspectFactory, and invokes that object. With polymorphism calls, AspectModerator requires less information about each SystemAspect, so the AspectModerator only needs to have the right SystemAspect interface.

The abstract aspectual class defines a SystemAbstractAspect interface that controls the implementation of an aspectual property class. This class is implemented using the concrete classes of aspectual properties, which implement the virtual functions before() and after(). The AspectModerator creates instances of an aspectual property, which requires composing a requested service. If an aspectual property crosscuts more than one method in the same component, it must have a parameter ServiceName identifying what it should be done for each method. If an aspectual property crosscuts more than one component, it must have two parameters: ServiceName and ComponentName identify what it should be done for each method of each component.

6 Example Readers/Writers Problem

There are two kinds of processes - readers and writers, which share file(s). Readers execute the requests that request accessing to and reading of the sharing file(s). Writers execute the requests that request accessing and writing to the sharing file(s). A file is assumed initially to be in a consistent state, where the data in the file is meaningful and not corrupt. Each request transforms the file(s) from one consistent state to another. Writers’ requests must have exclusive access to the file.

The client/server relationship is a usual pattern in distributed systems where files are sharing resources among clients and controlling by server. A client process can be either a reader or a writer. A client process requests services; read or write to the sharing file(s), then waits for the request to be handled by a server providing the services. A server process repeatedly waits for a request, serves it, then sends the result to the requested client. Figure 1 represents the scenario of the readers/writers problem.

This section shows the experiments of two distinct solutions to the readers/writers problem. The first solution approaches it as a regular programming problem using object-
oriented techniques. This solution is longer and appears to be more complex and harder to modify and understand.

Figure 1. Readers/Writers as a Client/Server Relationship

The second solution uses the aspect-oriented framework in their implementation. The second solution approaches it as an aspect-oriented programming with separating functional components from aspectual properties. It is easy to understand implementation of the solution, but more work needs to be done. However, the second solution can easily be modified to implement different aspectual properties and add functional components between readers and writers. The aspect-oriented framework introduces a powerful approach.

6.1.1 Readers/Writers Problem as Object-Oriented Programming

This section presents the implementation of the Object-Oriented Approach for the Readers/Writers Problem using the C++ language. The implementation consists of three classes; ReaderObject, WriterObject, and BufferObject. Client objects are the WriterObject and the ReaderObject classes.

Figure 2. Collaboration Diagram of Classes

A ReaderObject process must make a request to read a file before reading the file and a WriterObject process must make a request to write a file before writing the file from a BufferObject object as illustrated in Figure 2. The BufferObject object will grant permission either to the WriterObject or to the ReaderObject processes one after other until all processes are done. Access to the shared file is controlled by the BufferObject object using semaphore objects in the C++ language as illustrated in Figures 3 and 4. The ReaderObject objects must call the ReadFile(file) operation to access reading the file. The WriterObject objects must call the WriteFile(file) operation to access writing the file.

From Figures 3 and 4, there are aspectual properties (e.g. synchronization and tracing aspects) crosscutting methods ReadFile() and WriteFile() operations. These aspectual properties make the code of the ReadFile() and the WriteFile() method tangled. This means that a developer will find it hard to understand, develop, maintain, and reuse the code.

```cpp
int BufferObject::ReadFile(const DWORD &dwThreadId)
{
    DWORD dwSemaphore;
    DWORD dwMutex;
    dwMutex = WaitForSingleObject(hMutex, INFINITE);
    cout << "Get Acquired the mutex: " << dwThreadId << endl;
    iReader++;
    if (iReader == 1) {
        dwSemaphore = WaitForSingleObject(hSemaphore, INFINITE);
        cout << "Get Acquire the Semaphore: " << dwThreadId << endl;
    }
    cout << "Released the mutex: " << dwThreadId << endl;
    ReleaseMutex(hMutex);
    cout << "Thread Id: " << dwThreadId << " reading file..." << endl;
    Sleep(1000);
    WaitForSingleObject(hMutex, INFINITE);
    cout << "Get Acquired the mutex: " << dwThreadId << endl;
    iReader--;
    if (iReader == 0) {
        cout << "Release the Semaphore: " << dwThreadId << endl;
        ReleaseSemaphore(hSemaphore, 1, NULL);
    }
    cout << "Released the mutex: " << dwThreadId << endl;
    ReleaseMutex(hMutex);
    return 0;
}
```

Figure 3. ReadFile() Operation by BufferObject Class

When we analyze the implementation of the Readers/Writers problem, we found that there are two aspectual properties crosscutting the functional components; reader, writer, and buffer components. Those aspectual properties are the synchronization and tracing aspects.

An aspectual property can be divided into intra-crosscutting and inter-crosscutting concerns. Intra-crosscutting concerns crosscut methods in the same object. Inter-crosscutting concerns crosscut methods in different objects. However, the synchronization and tracing aspects might be inter-crosscutting concerns if they crosscut between methods of the different objects. For example, the buffer provides two files. Changing the tracing aspectual property needs to go through both WriteFile() and ReadFile() operations because it is tightly coupled to both operations.
Figure 4. WriteFile() Operation by the BufferObject Class

Therefore, the synchronization and tracing aspects crosscut methods of two objects. Figure 5 illustrates the crosscutting concerns of the Readers/Writer problem. These aspectual properties can be extracted and captured in a localized object using the aspect-oriented approach.

Figure 5. Intra-Crosscutting Concerns of the Readers/Writers Problem

In the next section, we will present the implementation of Readers/Writers problem using the aspect-oriented framework, which makes an implementation cleaner and better.

6.1.2 Readers/Writers Problem as Aspect-Oriented Framework

This section presents the implementation of the Aspect-Oriented Approach to the Readers/Writers Problem using the C++ language. The implementation consists of nine classes; ReaderObject, WriterObject, FileBuffer, ProxyObject, SystemAbstractAspect, SystemAbstractFactory, SystemAbstractAspectFactory, SynchronizationAspect, and AspectModerator. Client objects are WriterObject and ReaderObject classes.

Figure 6 illustrates the extraction of crosscutting concerns from the buffer object. Each aspectual property will be put into one aspect object. Each aspect will be defined before and after advices (method calls).

The horizontal composition must be derived from the AspectModerator Class. The ProxyObject is responsible for intercepting and carrying out the requested messages from the ReaderObject and the WriterObject classes. In the aspect-oriented framework, an AspectModerator class will compose any aspectual property with the FileBuffer class to produce one abstraction. The FileBuffer class is a shared file object, which provides the ReadFile(file) and the WriteFile(file) operations. The SystemAbstractAspect class is an abstract aspect class, where a different implementation of an aspectual property is implemented in the SynchronizationAspect class. The SystemAbstractFactoryAspect class is an abstract class of SystemAbstractAspect. The SystemAbstractFactory class is an abstract class of SystemAbstractFactoryAspect. The overall architecture of the Readers/Writers problem after extracting crosscutting aspectual properties from the functional component, make a buffer object cleaner and better separation of concerns. The AspectModerator is added to weave aspectual properties and functional components at runtime.
The framework separates functional components and aspectual properties from each other. This technique makes the design and implementation of the Readers/Writers problem easier to understand, reuse and adapt. Change of the functional code or aspect code does not affect each other. Adding of functional components or aspects requires new PointCuts to be defined in the AspectModerator object. The implementation of the Readers/Writers problem using the aspect-oriented framework is cleaner and better separation crosscutting concerns. Figure 7 illustrates the implementation of PointCut as a method in the AspectModerator class.

```c
int AspectModerator::PointCut(const char * sFuncObject, const char * sFuncNamen)
{
    if (strcmp(sFuncObject, "FileBuffer") == 0) {
        if (strcmp(sFuncName, "WriteFile") == 0) {
            sysAspectFactoryObj[0].before(sFuncName, obj);
            sysAspectFactoryObj[1].before(sFuncName, obj);
            functionalObject.WriteFile(obj);
            sysAspectFactoryObj[0].after(sFuncName, obj);
            sysAspectFactoryObj[1].after(sFuncName, obj);
        }
        else if (strcmp(sFuncName, "ReadFile") == 0) {
            sysAspectFactoryObj[0].before(sFuncName, obj);
            sysAspectFactoryObj[1].before(sFuncName, obj);
            functionalObject.ReadFile(obj);
            sysAspectFactoryObj[0].after(sFuncName, obj);
            sysAspectFactoryObj[1].after(sFuncName, obj);
        }
    }
    return 0;
}
```

**Figure 7.** The Implementation of PointCuts in the AspectModerator Class

# 8 References


An Improved Story card Based Requirement Engineering Practice for Extreme Programming

Chetankumar Patel, Muthu Ramachandran
Innovation North, Faculty of Information and Technology
Leeds Metropolitan University, Leeds, UK, LS6 3QS

Abstract – Developing software that meets the customers or stakeholders’ needs and expectation is the ultimate goal of the software development methodology. To meet their need we have to perform requirement engineering which helps to identify and structure requirements. In traditional software development methods end users or stakeholders predefined their requirements and sent to the development team to analysis and negotiation to produce requirement specification. In many cases it is risky or very difficult and not economical to produce a complete, verifiable set of requirements. Traditional software development has a problem to deal with requirement change after careful analysis and negotiation. This problem is well tackled by the XP as XP recommends an on-site customer to represents their requirements through user stories on story cards. Generally customers have rarely a general picture of the requirements or system in their mind which leads problems related to requirements like requirements conflicts, missing requirements, and ambiguous requirements etc, and does not address non-functional requirements from exploration phase.

Keywords: Story Cards, User Story, Agile Requirements, Agile requirements elicitation.

1 Introduction

Developing software that meets the customers or stakeholders’ needs and expectation is the ultimate goal of the software development methodology. To meet their need we have to perform a requirement engineering step, which is one of the crucial steps in to software development methodology. Overall project success and failures of the project is depending on the user requirements. Requirements elicitation process is one of the challenging processes in the software development methods. In traditional software development methods end users or stakeholders predefined their requirements and sent to the development team to analysis and negotiation to produce requirement specification. Traditional software development has a problem to deal with requirement change after careful analysis and negotiation. This problem is well tackled by the XP, which is one of the agile software development methodologies.

Extreme (XP) programming is a conceptual framework of practices and principles to develop software faster, incrementally and to produce satisfied customer. It is a set of twelve practices and four principles, which makes XP successful and well known among all the agile software development methods. The goal of XP is to produce the software faster, incrementally and to produce satisfied customer [1]. According to Bohem the cost of change grows exponentially as the project progresses through it lifecycle [2]. The relative repair cost is 200 times greater in the maintenance phase than if it is caught in the requirement phase [3]. XP maintain the cost of change through iterative software development methods and Refactoring.

In XP, Development starts with planning game where customer writes user stories on story cards. Those cards are estimated by the developer, based on those estimation customer priories them depends on their needs to establish a timebox of an iteration. Developers develop those story cards through pair programming and test driven development. At last customer provides acceptance test to accept the developed functionality. In between they consider all of the XP practices in mind to improve the quality of the software.

Story cards are one of the important aspects of the XP. They are playing vital role in XP. It describes functionality of system or software to be build that will be valuable to either purchaser or user of software. User stories are composed of three aspects [5]:

- A written description of the story used for planning and as a reminder
- Conversation about the story that serves to flush out the details of the story
- Tests that convey and document details and that can be used to determine when a story is complete

Story cards are written by the customer in XP to articulate their business needs. According to Cohn story cards must be testable, estimatable, valuable to the customer, small and independent [5]. These story cards must be written by the customer because they know their business need very well compared to developer.

XP strongly recommend an onsite customer to write their business need. Business is well understood by the customer. But generally customers have rarely a general picture of the requirements or system in their mind [6]. Traditional XP story card framework or template is not well defined for the requirements elicitation. It supports to write requirements or
user needs in two to three sentences and it not discover any information rather than user functionality. Different stakeholders have different needs. End user has rarely a picture of a clear of system to write down the user stories. This will lead to problems related to requirements like requirements conflicts, missing requirements, and ambiguous requirements etc, and not address non-functional requirements from exploration phase. Due to this reason they hardly make a decision or predict wrong priority of the requirements or story cards. Two third of the projects are failed because of ambiguous and incomplete user requirements, and poor quality of the requirements. For small to medium organizations, proper requirements prioritization and selection can mean the difference in not only project success or failure but also overall company survivability [14]. Different users have different perspective of system in same organization. Different background can make problems to priories the requirements in XP. Different stakeholders have different needs and different requirements and different prioritization values so requirements conflicts there. A critical aspect of the requirements process is the selection of the an appropriate requirements set from the multitude of competing and conflicting expectation elicited from the various project stakeholders or from an onsite customers [9]. The CHOAS report published in 1995 shows that almost half of the cancelled projects failed due to a lack of requirements engineering effort and that a similar percentage ascribes good requirements engineering as the main reason for project success [13].

As a result of this investigation we propose a new prototype to improve requirement elicitation process in XP. This will help to customer and developer to improve the quality of the user stories or story cards, and to address functional and non-functional requirements on story cards based on the story cards and requirements engineering guidelines. We also propose an ‘INSERT’ model or methodology to perform requirements engineering in XP. This article compares, traditional requirement engineering and traditional XP requirements engineering approach with our new improved ‘INSERT’ technique to capture user requirements for agile software development environments. We also analyze commonalities and differences of both approaches and determine possible ways how agile software development team and customers can benefit from our improved requirements elicitation methods. In this paper we discuss about extreme programming and requirement elicitation process through story cards first and then after the challenges and problems on XP software development methodology. This is followed by discussion of related research regarding to an ‘INSERT’ requirement elicitation method of the story cards for XP based projects.

2 An ‘INSERT’ Process in Extreme Programming

XP methodology highly relies on the onsite-customer interaction with the developer to identify or to tell which features to implement in a next release. XP builds software systems based on customer’s domain knowledge and his/her expertise. In traditional XP or agile software development methodology story cards are written by the customers perhaps with the help of developer. Therefore each user story or story card must be split into unique and independent requirement.

Traditionally user story is written on the 2” X 3” Cards. Usually any size is acceptable to write user story on story cards. Following Figure 1 shows an example of the story card used in the real project as proposed in (Beck 2000) which provides a traditional structure of story card.

Figure 1 Traditional Story Card (Back 2000).

According to general template of story card, it is really difficult to well tackle the user requirements expressed on the cards. Traditionally developed story cards are not providing enough information of user functionality. They express user functionality in single to couple of sentences, which is really difficult to do analysis, and they lead problem related to under or over estimate, and based on that estimation there is a possibility of wrong story cards prioritization as well. We apply this traditional story card method to the real project user story Figure 2 shows the story cards written through a traditional way, which is just a short and vague statement of user requirement. This story card does not provide any information related to acceptance testing as well.

Figure 2 an Example of Traditional Story Card

This is a story card for an online e-commerce store. In this story cards user trying to express their requirement as

‘Each and Every online purchaser needs to register with unique username and password before purchasing anything from the online store’
As a result of this investigation we propose a new ‘INSERT’ model to improve the quality of user story and to address customer requirements properly and on verifiable way the acronyms INSERT is as:

I: Independent
N: Negotiable
S: Small enough to fit into iteration
E: Estimatable or easy to Estimate
R: Representation of user functionality (Requirement)
T: Testable

2.1 Independent

Story cards must be independent or we have to take care as much as possible to avoid dependencies between story cards. Dependencies between story cards lead a problem related to estimation and prioritization. For example customer selected a high priority of story cards which is depend on low priority story cards, this situation make estimation harder than it suppose to be. In our insert model we do take care of dependencies between story cards. We take care as much as possible to flush out the dependencies between story cards. This type of story cards mostly captures atomic requirements which are directly transferable to a single use case and a design class.

2.2 Negotiable

Stories on Story card also be negotiable. Story cards are short description of the user requirements they are not working as a contract. User story is a reminder to have a conversation between developer and customer. If it is negotiable than only it gives better chance to developer to understand customer needs, their business need and their domain knowledge as well. This type of story cards mostly captures complex requirements which relates to more than one used cases and scenarios.

2.3 Small

Stories on the story cards need to be small enough to fit into iteration. Too big story or too small story is not going to fit into the timebox of iteration. To solve this problem we suggest to write an acceptance test with the story it self. There is a direct co-relation between story and acceptance tests. If story many acceptance tests that means it is big to fit into iteration and needs to be split into two story cards based on the acceptance tests. If story is small on the story card then combine them with another small story to fit them into the same iteration. This type of story cards captures a part of a independent and negotiable requirements and may also represent a non-functional requirements.

2.4 Estimatable

Estimation is a crucial value of the story card. Based on the developer’s estimation customer decide which functionality is going to be first and which one is next. On traditional XP cards estimation is complex. There are several reasons for that like developers do not have domain knowledge, or they are technically not sound, or story cards are too big. Our proposed model considers these all problems and tries to solve this problem by acceptance tests, which will help to bring domain knowledge and technical knowledge to customers and developers.

2.5 Representation of system functionality

This is an import and crucial part of the story cards. There isn’t any tool or documentation that proves that the user story expressed on the story cards is valuable to the user or not. Stories on the story cards are written by the customer, so XP assume that requirement is correct, which leads problem related to requirement change and rework. To solve this problem we again focused on acceptance tests and strongly recommended to write them with the story cards. This acceptance test will help to write user stories on the verifiable ways.

2.6 Testable

Story cards must be testable. If it is difficult to test the story then that means story card is expressing non-functional requirements instead of user functionality. It is easy to write functional test or acceptance test for the functionality (functional requirements) in our model if you are able to write functional test or acceptance test that means the story is testable. Successfully passed all acceptance test means story card is fully developed.

Consider the following figure 3 which shows our new improved requirements elicitation process to capture user story on story cards based on the INSERT values.

---

**Figure 3 INSERT story card method for requirements elicitation process in XP**

In our approach we recommended a customer or stake holder who is on site has comprehensive application domain and business knowledge to put a developer into the picture.

Application domain knowledge and customer business knowledge from customer will help developer to focus on stockholder’s business needs and requirements and help them to cover any missing functionality. Customer business knowledge helps developers to understand how the system will affect and interact with the different part of the business,
and help to identify and understand different stakeholders, who are directly or indirectly affected by the system.

At the end of this successful discussion, customer starts story elicitation process and write the draft statement of requirements on story cards. These story cards are further analyzed, which assist to customer and developer to identify any problems and missing functionality. Missing functionalities become defect in working software. This scenario will help developers to focus on non-functional requirements. Also helps to keep in mind what they have to do to improve all the aspects of the business through structured system. Identified problems are being negotiated between developers and customer to acquire story cards. Following figure 4 shows an example of story cards captured through the new improved requirement elicitation process based on the INSERT model.

Figure 3 INSERT story card method for requirements elicitation process in XP, addresses the requirements capturing in XP with on-site customer. INSERT is based on a set of best practice guideline that helps both developers and customers to refine, clarify, and re-solve requirements conflicts with mutual discussion. The key benefits to apply the best practice guidelines are as

- Higher quality, lower cost requirements documents (Story cards).
- More understandable story cards compared to traditional story cards.
- Avoids misunderstandings among user requirements captured on story cards
- To reduce cost of changing the requirements at any stage of project life cycle
- To reveal technology we are using for a project is realistic
- Discovery of all likely sources of story cards
- Requirements are focused on core business needs
- Domain constraints often leads to critical requirements identification
- User finds easy to understand scenarios and to describe associated requirements.
- Easy to prioritize requirements
- Reveals ambiguities and inconsistency in the requirements
- Acceptance testing allows more stakeholders to participate in requirements validation.
- To support non-functional requirements

We also provide automated support tool to direct the processes of story card driven requirements capturing.

Figure 4 Story cards captured through the new improved requirement elicitation process based on the INSERT model.

Traditional method of story cards is still valuable and some of the guidelines are really crucial and unique. Onsite customer and simplicity of story cards are among them. In our proposal we also strongly recommended an onsite customer which is traditional story cards practice or guideline. In this section we tried to identify the improvement area of the story cards the following table will shows the commonality and variability between the story cards through traditional and INSERT model. This is just a prototype we still working on the story cards guidelines to extend them up to research level.

We apply INSERT technique to the few functional and unique requirements of the same project called e-commerce online store. Following table and graph shows a result, this shows that the INSERT model increased quality of the user story on story cards compared to the traditional methods. We apply this INSERT model to the key requirements or key story cards of the project to set up a prototype of this model.

<table>
<thead>
<tr>
<th>Story Card No.</th>
<th>User Story Title</th>
<th>No Of Passed Guidelines</th>
<th>Quality Percentage of user Story</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Admin Login</td>
<td>12</td>
<td>30</td>
<td>35</td>
</tr>
<tr>
<td>2 User Registration</td>
<td>14</td>
<td>32</td>
<td>40</td>
</tr>
<tr>
<td>3 Payment Method</td>
<td>13</td>
<td>29</td>
<td>38</td>
</tr>
<tr>
<td>4 Shipping Products</td>
<td>10</td>
<td>27</td>
<td>29</td>
</tr>
</tbody>
</table>

Table 1 Comparison of traditional methods and INSERT method on key story cards
Figure 5 Comparison graph of traditional methods and INSERT method on key story cards

3 Conclusions

The use of story cards for user stories in many Extreme Programming software development projects has been widespread. Several popular traditional methods for story cards (e.g., Cohen M, Kent B) have been used in successful fashion at some extent, but all lack of the powerful features for story cards guidelines, right sort of information on story cards and quality of user stories on story cards. They also do not involve anybody apart from customer on story writing workshop. This paper has described the INSERT model, new proposed frame work of story cards, and a new improved requirements elicitation process in XP. The experience with INSERT model and new framework of story cards indicates that it is feasible to contemplate improving user stories and story cards in Extreme programming.

4 References


Mapping Data-Flow Dependencies onto Distributed Embedded Systems

Stefan Kugele
Institut für Informatik
Technische Universität München
Boltzmannstr. 3, 85748 Garching
Germany

Wolfgang Haberl
Institut für Informatik
Technische Universität München
Boltzmannstr. 3, 85748 Garching
Germany

Abstract—Model-driven development (MDD) is an emerging paradigm and has become state-of-the-art for embedded systems software design. In the overall design process, several steps have to be taken in order to get from a high-level system design to the deployed binaries on the target platform: starting from model design, software partitioning and code generation reaching down to task and bus scheduling.

In this paper we focus on the later steps in the overall developing process and present a way to deploy clusters, which are tasks from an operational point of view, specified using the Component Language (COLA) [1]. In this context, we introduce the notion of a Cluster Dependency Graph (CDG) which forms the basis for scheduling, address generation and estimation of memory requirements for the used middleware. Moreover the CDG provides clues about possibly parallelizable tasks.

A case-study, namely an adaptive cruise control system (ACC), taken from the automotive domain serves as example throughout this paper to demonstrate our new approach.

Index Terms—Model-based development, data-flow graphs, embedded systems, distributed systems, code generation

I. INTRODUCTION

During the last years, model-driven development (MDD) has become state-of-the-art for the design and development of safety-critical embedded systems. Control systems such as those used in the automotive or avionic domain, demand for special requirements concerning reliability, robustness, and correctness. COLA as the used data-flow language turned out to be very promising because it provides support for a consistent development process from a high level system model design down to a level taking very specific platform details into account.

A. Data-flow languages

Over the past years, data-flow languages have become popular for the definition and design of safety-critical embedded control systems. Data-flow networks for example, are used in CASE-tools like MATLAB/Simulink [2] to describe complex automotive systems. There are some approaches for model-based development and design for embedded control systems based on the synchronous paradigm [3], [4]. Components defined in such a synchronous data-flow language operate in parallel and process input and output signals at discrete points in time, so-called clock ticks. Computation within data-flow networks and the communication associated therewith is assumed to elapse infinitely fast.

B. Introduction to COLA

In this paper, we use the Component Language COLA as a representative of synchronous data-flow languages. COLA is intended for the design of complex and reliable software systems, such as automotive or avionic control systems. COLA designs are modeled in terms of hierarchical components using a graphical and textual syntax respectively. The fundamental modeling concepts can be recognized in an akin manner in other industrial standards like the Unified Modeling Language (UML) [5] or MATLAB/Simulink. But in contrast to those, COLA is based on a rigorous semantics. Since COLA is a synchronous formalism, it follows the hypothesis of perfect synchrony [6] which means that in a given system, computation as well as communication occur instantly and therefore need no time.

Units are at the very heart of the COLA syntax definition. They can interact with their environment via so-called typed ports. We distinguish between input and output ports and summarize them in the units’ signatures. Units can either be composed in a hierarchical manner to build complex networks, or occur in terms of blocks forming the basic building elements of COLA like arithmetic (+, −, *, /) and comparison operators (<, ≤, =, ≠, ≥, >).

Data-flow is realized by channels which connect a source port with one or more suitable typed destination ports.

In addition to blocks and networks, units can be decomposed into automata, that is, finite state machines similar to Statecharts [5]. Each of their states is realized by a sub-unit which determines the respective behavior. Hence, this formalism is well suited to express disjoint system behaviors. These different behaviors are referred to as operating modes (see also [3], [7], [8]). In this paper, we describe a profound automated way to deploy COLA systems including a brisk usage of mode automata. This includes a foundation for schedule plan generation for the target platform as well as the generation of logical addresses for the used middleware.
C. Related work

Similar to our approach, Els et al. [9] use a graph-based method to calculate schedules for a hardware architecture consisting of processors, ASICS, and shared buses. Their notion of a Conditional Process Graph is used for analyzing control and data-flow dependencies of tasks that were already assigned to processors. Similar to Pop et al. [10], their focus is on scheduling. In contrast, our method provides inter alia a basis for scheduling but focuses on the deployment of mode tasks extracted from COLA models. The graph structure we are presenting is generated in a fully automated way from our COLA model and fits perfectly into the overall MDD process. Moreover, it provides all necessary information to generate C code and configure the platform. A middleware for distributed real-time systems is used to map inter-task communication.

D. Organization

In the following section, we motivate the usage of so-called mode automata to express distinct system behaviors. This leads to a distinction between different types of COLA clusters which partition the system model. Section II introduces the notion of a Cluster Dependency Graph. Possible task orderings as well as logical addresses are calculated according to this data structure. A case study outlined in Section IV demonstrates our approach using an adaptive cruise control as example. We conclude with an outlook on possible extensions and current work. Throughout the paper we use parts of the model of the case study to illustrate the presented approach.

II. System Organization Using Operating Modes

During our work, it turned out to be very useful to have a way to formulate distinct system behaviors in terms of different operating modes. This prompted us to include a language construct in the Component Language which is commonly called mode automata in the literature [1], [8], [11]. Fohler [12] describes issues of handling mode changes in the context of MARS [13].

In the case of COLA, the results present at the output ports of the unit which is realized by a mode automaton, depend on the automaton’s current state and the values at the output ports of the particular unit implementing that state. Figure 1 shows a COLA automaton which illustrates the principle of operating modes by means of a fictitious adaptive cruise control system (ACC) used in modern automobiles. Depending on whether the driver switches the ACC system on or off, the mode is altered. This design enables the developer to decouple the modeling process of the—in this case—two different behaviors. This caters for a reduction of system complexity and results in improved software quality. Moreover, the separation into different operating modes advances the reusability of COLA components and therefor contributes to saving development costs.

To seize the given example, one can assume that most of the ACC functionality is not used in the mode net_acc_off, whereas in the mode net_acc_on a lot of sensor processing like velocimetry and distance measurement has to be done.

A. COLA clusters

In a tool-backed model-driven development process, not only the modeling of system behavior is in focus, but for the sake of clearness and maintainability different views onto the system at hand are defined and have to be distinguished. Following the nomenclature of Pretschner et al. [14] we distinguish between the logical architecture and technical architecture. The first one describes the functional system behavior whereas the latter contains target hardware platform information and other non-functional requirements.

Operating modes are initially defined in the logical architecture. In order to clarify the transition from the logical to the technical architecture, we introduce the notion of a cluster in the following.

In the course of advancing from the logical to the technical architecture, the question arises in which way COLA units have to be mapped onto tasks from the operating system’s point of view. Such a task is called cluster in the technical architecture. Actually the process of clustering a COLA system is done manually. Yet, in a future version, a fully automated workflow is intended for this activity.

In the following, we are assuming the availability of such a clustered COLA system. No matter how we got the clustering—manually or fully automated—a valid one has to fulfill the clustering condition: the set of all clusters \( C \) has to cover the complete COLA system. More precise, a valid clustering of a system is given if and only if \( \text{clustered}(\text{root}) \) is satisfied. Here, \( \text{root} \in UI \) denotes the unique identifier of the top-most unit and \( UI \) is the set of all unique identifiers of a COLA system (see also [1] for more details). We distinguish the following two cases:

1) There exists exactly one cluster \( c_l \in CL \), where \( CL \) denotes the set of all clusters, which refers to the unit instance with unique identifier \( x \). In the following, this relation is denoted by \( c_l \sim x \). Furthermore, no sub-unit instances uniquely identified with \( x,y \) are allowed to be clustered separately.

\[
C1(x) \iff \exists c_l \in CL: c_l \sim x \land \\
\forall x,y \in UI : \#c_l' \in CL: c_l' \sim x \cdot y
\]

2) There exists exactly one cluster \( c_l \) which refers to a unit instance with unique identifier \( x \) implementing an automaton (denoted by \( \text{is}_{\text{atm}}(x) \)). Its sub-unit instances
The signature of automata and that of the unit instances implementing the states is the same.

\[ C2(x) \iff \exists cl \in CL : cl \sim x \land is_atm(x) \land \forall x,y \in UI : clustered(x,y) \]

The first-order predicate \( clustered(\cdot) \) is defined as follows:

\[ clustered(x) \iff C1(x) \lor C2(x) \]

B. Relating mode- and working-clusters

As mentioned in the previous section, the entire COLA system model has to be clustered. In this section, we introduce the distinction between different cluster types: mode clusters and working clusters.

Mode clusters realize mode automata which are responsible to initiate further control- and data-flow, that is, the next operations to perform. In our terms, these next operations are a set of working or mode clusters. A mode cluster can either be the top-most cluster of a system or can be initiated by another mode cluster.

Furthermore, there exists a very important correlation between these two cluster types: input and output ports of an automaton and those of the units realizing the automaton’s different states have the same ports, that is, these units have the same signature and therefor share the same input values.

In Figure 2 the dashed lines indicate port correlations, which means that these ports share the same value and actually are identical.

III. Cluster Dependency Graph

Data dependencies which arise due to channels between interconnected units in COLA networks, lead to dependencies in COLA clusters, too. For more complex COLA systems it is administrable to have a clearer understanding of data dependencies which in the sequel will help us to generate schedule plans and C code (see also [15], [16]). Further the contained information serve for the configuration of the used middleware. This middleware will be subject to another paper and will only be roughly discussed where it is necessary for a better understanding. We’ll give a short introduction into its functionality in Section III-C and IV.

Definition 1 (Cluster Dependency Graph): A Cluster Dependency Graph (CDG) is a directed, acyclic graph \( G = (V_w, V_m, V_b, E_d, E_m) \) with three types of pairwise disjoint vertices: working cluster vertices \( V_w \), buffer vertices \( V_b \) and so-called mode cluster vertices \( V_m \). The set of directed edges is divided into data-flow edges \( E_d \) and mode edges \( E_m \) with \( E_d \cap E_m = \emptyset \). For the edges it holds: \( E_d = \{(u,v) \mid u \in V_w, v \in V_b\} \cup \{(u,v) \mid u \in V_m, v \in (V_w \cup V_m)\} \) and \( E_m = \{(u,v) \mid u \in V_m, v \in (V_w \cup V_m)\} \).

Solid data-flow edges going out of a working cluster vertex (visualized by a rectangle) and pointing into an octagon (buffer vertex) symbolize that the working cluster vertex writes data into a buffer, whereas an edge pointing from a buffer vertex to an working or mode cluster vertex (symbolized by a diamond) indicate the fact that they read from the buffer. In case of mode cluster vertices, only mode edges (drawn as dashed edges) can start here and the edges can only point to working and mode cluster vertices. This distinction is made in order to emphasize the different exclusive control- and data-flows depending on the current mode. Figure 3 shows an example for a cluster dependency graph of the case study explained in Section IV.

A. Data dependencies

Data dependencies arise when two or more COLA units are interconnected via channels. Data-flow is then given in the following three cases:

1) Data-flow from an input port of a network to either an input port of a connected sub-unit or to an output port of the network at hand, or
2) between sub-units of a network, that is, from a single output port to at least one input port of another sub-unit, and finally
3) between an output port of a sub-unit and an output port of the surrounding network.

These data dependencies between COLA units in the logical architecture are reflected in the technical architecture, that is, dependencies between clusters. These dependencies on the
technical architecture are represented as edges in the CDG.
We distinguish between two kinds of edges: solid data-flow and dashed mode edges, as introduced above. 
Before stating the accurate relation between the logical and technical architecture, let us first introduce necessary notations according to Kugelo et al [1].

Let $u_1 = \langle n_1, \sigma_1, c_1, I_1 \rangle$ and $u_2 = \langle n_2, \sigma_2, c_2, I_2 \rangle$ be two units of the COLA system $S = \langle u, U \rangle$ with $u_1, u_2 \in U$, being the set of all units, and $u$ denoting the root unit of the COLA model. $\sigma_1$ and $\sigma_2$ are the signatures of both units, that is, the particular vectors of typed input and output ports (accessible using $in(\sigma)$ and $out(\sigma)$). Furthermore, let $ch = \langle a, s, \{d_1, d_2, \ldots, d_k\} \rangle$ be a channel connecting a single source port $s$ with at least one destination port $d_i$, with $1 \leq i \leq k$.

Then, two cluster vertices $cl_1$ (working cluster) and $cl_2$ (working or mode cluster), are connected by solid edges with a buffer vertex $b$ in between ($cl_1 \rightarrow b \rightarrow cl_2$), if there exist two connected unit instances $u_1$ and $u_2$ (with unique identifiers $x_1$ and $x_2$) in the COLA system model, which are associated with these clusters: $cl_1 \sim x_1$ and $cl_2 \sim x_2$. It holds

$$\forall cl_1 \in V_w, cl_2 \in (V_w \cup V_m) \forall x_1, x_2 \in UI :$$

$$cl_1 \sim x_1 \land cl_2 \sim x_2 \implies$$

$$[\exists e_1, e_2 \in E_d \exists b \in V_b : e_1 = (cl_1, b) \land e_2 = (b, cl_2) \implies (\exists ch \in C \exists x_1, x_2 \in UI : s \in out(\sigma_1) \land$$

$$\exists d_i \in \{d_1, \ldots, d_k\} : d_i \in in(\sigma_2))$$

whereas $C$ is the set of all channels in the network containing $u_1$ and $u_2$. In this scenario, we dictate that $u_2$ is not a sub-unit of $u_1$ since all sub-units of a network are automatically contained in the same cluster as $u_1$. But then, for dashed mode edges we have a different characteristic. Let $cl_1$ be a mode cluster vertex and $cl_2$ be either a working or a mode cluster vertex. Then, there is a dashed edge connecting these vertices ($cl_1 \dashrightarrow cl_2$) if $u_1$ is an automaton with a set of states $Q$, and $u_2$ is the instantiation of one of its states $q \in Q$, denoted by $inst(q) = u_2$.

$$\forall cl_1 \in V_m, cl_2 \in (V_w \cup V_m) \forall x_1, x_2 \in UI :$$

$$cl_1 \sim x_1 \land cl_2 \sim x_2 \implies$$

$$[\exists e \in E_m : e = (cl_1, cl_2) \implies$$

$$is_atm(x_1) \land \exists q \in Q : inst(q) = x_2]$$

**B. Causality and task execution order**

The cluster dependency graph provides a basis for reasoning about causality and is appropriate to cover the execution order of tasks. In Figure 3 causality is given in the following way: the working clusters Rotation, Radar and UI have to be executed in order to write their results to the suitable buffers $s\_act, dist, mode, and s\_user$. In addition, these working clusters are again dependent on the values of their input buffers which are in turn set by the working clusters DEV_ROTATION, DEV_TIME, and so on. Once the buffers ($s\_act, dist, mode, and s\_user$) have been set, the mode cluster ACC can be executed using the data present at its connected buffer vertices. Depending on the active mode, either ACC_off or ACC_on will be executed afterwards and write its results to the respective buffers. Based on this causality, a possible execution order of clusters—tasks in the sense of operating systems—can be derived in a straightforward manner. We have to point out, that there can be a plethora of possible execution orders. The task of our offline-scheduler which is currently under development, is to select the best order with respect to a set of given constraints. In the example given in Figure 3, hundreds of possible execution orders can be found. Some of them are exemplarily presented in Figure 4.

![Figure 4. Examples for feasible task orders.](image)
and beneficial.

C. Logical address generation

As mentioned before, the presented approach for a dependency graph is intended for use in a MDD process. Haberl et al. [15] presented an approach to generate code fully automatically, to minimize the possibility of programming faults. In order to use the process for development of distributed systems, it is necessary to allow for communication during runtime. Each channel between two clusters, which is transformed into C code and thus turn into tasks during runtime, indicates the need for a communication link at runtime. As described before, buffers are used for temporary storage of data sent from one cluster to another. These buffers correspond to memory allocated by the middleware used on the execution platform. It is, amongst others, the middleware’s task to enable for transparent and timely communication between the running tasks.

The middleware can be managed using a configuration file which contains information about the sizes and addresses of local and remote buffers. The dependency graph provides assistance during construction of this configuration file as well as the correct addressing of middleware API calls while generating the C code for each cluster. Every buffer vertex in the graph is given a logical address, which will be used later on in the development process for middleware configuration and appropriate read and write calls by the connected clusters. Of course, the logical addresses of all buffer vertices have to be different to avoid race conditions.

In addition to inter-cluster communication, there is a need for storing the state of each cluster. As the hypothesis of perfect synchrony assumes the periodic invocation of each unit, we use a time-triggered scheduling scheme. Thus each generated task is started over and over again. In order to keep the tasks’ states between invocations, the middleware saves their local variables. This state buffering is realized using a logical address, too. Each cluster vertex in the dependency graph is assigned its private logical address for this purpose. When executed, a task’s first job is to read its state using the assigned address. Accordingly, the last instruction before the task’s termination writes the actualized state back to the middleware. Figure 5 shows a possible addressing for the dependency graph introduced in Figure 3. As described, each buffer in this example is assigned an address pointing to an appropriately sized data buffer. The addresses reserved for the tasks’ state storage are enlisted as well.

While the stated concept for saving states is true for both kinds of clusters, a mode cluster is given an additional address. As we explained in Section II-B, mode clusters decide on which clusters to be executed next. The information about their decision which state to execute is not depicted using a channel in the functional model, but by means of hierarchy. All working clusters contained in a unit implementing a state, form the active cluster set. But as there is no channel in the functional model, the cluster graph does not contain a buffer for filing this decision. That is the reason why mode clusters are given an additional address. Considering our example graph, this is true for the ACC mode cluster. Figure 5 states both addresses for that cluster. When being executed, the mode cluster stores a numeric value, which points to a mode, at that logical address. As the middleware is the first instance to see the decision about the active mode, it hands the tasks realizing the functionality of that mode over to the operating system’s dispatcher for execution.

IV. Case Study

In the following we will exemplify the usage and the benefits of our dependency graph considering a model for a fictitious adaptive cruise control (ACC). This system is intended to keep an automobile’s velocity at a constant value, while maintaining a defined minimum distance to the car driving ahead.

The top-level diagram of the according COLA model is shown in Figure 6. While this is a fictitious model and not derived from any real implementation, it is well suited to demonstrate our concepts in this paper. The COLA unit shown in this diagram is a network containing several sources, sinks (both are indicated by solid rectangles in the upper right corner) and other sub-units. Sources and sinks refer to sensors and actuators of the real system, while the rest of the shown sub-units implements the ACC’s behavior. In the following we will refer to this diagram in order to exemplify some concepts.

The dependency graph for the ACC example has already been given in Figure 3. As can be seen in the figure, the...
The example also shows, how the task reads its actual state in buffers which are connected to the working cluster in Figure 5. This realizes the inter-cluster communication described before.

In lines 6, 7, and 10 data are read from or written to the buffers which are connected to the working cluster in Figure 5. This realizes the inter-cluster communication described before. The example also shows, how the task reads its actual state in line 5 and writes back the actualized state in line 11.


```c
void net_rotation200399()
{
    state_rotation200399 unit_state;
    int rotation_0, time_1, rotation_out_0;
    nw_read(22, &rotation_out_0);
    nw_read(17, &time_1);
    nw_read(16, &rotation_0);
    nw_restore_task_state(14, &unit_state);
}
```


```c
switch(unit_state.atm_state)
{
    case 0:
        if ((mode == 1))
            unit_state.atm_state = 0;
        break;
    case 1:
        if ((mode == 1))
            unit_state.atm_state = 1;
        break;
}
```

V. CONCLUSIONS

In this paper we introduced the concept of a Cluster Dependency Graph. We showed that this formalism is well suited to capture the dependencies arising from data-flow models using COLA as an example language. We defined clusters as distributable entities which build up the partitioned system model.

The definition of clusters together with the dependency graph enables for unattended generation of application code. This results in code including the functionality captured by the model as well as the communication needs imposed by the allocation of clusters onto a distributed system. Therefor, channels included in the data-flow model are mapped to logical addresses. These addresses are then attached to the corresponding vertices of the graph. Later on in the development process this information is used by the code generator when inserting communication calls to, and generating a configuration file for, the middleware.

Additionally, the dependency graph forms a basis for automatic construction of feasible system schedules and provides a clue for distribution of tasks to the available system nodes. These extension are subject of current research and will be discussed in future work.

REFERENCES


Agile Intelligence – Principle Methods and Mechanics

Koji Matsumoto¹, Martin Eggenberger²
¹Delta Dental of California, San Francisco, CA, USA
²Quanis Inc, San Francisco, CA, USA

Abstract - Agile Development Methodologies have evolved since the mid 90’s and today most Software Development and IT Organizations have adopted and used an agile development methodology. In this paper we propose and validate a formal, albeit non-traditional, approach to the Agile Development Process. To that end, we have established process criteria that can and should be measured in any development methodology and can be adapted to an organizations needs as necessary.

1. Introduction

Since the beginning of the computerized age most organizations have been struggling with the volatility of software development efforts. To address these issues numerous development processes have been introduced and later been abandoned. Initially most development processes such as the Waterfall [1] and its related models were based on a tayloristic paradigm. Taylor, the forefather of modern management introduced “scientific management” – A job science that includes the selection and training of workers, and proper supervisory support to increase overall productivity. Applied to software engineering, taylorism promotes a strong conformance to a phased development approach. Requirement Gathering is followed by architectural design and later by development and testing. Additionally, it also encourages a strong division of labor and roles (business analysts, architects, developers, etc.). This strong conformance to processes and roles has given room for more agile methodologies. The Agile Development Manifesto [2] takes an ownership approach, in which individuals and interactions are encouraged over processes and tools. Such an approach provides the agility necessary to continuously realign the goals of the effort with the needs and expectations of the customer. Even though this approach is very promising, some additional considerations and metrics have been missing since the inception of agile methodologies and this paper examines a mathematical foundation for measuring the process itself.

2. Development Methodologies

Software development processes have evolved from being heavily process oriented to a more individual or team based approach. According to Highsmith and Cookburn [4] – “What is new about agile methods is not the practices they use, but their recognition of people as the primary drivers of project success, coupled with an intense focus and maneuverability.” This leads to a new set of principles and rules that define the agile ecosystem. According the planning system (Figure 1) [5] depicting the overall spectrum of potential development methodologies one can find Hackers on one end of the spectrum and inch-pebble ironbound contract on the other hand. Given such a planning system we can quickly recognize that in the Milestone driven approach a slew of process and metrics exist that allow measurement and simulation of pre defined criteria such as amount complete, level of completeness and others.

![Figure 1: The Planning System](image)

One the other hand, though, no or few metrics exist to measure Agile Methodologies and their success. Further more an enterprise may decide that only some projects should use an agile approach whereas others should use a Milestone driven approach.

2.1 Organizational Maturity

Any mature organization using or implementing the Capability Maturity Model for Software (CMM) [6] soon discovers that the need for information and process measurement criteria as a feedback mechanism is essential for ongoing process improvement, and therefore, more rigorous Development Processes such as the waterfall model are easier to measure. Because agile processes tend to be viewed more chaotic and less traceable by Managers who are used to dashboard and progress reports, the agile processes must be formalized in such a way that all stakeholders can be satisfied. Generally, the larger the organization, the more such measurements are necessary as people need to be kept in the loop.
3. Formal Agile Development Process

Traditional SDLCs such as waterfall and iterative, (including hybrids and derivatives of the like) can be represented by a very common set of procedures (Figure 2) typically performed to illustrate any number of analytic endeavors including “as-is” and “to-be” business models (sometimes called current state and future state assessments, straw-man diagrams, activity diagrams in RUP, etc.).

![Figure 2: Activity / Process View](Image 84x555 to 131x567)

Although these logical representations are certainly useful in their own right for the purposes they are meant to serve, they are inherently spatially limited; subsequently limiting the amount of information that can be fitted and the degrees of freedom objects or the relationships between them can encompass without sacrificing some clarity or coherence (although analysts over the years have found clever ways of packing more information into their diagrams to illustrate 3 to N dimensional objects or diagrams, but not fraught without all the manually tedious customizations).

In other cases, it is the level of expertise of the analyst themselves or the modeling tools available for analysis, design, and construction that do not allow for more robust solution oriented representations.

Agile especially is a sore spot for more traditional approaches to workflow / development process depictions and almost impervious to any kind of holistic illustration without brushing up against any of the aforementioned limitations or sacrifices. With the advent of dual-commits and multi-casting capabilities provided by the hardware and software components of the 80’s and early 90’s – forcing an almost n-dimensional consideration and treatment of software architecture, the same (if not more) mind-shift is required of the business, project, and development management disciplines as it relates to our conceptual understanding of business / workflow processes, project planning and implementation to complement the many requirements and architectural and technical considerations that are formalized.

Without the right tool or level of expertise to provide a more holistic view of the Agile process, the industry has resorted to more rudimentary practices of simply encouraging the use of common sense, following generic principles, and presumption of high competency teams.

Likewise, illustrations of the agile process are typically time / scope based cross-sections (Table 1) implying / assuming the very existence of evenly reducible workloads across identical periods of time, mature organizational infrastructure, and the surely innocent tenet of open and honest communication between all groups or individuals involved (political agendas aside); especially when dealing with vendor / contractor heavy project teams (how do you deal with change management when the vendor is fixed-bid? Doesn’t the whole idea of expectation of change and agility conflict with agile when a change may require renegotiations of the contract or an extra assessment (cost / benefit) to be performed each time a change is introduced? How about when it is Time & Materials? Can’t the vendor abuse the amount of change introduced? How about the legal ramifications of telling a contractor how to do something rather than what to do?).

<table>
<thead>
<tr>
<th>Table 1: Typical Agile Sprints</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sprint K</strong></td>
</tr>
<tr>
<td>Baseline Module K+1</td>
</tr>
<tr>
<td>Biz Req’s</td>
</tr>
<tr>
<td>Prioritize Module K+1</td>
</tr>
<tr>
<td>Biz Req’s</td>
</tr>
<tr>
<td>Develop Module K+1</td>
</tr>
<tr>
<td>Biz Req’s</td>
</tr>
</tbody>
</table>

Unfortunately, these cross-sections alone (Agile WBS, Work-log, Product Backlog, etc.) when compiled / consolidated do not additively sum to produce a reliable “big picture” of the overall Agile process / methodology.

The other issue with current references on Agile is its extreme bias towards Project Managers, Development Managers and the like, leaving other key individuals (such as architects) or critical functional groups including business users and stakeholders in the dark.

The proposed solution is an “at-a-glance” illustration of the agile methodology so that anyone from business stakeholders, architects, project managers, analysts, to end-users, production support, release engineers have an idea of direction, timelines, and can more effectively manage / coordinate their own time. In a sense, it is the missing piece which would integrate all of these forcibly coupled cross-sections and in most cases replace them altogether, reducing the number of overall artifacts to begin with and thereby avoiding the N2 problem in which the introduction of every new element to a set exponentially increases its complexity.

3.1 Process Conventions

In classic flowcharts, the relationship between two activity boxes are almost always “finish-to-start” (linear) in which activity A must complete before activity B is triggered as in figure 2 above (notice the expectation here is
that Activity 1 needs to complete before moving on towards Process 1). The proposed agile flowchart, semantics aside, is noticeably different (Figure 3):

![Figure 3: Basic Sample Agile Workflow](image-url)

The primary objects being the artifact and activity nodes as illustrated in Figure 4 below.

![Figure 4: Agile Workflow Legend](image-url)

Although the code / build is the centerpiece to agile development with documentation more as a support vehicle for quality control and maintainability, use of the word ‘artifact’ has been abstracted here to imply not only code, but documentation as well (e.g. requirements, architecture, etc) or the tool utilized for data capture.

Careful placement and the relationships between the objects create the effect of a traditional flowchart with all the usual pieces of information such as (but not limited to) chronology, flow logic, entity type, activities. The resemblances stop here with a slew of out of the box features that are additionally embedded with this new approach.

- Emphasis on driving forces can be placed if the creation of one artifact for instance kick starts another activity.
- Activity type and relationship (checkpoint criteria, “start-to-finish”, “finish-to-finish”, etc).
- Artifact distinctions (An artifact node that is pink in color indicates that the artifact(s) contained may be a milestone or some entry criteria for a key activity to take place).
- Communication Methods (Figure 5).

![Figure 5: Artifact / Activity Node relationship](image-url)

Vertically (or horizontally) displacing activities from each other or from decision / checkpoint nodes can imply gradations of priority or communication method (since the size of each object type must be identical and conform to the same gridlocked structure of the large and smaller squares – the observant reader will also notice the proof of Pythagoras’ Theorem – object displacement / placement is not a haphazard process).

Consider the performance of an environment planning activity. The subsequent activity to set up and configure the development environment should be directly stacked on top of the previous activity so that it either immediately follows, or indicates a finish-to-finish relationship with said activity, in order to illustrate the seamless communication (verbal) between the resources involved in the two logically (not necessarily physical since resources often play multiple roles) separate activities. On the other hand, the procurement process for production hardware can be safely placed multiple activity node distances away, indicating that it not only can take place further down the line, but the method in which the environmental requirements are communicated can be via approved documentation, or some formalized entry process.

Although initially grouping agile flowchart objects in this fashion may appear more complicated and require some amount of acclimation, in application it produces a unexpectedly opposite effect and actually provides more clarity / guidance for the activities a project resource would have had to do anyway (Figure 6 below).
Traditional swimlane delineations are not required with agile MODULATION. Figure 6: Intermediate Sample Agile Workflow

Now project resources and stakeholders have a single, unified view of all the high-level project details and a roadmap for next steps through the framework provided. Keeping track of the number of project nodes or activity nodes traversed now becomes an easy exercise in order to realize the additional project monitoring benefits (see section 3.1 regarding more on status tracking) without resorting solely to date-driven sprints which may not accurately identify the recurring or troublesome areas within, since increments in sprint number do not necessarily imply progress made without also considering the ratio of node traversal vs. milestones achieved (i.e. # of nodes (activity or artifact) revisited (performed again) vs. # of measurable achievements). For example, a low number may indicate the existence of outstanding dependencies or an unresolved issue. These checks and balances keep communication channels more open amongst the team members, business stakeholders, and executive management.

The reader will have also undoubtedly noticed the introduction of 2 new conventions we have not yet discussed (Figure 7). Again, we have abstracted here and we are calling the various flavors of Agile Development [7] simply a ‘Development Cycle’ with no specific mention of duration and leave that up to the discretion of the project. Notice that the development cycle is represented by a series of shaded discs, common to most SDLC phases, to emphasize the fact that although each development effort may be only 2-4 weeks, the project team members are still in fact performing these traditional tasks and measuring progress against these waterfall concepts, albeit within a much faster paced and less formal environment. The anticipation being that each successful subsequent development cycle would result in an incremental release.

Certain artifacts contained within the artifact nodes are stateful (to be finalized in piecemeal fashion), hence it is necessary to indicate the iterative nature of artifact completion by circumscribing the node by a series of distinct arrows connected in a manner that implies the logical order in which they are to be completed.

In Figure 8, we have chosen to highlight only 4 primary components for each artifact, but the inclined reader may, as necessary, find a more appropriate topological structure such as a 2 dimensional octagonal polygon, with each side indicating a critical piece of the artifact / deliverable in question. Certainly the realm of geometric structure isn’t limited to artifacts and it can / should extend well beyond the scope of the objects themselves to adapt to each organization’s unique environment / structure.

3.2 Process Evaluation Criteria

The formalization of a project or program specific evaluation checkpoint should follow the overall established development framework, methodology, and governance in place – with the heavy assumption that the IT controls in place are first and foremost accommodating the business needs and their more detailed corresponding business requirements.

However, since Agile development seeks an orderly evaluation mechanism, the concrete rigor illustrated also provides a means for implementing / exercising process evaluations in the absence of a formalized governance process or sophisticated business partnership with a consensus based approach to driving checkpoints and computing their associated metrics which are critical for monitoring, controlling, and ensuring quality control for the project or program.

3.3 Process Validation Criteria

The presence of Abstract Rigor regardless of maturity, does not burden the organization as it is able to steer clear of subjective insight, foresight, and hindsight as well as address the aforementioned as input for the very reason of determining deviations from objectivism. However, these poignant items do become more prominent with time, and if
not addressed these issues will turn into risks that assume the burden of mitigation strategies.

Simple Business Intelligence tools or ad hoc reporting will not suffice for the level of mathematical rigor required for the purposes of programatically automating the validation process, but most medium to large industries do have an actuarial department or have even elected to define a financial engineering charter that can carry out the calculations referenced within this document.

4. Process Validation

We can represent a Project Activity as a basis, $PA$, of some vector space $S$ (with all the usual properties of linear independence satisfied and defined for the basic arithmetic operators). We can now form a trivial partition on $PA$ into $n$ subsets we call tasks, $T_i$ (the partition is trivial if we take $i=1...n \in \mathbb{N}$’s where $N' \in \mathbb{N}$ (set of all natural numbers) and $n > 0$ and finite, $|T|$)

In practice however, project managers are typically aware of up to $k$ relevant tasks for said activity, but may only choose to address $n$ of them (where $k > n$) within the Project Plan based on ROI, time to market, best practices, enterprise strategy (business & IT), and individual experience. We must further assume based on competency limitations, experience levels, and the presence of political obstacles, that there are $n+1, n+2..m$ tasks that are unforeseen (or simply not possible) and inversely proportional to the aforementioned attributes. Additionally, empirical evidence suggests that the partitioning of tasks itself is heuristic and assumes a refinement process over time to minimize the overlap (redundancy of work) between each successive task such that if $a \in T_i$, then the goal is such that $a \in T_{i \neq 1}$:

$$PA = \frac{1}{k} \sum_{i=1}^{n} T_i + \sum_{j=m+1}^{n} T_j$$

Where $PA$ completion can be gauged by the summation of the $T_i$’s divided by $k$, the total # of tasks considered, plus some value resulting from the number of unforeseen tasks, the $T_{i \neq 1}$’s.

The amount of overlap or unforeseen tasks can be mitigated by the existence of skilled staff, organizational maturity, good governance and corporate controls (enterprise accessible, published standards and guidelines and compliance adherence) and a more appropriate form for a true % completion is the following:

$$PA = \frac{1}{k + m - (n + 1)} \sum_{i=1}^{n} T_i$$

The $T_i$’s themselves can be functions as necessary (e.g. for $i=1$ and $T_1 = a_1(x_1)+a_2(x_2)+...+a_{k}(x_{k})$, where each $a_i(x_i)$ represents some role specific unit of work, for $T_1 \neq 1$ would indicate that some $a_i(x_i)$ was insufficiently done, indicating for example a mismatch between role and unit of work).

Therefore, we clearly see that only ideal or perfect conditions give rise to $PA$ attaining 100% completion due to the aforementioned limitations.

The Project Plan, $PP$, can then be formalized as the careful compilation of these individual activities:

$$PP = \frac{1}{k + M - (N + 1)} \sum_{j=1}^{n} PA_j$$

Where $K,M,N$ from (2) above since we are now considering the $PA_j$’s instead of the $T_i$’s.

With this approach, a mechanism exists for managing project unknowns from planning, budgeting, and mitigation perspectives for the risks and issues produced / encountered by a project or program. Suppose that for each $PA_j$ there is an $R_j$, each with its own associated weight and independent number of risks, $x$, logged against it:

$$Risk = R_j = c_j \cdot \sum_{i=1}^{n} a_j r_j (x_i)$$

Note: Here we take $n = \max \ (# of X_i$ for each $j$).

- $c_j = [(k-n)+1]$, the number of tasks not completed and inversely proportional to $PA_j$, since not doing a task increases risk and the larger $k-n$, the larger the risk (note that we exclude the term m-(n+1) as you can not log a risk against an activity that is unforeseen).
- $a_j$ represents some arbitrary weight based on lessons learned, industry best practices, etc.
- $x_i$ represents the number of risks actually logged for each risk type, $r_j$

Alternatively, you may constrain the number of risks for each $PA$ in order to create a well defined matrix with the following structure:

$$A = \begin{bmatrix} a_1 r_1 & 0 & \ldots & 0 \\ 0 & a_2 r_2 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & a_m r_m \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} x_{11} & x_{12} & \ldots & x_{1n} \\ x_{21} & x_{22} & \ldots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \ldots & x_{mn} \end{bmatrix}$$

The placement of zeroes above in matrix $A$ is contrived for the usual laws of matrix multiplication:
For example, the risks associated with not performing a project activity (set of tasks) that identifies a high-level architecture would certainly have a different set of risks (risk type) and different proportions of criticality (weighting) associated with it than say missing a project activity to establish a project accessible document repository.

It should be noted that issues can also be treated as risks, since an issue can convert into a risk if not addressed after some duration of time as determined by some underdamping harmonic function (since the behavior of an issue is not a simple monotonic function but rather oscillates with time according to the performance of certain project activities) with a bounding envelope that stabilizes below R (risk criteria) as illustrated in Figure 8 below.

\[
AB = \begin{bmatrix}
    a_1 r_1 x_{11} & a_1 r_1 x_{12} & \cdots & a_1 r_1 x_{1n} \\
    a_2 r_2 x_{21} & a_2 r_2 x_{22} & \cdots & a_2 r_2 x_{2n} \\
    \vdots & \vdots & \ddots & \vdots \\
    a_m r_m x_{m1} & a_m r_m x_{m2} & \cdots & a_m r_m x_{mn}
\end{bmatrix}
\]

The accumulation of risk can then be defined by the following:

\[
\sum_{j=1}^{m} R_j = \sum_{j=1}^{m} c_j a_j r_j (x_j)
\]  \hspace{1cm} (5)

- Where \( c_j \) is presumed to be a constant, but can, for the more inclined, be represented as a function with \( |k-n| + 1 \) as its origin, in order to accommodate for the non-linear behavior of accumulative risk due to the compounding / additive effects risks may have.

A trivial (simple) example of the whole being greater than the sum of their parts would be to either skip or neglect the set of activities that comprise the traditional analysis phase of a waterfall project. Surely, the effects of doing so would not simply be isolated to that phase or be linear, but instead would have significant downstream effects on design and development activities and have exponential impacts to quality, budget, and time to market.

4.1 Measuring Status / Progress

Project Progress may be measured by any of the primary attributes of the project deemed appropriate (ROI, business stakeholder value-add, federal / state mandate, etc.) or by a more distributed set of project attributes. For purposes of elucidating the metrics process, we will be considering only 2 primary inputs to the calculus of such an effort by first examining some subset of the project plan, \( PP \).

Let us assume further that there are only 3 primary analysis activities for the necessary and appropriate transition to design activities and that they fall within some sequence of \( PA_k, \ldots, PA_{k+p} \) (in this case, \( p=2 \)). This would be a relatively simple procedure to apply since most project managers will explicitly track requirements related activities / milestones such as (but not limited to) Gathering requirements, Analyzing requirements, and Prioritizing requirements:

\[\sum_{j=k}^{k+p} PA_j = \sum_{i=1}^{n1} g_i + \sum_{i=1}^{n2} a_i + \sum_{i=1}^{n3} p_i \subseteq \sum_{j=k+1}^{k+3} PA_j\]

- Where \( g_i \)'s, \( a_i \)'s, \( p_i \)'s stand for the gathering, analyzing, prioritizing activities respectively, and is clearly a subset of the larger set which includes activity \( k-1 \) through \( k+3 \).
- With \( m_1, m_2, m_3 \) representing the values \( m-(n_1+1), m-(n_2+1), m-(n_3+1) \) for brevity.
- The most critical of requirements are gathered first for analysis and incrementally developed as illustrated in Figure 9 below.

Next, we take \( n = \text{max} \ (n1, n2, n3) \) and add \( n-n1, n-n2, n-n3 \), dummy tasks only where \( n1 \) or \( n2 \) or \( n3 < n \) and substitute values reflecting the \% completion of actual tasks for the dummy tasks respectively:

\[\sum_{j=k}^{k+2} PA_j = \frac{\sum_{i=1}^{n} (g_i + a_i + p_i)}{k + m - (n + 1)} = R_{\text{Off}}\]  \hspace{1cm} (7)

- More emphasis (by means of weighting) on the \( g_i \)'s may be more appropriate since they yield a higher number of requirements, \( N \), than performing the \( a_i \)'s,
although decomposing requirements during analysis may yield an additional set of requirements \((N+K)\) where \(K<N\). Alternatively, simply gathering requirements and not appropriately performing the analysis activities may present enough risks that the weighting becomes an insignificant matter.

For example, suppose there are no unforeseen tasks, and we perform all foreseeable tasks, \(k+m-(n+1)=n\). Further, we only consider \(g_i\) and \(p_i\), and assume that there are 10 \(g_i\) tasks and only 1 \(p_i\) task and obtain completion levels of 1 and 0.5 respectively. But we have \(n=10\), and so we must add 9 dummy tasks to the \(p_i\)'s. The \(g_i\)'s then add up to 10 and the \(p_i\)'s add up to 5 since the value of the one actual \(p_i\) performed is reflected 9 more times. Dividing this total \((10+5)\) by the denominator \(20\) then gives the appropriate \% completion, 75\% as expected.

It is important to realize that although we have consolidated 3 logically separate project activities here, we do not lose a sense of chronology as some amount of waterfall is both unavoidable and a beneficial best practice when applied appropriately as shown in Figure 10 below.

![Figure 10: Optimizing Task Performance](image)

**Figure 10: Optimizing Task Performance**

It is clear from the diagram that the prioritization activities are most optimal when it intersects the logarithmic requirements gathering curve. Requirements Completion (as deemed by the business stakeholders and not the Project or Development Manager unless they additionally play that separate logical role) can then be represented as follows:

\[
R_{Comp} = R_{Coeff} \cdot \log \left( \frac{Re_{q_{Tot}}}{Re_{q_{Curr}}} \right)
\]

\[\text{(8)}\]

Note: Both \(Re_{q_{Tot}}\), \(Re_{q_{Curr}}\) > 0 (which makes sense since a checkpoint without any requirements to review would be quite unnecessary).

- \(Re_{q_{Comp}}\) is equal to the \% completion of the core project activities that comprise requirements elicitation, analysis, and prioritization (by criticality) times the number of perceived requirements realized.

- \(Re_{q_{Curr}}\) = \# of requirements so far and \(Re_{q_{Tot}}\) = total \# of requirements (with agile or any other like-effort, this is an approximation likely addressed by calculating the risk level of performing the approximation itself).

- The use of the log function clearly indicates that the first set of requirements are the most business critical and should weigh more heavily on progress than subsequent requirements. It can also be customized to fit each organization’s needs by adjusting the base number (e.g., 2, 10, are some common base numbers).

We now consider the scenario where the project stakeholders are alerted of a potential issue via email that only 20\% of requirements have been accommodated for by development while there is over a 50\% requirements completion being reported by the business (gathered, analyzed, and prioritized with relatively no pressing issues / risks).

Without more information about development progress, we may overlook the scenario in which the requirements that have been accommodated for are of the most critical and most architecturally significant nature, providing an underlying architectural framework (a very non-trivial milestone from a developmental perspective) with the implication that all future requirements can be implemented more efficiently and quickly (something the business would certainly be interested to hear before berating the developers).

To prevent this kind of unnecessary alarm (it should be verified in person or over the phone) we look at the development activities and perform a very similar exercise to the one we performed for (7) and obtain the following:

\[
\sum_{j=1}^{k+3} PA_j = \sum_{j=1}^{k+3} (d_i + c_i + t_i) \quad \text{and} \quad n = \frac{k+m-(n+1)}{k+m-(n+1)} = D_{Coeff}
\]

\[\text{(9)}\]

Note: \(\sum_{j=1}^{k+3} PA_j \cap \sum_{j=1}^{k+3} PA_j = \emptyset\)

- Where \(d_i\)’s, \(c_i\)’s, \(t_i\)’s stand for the designing, coding, testing activities respectively.

- Here we assume \(t\), from (7) above, indicating that these 3 new sets of activities must be clearly different from the previous 3 sets of activities from requirements completion.

We are now ready to introduce the secondary input for measuring overall project progress, Development Completion, as deemed by the Product or Development Manager (not the business stakeholders or end users unless they additionally play that separate logical role):
\[ D_{\text{Comp}} = D_{\text{Coeff}} \cdot \frac{\text{Re} \ q_{\text{Dev}} + \log \left( \frac{\text{Re} \ q_{\text{Tot}}}{\text{Re} \ q_{\text{Dev}}} \right)}{\text{Re} \ q_{\text{Tot}}} \]  

(10)

Note: Both \( \text{Req}_{\text{Tot}}, \ \text{Req}_{\text{Dev}} > 0 \) and \( \text{Req}_{\text{Dev}} \leq \text{Req}_{\text{Curr}} \) since development may not cover all of the requirements identified for a particular development cycle and may need to offload to a subsequent development cycle.

- Since Development is critically dependent upon the appropriate completion of requirements sets, it is used as input into \( D_{\text{Comp}} \).

Project Completion can then be summarized as the aggregate of Requirements Completion and Development Completion as follows:

\[ \text{Pr} \ oj_{\text{Comp}} = \frac{\text{Re} \ q_{\text{Comp}} + \text{Dev}_{\text{Comp}} + \ldots + X_{N-\text{Comp}}}{N} \]  

(11)

- Where \( N=2 \), indicating that for our example, we have chosen to only take into consideration 2 primary inputs.

Reports of substantial requirements coverage from a development perspective can mask poor architectural decisions that make accommodating any further requirements a monumental effort or even worse not feasible without a serious compromise to security, performance, software / hardware expectations, etc. By considering \( N=3 \), and setting \( X_{\text{comp}} = \text{Arch}_{\text{comp}} \), we can evaluate and integrate architectural completion into (11) and thereby address the aforementioned scenario.

Furthermore, by forming a Project Completion Tensor with the below components, we can evaluate \( \text{Pr} \ oj_{\text{Comp}} \) index against the major accumulated risks for a more objective measure of the level of confidence in the overall project completion:

\[ \text{Pr} \ oj_{\text{Tensor}} = \begin{bmatrix} \text{Pr} \ oj_{\text{Comp}} & \text{Risk}_{\text{Time}} & \text{Risk}_{\text{Budget}} \\ \text{Risk}_{\text{Scope}} & \text{Risk}_{\text{Time}} \\ \text{Risk}_{\text{Scope}} & \text{Risk}_{\text{Time}} \end{bmatrix} \]  

(12)

- Here we have assumed only 3 primary Risk Types that are common industry key indicators, although with Agile, \( \text{Risk}_{\text{Time}} \) would weigh heaviest since it is the index to be fixed.

\[ \sum_{j=1}^{3} R_j = R_{\text{Time}} + R_{\text{Scope}} + R_{\text{Budget}} \]

- Where \( R_{\text{Time}} \), in addition to (4), should take into consideration the # of sprints complete versus the total # of sprints planned for which can be determined by tracking artifact / activity node traversals.

5. Findings and Experiments

In order to validate the above formalism, we evaluated a medium sized development project and reported its progress with discussed formalism. In our experience we have found that prior to the before mentioned project, almost all projects in the last decade used a plan driven approach.

Given this long standing tradition we were faced with an adoption problem and our formalism was evaluated on a single project to gain the acceptance necessary within the enterprise. To evaluate the methodology we used questionnaires at various stages of the project and reported the metrics based on the stakeholders distance to a task group. Initially, the approach was only accepted reluctantly as management didn’t believe that the project information and metrics could be trusted, yet over time, seeing the results and the accuracy of both deliverables and metrics, a dedicated governance group was established to further implement this approach.

6. References


An Approach to Reference Architecture Design for Different Domains of Embedded Systems

Liliana Dobrica¹ and Eila Niemelä²
¹Faculty of Automation Control and Computers, University Politehnica of Bucharest, Bucharest, Romania
²VTT Technical Research Center of Finland, Oulu, Finland

Abstract - The content of this paper addresses the issues regarding the reference architecture design for different domains in the context of a system of systems that is specific to today's embedded systems. The reference architecture contains core services of the domains included in abstract features package. The appropriate architectural style is provided by a knowledge base through service taxonomy. Services, commonality, variability management and rules for product derivation and configuration are the main issues considered in the architectural design process. Our contribution is the integrated vision based on our experiences and studies of the recent publications.

Keywords: cross domain product line, software architecture, service, quality, embedded systems.

1 Introduction

One of the most challenging engineering areas is embedded systems (ES). Among the requirements and constraints that have to be satisfied we can mention a higher diversity and complexity of systems and components, increased quality, productivity and reuse content, standardization, stricter requirements for time-to-market, fault tolerance and robustness. For a long perspective the design process for ES requires introduction of the higher level abstractions that are blurring the boundaries between hardware and software design. Also the domain technology causes exponential growth of the designed systems. Systems of yesterday become components of today. The fundamental principle stating that “any system consists of components” is common for any technical system as well as for a mature engineering discipline and it is sometimes called “a law of nature” [19]. Embedded systems (ES) are used as subsystems in a variety of domains, e.g. automotive, avionics, health care, industrial control, and consumer electronics. These domains include a variety of functions; however they are composed of a limited number of common software/hardware components. Nowadays in the embedded systems industries it has been recognized a significant duplication of development effort for hardware, software and services [1]. Due to the escalating complexity level of embedded systems, the technology trends and the bigger competition in the world market, a coherent and integrated development strategy for embedded systems is required. It becomes a research priority to create a generic platform and a suite of abstract components with which new developments in different application domains can be engineered with minimal effort [2]. Generic platforms, or reference designs, will be based on a common architectural style that supports the composition of systems out of pre-validated independently developed subsystems that meet the requirements of the different application domains. Given a core architectural style, different components can be created for different specific application domains, while retaining the capability of component reuse across these application domains. In this paper we propose a coherent and integrated development strategy for embedded systems that considers the architecture the main driver. We argue with our experiences in the software architectures design and analysis for embedded systems domains [11, 12] and other researchers’ recent studies that will be revealed during the paper. Our contribution is the synthesis of the most important issues of product line architectures (PLAs) in our development strategy for cross domain architecture design of embedded systems. We propose a service based approach for reference architecture design. The reference architecture is built based on structured existing domains knowledge.

2 Background

2.1 The Software Product Line development

In general the software product line development consists of two stages which are domain engineering and application engineering [4,5]. The main processes and results for each stage are described in Figure 1. Domain engineering can be divided in three main processes: Domain Analysis, Domain Design and Domain Implementation. The domain analysis consists in capturing information and organizing it as a model. Some methods, such as FODA (Feature-Oriented Domain Analysis) [3] propose a set of notations for the domain modeling using the notion of "features" to refer to products properties. The input represents domain knowledge and outputs are domain requirements. The domain design consists of establishing the PLA. The domain implementation realizes the PLA defined during the domain design as software components. The results represent core assets such as, domain requirements, PLA and components. Application
engineering builds products based on the results of domain engineering and users needs. During application analysis of a new system, the requirements from the existing domain model, which matches the user’s needs, are selected. Applications are assembled from the existing reusable components.

![Figure 1. Software Product Line development processes.](image)

Variability management is a key issue in the success of product line development. Approaches based on model driven architecture (MDA) [13] or ontology based support [14] have been identified in our researches. Deriving individual products from shared core software assets is still rather time consuming and expensive for a large number of organizations. A method for building product populations with software components has been applied with success in industries [8]. However various studies and experience reports of the current practice in industry [6,7,18] have identified several problems associated with products derivation. Knowledge externalization, variability management and scoping and evolution are three important areas of concerns. Under development are automated approaches [15] or specific tools [17] that are still immature for industrial processes.

### 2.2 Service architecture description and quality

Service architecture is a set of concepts and principles for specification, design, implementation and management of software services [20]. This definition is similar to software architecture that also includes the principles for guiding its design and evolution and has a strong influence over the lifecycle of a system [22]. Service architecture refers mostly to the software architecture of applications and middleware which is the software that is located between applications and the network layer. A middleware layer hides the underlying network environment complexity and masks heterogeneity of platform technologies from applications [21]. A service based approach supports customization and adaptation of services for limited resources, and configurations.

Under a software perspective architectural styles are recurring patterns of system organization whose application results in systems with known, desirable properties. As such, styles are key software design idioms. Examples of well known styles are layered, pipe-and-filter, client-server, push-based, peer-to-peer, and event-based. In practice an architectural style consists of rules and guidelines for the partitioning of a system into subsystems and for the design of the interactions among subsystems. The subsystems must comply with the architectural style to avoid a property mismatch at the interfaces between subsystems. Important contributions have been also identified regarding patterns used in embedded systems development, e.g. object analysis [16], software architecture [36], design of distributed real-time [34,35,37], fault-tolerant telecommunication system [33], real-time design [38], security [39], etc.

A design approach of services at architectural level has to consider quality attributes specification and standards. In IEEE 1061 [27], software quality is defined as a degree of software to process a desired combination of quality attributes. The software quality model [28] defines six categories of characteristics (functionality, reliability, usability, efficiency, maintainability, and portability) that are divided into subcharacteristics, which are externally or internally observable properties of software systems. The quality attributes can be classified into three categories. The first category defines a set of external quality attributes that is observable at runtime, such as performance, functionality, and usability. The second one is the set of internal quality attributes that cannot be discerned at the runtime, such as reusability and integrability. The third one includes quality in use attributes that are dependent on achieving the necessary external and internal quality. All three levels of qualities have to be measured. A Non-functional Framework (NFR) [29] provides a method for managing quality attributes during development by cataloging the types of nonfunctional requirements and their associated concepts and terminology, the development methods that aid in meeting quality requirements, and trade-offs and correlation among quality attributes.

### 3 Our Approach

#### 3.1 Design Description

Our approach is applied to design architecture for various application domains of embedded systems (e.g. automotive, avionics, consumer electronics, control, etc.). Embedded systems applications are doing control. Control activities could be measuring physical variables (sensing), storing data, processing sensors signals and data, influencing physical variables (actuating), monitoring, supervising, enabling manual and automatic operation, etc.

We consider an embedded system domain a collection of cooperating services that deliver required functionality. These
services may be executed in a networked environment and may be recomposed dynamically. We propose an approach that extends to three levels the architecture development method for embedded systems application domains. In Figure 2 the reference architecture includes core services and it focuses on commonality. Domain architecture includes domain specific services and it requires variability management concerns. Finally, the last level is dedicated to the set of products architectures. On this level rules for product derivation and configuration should be included.

![Cross domain architecture design](image)

**Figure 2. Cross domain architecture design.**

A feature model is a prerequisite of this approach. We propose a feature model as described in Figure 3. This model is essential for both variability management and product derivation, because it describes the requirements in terms of commonality and variability, as well as defines the product line dependencies. Feature types may be mandatory, optional, or optional alternative. Dependencies specified by the model are called composition rules between domain features. The requires rule expresses the presence implication of two features and the mutually exclusive rule captures the mutual exclusion constraint on feature combinations.

Reference architecture defines quality attributes, architectural styles and patterns and abstract architectural models (Figure 4). Quality attributes clarify their meaning and importance for core service components. The interest of the quality attributes for the reference architecture is how the quality attribute interacts with and constrains the achievement of other quality attributes (i.e., trade-offs) and what the user’s view of quality (i.e., quality in use) is. Embedded systems services have to meet many quality attributes, such as modifiability and integrability. Modifiability of a service can be divided into the ability to support new features, simplify the functionality of an existing system, adapt to new operating environments, or restructure system services. Integrability measures the ability of the parts of a system to work together and it depends on the external complexity of the components, their interaction mechanisms and protocols, and the degree to which responsibilities have been cleanly partitioned.

Interdependencies and tradeoffs also exist between quality attributes.

![UML Features model](image)

**Figure 3. UML Features model.**

The styles and patterns are the starting point for architecture development. Architectural styles and patterns are utilized to achieve qualities. The style is determined by a set of component types, the topological layout of the components, a set of semantic constraints and a set of connectors [30]. A style defines a class of architectures and is an abstraction for a set of architectures that meet it. A small taxonomy identifies five style categories: independent components, data-flow-centered, data-centered, virtual machine, and call-and return style [30]. An architectural pattern is a documented description of a style or a set of styles that expresses a fundamental structural organization schema applied to high-level system subdivision, distribution, interaction, and adaptation [31]. Design patterns, on the other hand, are on a lower level. They refine single components and their relationships in a particular context, and idioms describe how to implement particular aspects of components using the given language [32].

In this way the reference architecture creates the framework from which the architecture for new embedded systems is developed. It provides generic architectural services and imposes an architectural style for constraining specific domain services in such a way that the final product is understandable, maintainable, extensible, and can be built cost-effectively. Potential reusability is highest on the reference architecture level. Core services and the architectural style of the reference architecture are completely reused on the domain architecture level.

Reference architecture is build based on a service taxonomy. We adopted the idea from WISA [26] of an existing knowledge on software engineering that is integrated and adapted to service engineering for embedded systems. The standards related to each embedded system domain, applicable architectural styles and patterns and existing concepts services and components are the driving forces of embedded systems development. A service taxonomy defines the main categories called domains. Typical features that have been abstracted...
from requirements characterize services. The purpose of the service taxonomy is to guide the developers on a certain domain and getting assistance in identifying the required supporting services and features of services.

<table>
<thead>
<tr>
<th>Taxonomy of Constraints and Requirements</th>
<th>Standards</th>
</tr>
</thead>
<tbody>
<tr>
<td>Styles and Patterns</td>
<td>Quality Attributes</td>
</tr>
<tr>
<td></td>
<td>Service Taxonomy</td>
</tr>
<tr>
<td>Core Services</td>
<td>Reference Architecture</td>
</tr>
</tbody>
</table>

Figure 4. Reference architecture realization.

Domain architecture describes ready made building blocks that assist application/products developers in using specific domains services. When the reference architecture has been defined, the existing components and services are considered as building blocks in the architecture of set of products. The domains services provides variable assets repository. Variability appears in functional and non-functional requirements (including quality attributes). A structured domain architecture repository may be provided. A schema for this repository has to be defined in a form of relationships between services. In this way we are mapping domain specific services to core abstract services. Specialization relation is a solution to be used for variability management. Modeling run-time quality attributes variability requires tool support. Approaches [10] are under development and refining. Monitoring mechanisms, measuring techniques and decision models for making tradeoffs should be better defined and validated.

A product architecture consists of concrete services derived and configured based on the rules. The goal of product derivation is to reach a configuration of the product line in which necessary variabilities have been bound. The decision model for binding specific services of a domain to a product may be in a tabular form or a more comprehensive tool based on the feature types and composition rules. By selecting a consistent set of features required for the individual product, the corresponding domain specific services that realize those features are selected from the domain repository to constitute the product.

### 3.2 Architectural Development

There are several architectural development approaches that can be adopted to service based embedded systems development. The Model-driven Architecture [23] is an approach that guides the specification of information systems. The idea is to separate descriptions of functionality from the implementation specifications and thus increase the integrability and evolvability of a system. Implementation independent descriptions of functionality last longer than implementation specifications that change as soon as a better technology is available. In MDA, a model means a specification of a part of the function, structure and/or behavior of a system. The specification expects either textual or graphical language with strictly defined syntax and semantics.

Another design approach concentrates on multiple views of architecture. An architectural view is a representation of a whole system from a perspective of a related set of concerns [22]. Among these approaches [24,25], there is no agreement on a common set of views or on the way to describe software architecture. The need for different architectural views depends on three issues: the size, the domain and the number of different stakeholders [26]. Although a multiple view approach helps in developing software products, it is easy to introduce errors and inconsistencies in a multiple view model. It is therefore necessary to provide support for consistency checking among the views.

### 4 Example

#### 4.1 Description

In this section we describe a simple multiple domains example to illustrate our vision for product line architecture development. The cross-domain reference architecture considers measurement control, data acquisition control and data management domains. Our example is abstracted from our experiences with the architecture design of a scientific onboard silicon X-ray array (SIXA) spectrometer control software. SIXA is a multi-element X-ray photon counting spectrometer. It consists of 19 discrete hexagonally-arranged circular elements and specific domain hardware architecture. The SIXA measurement activity consists of observations of time-resolved X-ray spectra for a variety of astronomical objects. Figure 5 introduces the context view of SIXA considering it a measurement controller. External elements that interface with our measurement controller are a command interface and physical devices (detectors) representing sensors and actuators. The system is programmed and operates using a set of commands sent from a command interface.

![Figure 5. Context view.](image-url)

The role of the spectrometer controller is to control the following measurement modes:
• Energy Spectrum (EGY), which consists of three energy-spectrum observing modes: Energy-Spectrum Mode (ESM), Window Counting Mode (WCM) and Time-Interval Mode (TIM).
• SEC, which consists of three single event characterization observing modes: SEC1, SEC2 and SEC3.

Each measurement mode could be controlled individually. A coordinated control of the analog electronics is required when both measurement modes are on.

Figure 6. Mapping features into packages.

The analysis of requirements for domain engineering and scoping the product line has a result in a features model. The features model has been structured in packages (see. Figure 6). The with-reuse aspect of reusability is described by the abstract features of the PLA. The abstract features encapsulated into three main abstract domains MeasurementController, DataManagement and DataAcquisition, are completely reused in all product members. The AbstractSpectrometerFeatures package has the highest degree of reusability but also the highest degree of dependability. The abstract features depend on the commonality between EGY and SEC features. A change in the problem domain of one of the three products is mostly reflected in the degree of reusability of the abstract domain features.

The sets of products that could be derived from the domain specific services during application engineering are:

• P1 – EGYController includes specific services of a standalone control of EGY measurement mode;
• P2 – SECController includes specific services of a standalone control of SEC measurement mode;
• P3 – SECwithEGYController includes specific services of coordinated control.

4.2 Cross-domain architecture of spectrometer controllers

The spectrometer controller cross domain architectural approach is described in Figure 7.

Reference architecture. Looking top-down, the Reference Architecture encapsulated in the Measurement <<MultipleDomain>> is composed of three core abstract <<Domain>>s: MeasurementControl, DataAcquisitionControl and DataManagement.

In each core <<Domain>> abstract features are collected. The MeasurementControl is responsible for services of starting and stopping the operating mode for data acquisition according to the commands received from the command interface and according to the events generated in other parts of the software. DataAcquisitionControl services collects events (science data) to the spectra data file during observation of a target. This abstract service includes as well as hides data acquisition details. DataManagement abstract services provides interfaces for storing science data - opening/closing/writing the data files, hiding data storing details and controlling transmission of the stored data to command interface.

Domain architecture. Domain architecture consists of domain specific services and variability management services. Each of the three core services is specialized in domain specific services. For example, MeasurementControl is specialized in StandAloneControl (SAC) and Coordinated Control (CC), DataAcquisitionControl (DAC) is specialized in EGY_DataAcquisitionControl (EGY_DAC) and SEC_DataAcquisitionControl (SEC_DAC), DataManagement (DM) is specialized in EGY_DataManagement (EGY_DM) and SEC_DataManagement (SEC_DM). Domain architecture includes services associated to variability management.
**Product architecture.** Product architecture for the sets of products includes rules for product derivation and configuration.

<table>
<thead>
<tr>
<th>Domains</th>
<th>Specific Service</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measurement Control</td>
<td>SAC</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Acquisition Control</td>
<td>EGY_DAC</td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>SEC_DAC</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Data Management</td>
<td>EGY_DM</td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td></td>
<td>SEC_DM</td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

In Table 1 the set of products are horizontally distributed and the domain services are dispersed vertically. Each cell \( t_{ij} \) of the table is marked if product \( P_j \) uses component \( C_i \). For example, two products, P1 and P2, includes a SAC service of the measurement control domain.

In our example, we used a tabular form for the decision model. When the problem complexity increases, a more elaborated tool is required and is a subject for our future research. However, our incremental design and analysis approach is based on a systematic service oriented approach, which is more practical, easier to follow and benefits of advantages provided by service engineering.

Future research work is needed to develop systematic ways of bridging other requirements and constraints from a taxonomy to reference architecture. However, this paper presented the main concepts and justified why these concepts are required.

## 6 REFERENCES


Bridge Specifications for Separated Concerns

A. Fatwanto¹ and C. Boughton¹
¹Department of Computer Science, The Australian National University, Canberra, ACT, Australia

Abstract - In this paper we propose a method to create simple bridge specifications for separated concerns. The method was developed in the context of a concern-oriented model-driven software development framework [1]. We establish the ways to specify and model simple bridges to connect between separated concerns. We also further describe concepts related to bridging such as mapping, weaving, and transformation (refinement and translation).

Keywords: Separation of concerns, bridge, model-driven development, concern-oriented approach, transformation.

1 Introduction

Separation of concerns has been offered as a practical approach to cope with the development of complex software systems. Intuitively, handling each part of a system at a time is easier compare to deal with the whole system simultaneously. This principle also applies to Model-Driven Development (MDD). By implementing separation of concerns into MDD, a system will be decomposed based on concerns identified during initial stage of development. Subsequently, each concern within a system will be analyzed, specified and then modeled separately. Once concerns have been completely modeled, the next step is to specify the links (bridges) between these separated concerns.

This study extends our previous work regarding Concern-Oriented Model-Driven Development (COMDD) framework [1]. One aspect that remains unspecified within our previous work is about bridges between separated concerns. Since bridges in an integral part of the development framework hence the method to specify bridges is very important within this context.

In this paper we propose a method to specify bridges between separated concerns. Our propose method accommodates the technique to create simple bridges. We follow the classification of bridges as given in [2] and [3].

Our main contribution is providing a method to specify simple bridges at analysis level. Bridges are specified at high level of abstraction, similar to the specification of concerns in our COMDD framework, and they both architecture- and implementation-independent. Bridges are developed independent from any design aspects. Therefore this method could raise the level of reusability and portability.

The remainder of this paper is structured as follows: section 2 discusses works related to this topic, section 3 explains the concept of our proposed method along with examples, section 4 discusses some concepts related to transformation, and finally chapter 5 contains summary and description of future works.

2 Related Works

Currently only few literatures in MDD (especially which implement Shlaer-Mellor methodology) provide description on bridge specification. [4] describes the use of mappings as a way to specify bridges. Mappings define the correspondence between object instances in the source domain and the ones in target domain. Mapping can be performed either using mapping rules or enumeration. Mapping rule state the correspondence function between object instances in different domains. It is formed in general sense hence it can be used uniformly. Enumeration is used for correspondences which can not be simply (uniformly) mapped. Mapping can be carried out either for stating the correspondence of object instances from different domains in the same level of abstraction or transforming the object instances from the source domain to the target domain in different level of abstraction. Even so, Shlaer and Mellor did not provide any explanation about the way to weave object instances of different domains.

[5] proposes a method to design and construct bridges for different domains using interfaces. The domains are typically black-box therefore they will communicate with other domains only through interfaces. Grosberg defined two sorts of interfaces: one for provided service and the other for required service interface. Within his method, a bridge is built by linking two different types of interfaces between client and server domains. Examples given in his paper were using C++ as the notation (implementation language) for bridge specifications. Nevertheless, [5] only provides the simple bridge specification method at implementation level.

An early classification for bridges was given by Mellor and Balcer in [2]. They classified bridges as anonymous explicit bridges and implicit bridges. An anonymous explicit bridge represents signals or synchronous operations invocations where the domains in which they refer to remains
anonymous. Implicit bridges represent the way on how to one domain can be linked to other domains where explicit bridges are difficult to be applied. [2] categorizes bridges to the model compiler as a kind of implicit bridges. In their book, Mellor and Balcer gave an early classification for bridges but they did not provide a sufficient method to develop bridge specifications.

In [3], Flint and Boughton classified bridges as explicit and implicit. Explicit bridges represent signals to and from external domains and the invocation of operations on such domains. Implicit bridges, on the other hand, represent a set of rules which specify the use of subject matters in the bridged domains. Flint and Boughton further classified implicit bridges as mapping, transformation and weaving. Mappings are used to state correspondence between subject matters from two different domains. Transformations handle with transforming subject matter of one domain into another. Meanwhile, weaving deal with woven subject matters of one domain with subject matters of other domains. Although a good classification was provided, the method to do bridge specification and modeling did not provided.

[6] defines two types of bridges: simple and advance. This classification is quite similar with the classification given by [3]. Simple bridges represent the explicit connection between subject matters in two separate domains. Advance bridges represent the correspondences (mappings) between subject matters of different domains. To build the bridges, Raistrick et al. use interface as in [5] except that the interfaces are called as terminator class. Simple bridges use non-counterpart “terminator class” for the interfaces. Meanwhile, interfaces for advance bridges are counterpart “association terminator class” for counterpart associations and counterpart “specialization terminator class” for counterpart generalizations. The bridges are specified using Action Specification Language (ASL). The difference between synchronous and asynchronous signals across domains is stated through the description of contract type in the bridge specification. However, the classifications of contract types, i.e. open, closed non-blocking, and closed blocking, are rather complicated compare to the classification between synchronous and asynchronous communications. Furthermore, weaving between subject matters of separated domains had not been thoroughly elaborated.

A broad classification for mapping was presented in [7]. In this book, mappings were classified as horizontal, vertical and merging mapping. Horizontal mappings deal with models transformations from source domain to the target domains in the same level of abstraction. Horizontal mappings can be in the form of representing or migrating mappings. Representing mappings state the correspondences of elements in one metamodel with the associated elements in another metamodel (in the same level of abstraction). Migrating mappings state the correspondences of elements from one platform to another platform. Vertical mapping deal with models transformations from source domain to the target domains in different level of abstraction. It can be in the form of refining or abstracting mappings. Refining mappings state the correspondences of elements from source domain to the target domains in the lower level of abstraction. Abstracting mappings do the same thing for the other way around. Meanwhile, merging mappings state the function to weave elements of multiple source domains to combine them into a single target domain. It is used to map the elements from different domains that do not explicitly refer to each other. Even though they gave a broad classification on mapping, the method to do mapping specifications was not presented.

Another work that extended Grosberg’s method was proposed in [8]. Löfqvist offered a method to specify implicit bridging for separated domains using domain interfaces as in [5]. He then developed the method with a technique to state the correspondences between elements of different domains by introducing five types of mappings: counterpart object mapping, counterpart event mapping, counterpart attribute mapping, state to attribute value mapping, and new state transition to attribute value mapping. Action Language (AL) was used as the notation to specify mappings. Unfortunately, as in [5], Löfqvist only provide his specification method at implementation level. Moreover, the method to create weaving specification between entities of different domains do not presented in this thesis.

3 Bridge Specifications

In order to develop a complex software system using concern-oriented MDD framework, firstly a system will be decomposed into a number of different autonomous concerns each represents subject matter of interest. Each separated concerns will be analyzed, specified and modeled subsequently. In this framework, Shlaer/Mellor Object-Oriented Analysis (OOA) method is used as the modeling paradigm and executable UML is used as the modeling notation [2] and [6]. Although most concerns are modeled using executable UML, there are few concerns that are best modeled using other kind of notation (language) specifically developed for their nature. For example, Graphical User Interface (GUI) concern would be better modeled using a more intuitive notation even if it can be modeled using executable UML. Sometimes few concerns could have been realized (implemented), for instances concerns related to off-the-self components, reusable components, legacy systems, etc. Hence they do not necessitate to be re-modeled anymore. Once concerns have been completely specified and modeled, the next step is to develop bridges for separated concerns.

A bridge is defined as a specification of a set of mechanisms to integrate two or more concerns. Adapting the classification in [2], [3] and [6], we differentiate bridges as simple and complex. A simple bridge specifies a set of mechanisms to handle signals to and from entities reside in external concerns and the invocation of operations in such
3.1 Simple bridge Specifications

When a particular entity wants to connect to external entities for sending signals or invoke operations, then the integrating mechanisms used in this case is simple bridge. This is actually a basic (simple) form of bridge. A simple bridge specifies the mechanisms to send and receive messages between different concerns. The messages can be in the forms of signals or inputs for the operations invocation.

Simple bridges can be developed by constructing interfaces within every concern. Two associated interfaces will eventually be mapped and then specified as a simple bridge. Interfaces symbolize the gate for a concern to communicate with other concerns. Construction of interfaces has an objective to preserve the principle of separation of concerns. It avoids strong coupling between separated concerns. There are two approaches that can be applied in developing interfaces. First, by creates two interfaces for each concern. One interface encapsulates all provided services offered by the particular concern. Another interface encapsulates all required services for which the particular concern needs. All communications to and from the concern will then be directed through the interfaces. This kind of approach was first introduced in [5] and extended in [8]. Second, by creates a number of interfaces that each represent a concern to which a particular concern communicates. Each interface encapsulates the specification of provided or required services of a concern. Same as the previous approach, all communications to and from the concern will be passed through these interfaces. This approach was proposed in [2] and [6]. We will use the second approach within our work.

An interface which encapsulates provided services becomes the source of messages to the internal entities within a concern. It acts as a mediator for the particular concern in receiving messages from other concerns. Messages from a provided services interface can only be sent to the internal entities within its concern. On the other hand, an interface which encapsulates required services becomes the destination of messages originated from internal entities which are going to be passed onto other concerns. It accepts the messages and then forwards them to the interface of other concerns which provide the specific required services. Messages from a required services interface may only be sent to a provided services interface at another concern.

Inter-concerns communications, similar to intra-concerns communications, can be in the form of synchronous or asynchronous. Messages sent in asynchronous communications are called signals. When a signal is sent, the sender continues with its business and does not have to wait any reply from the communicating partners. In synchronous communications, sender who sends inputs to invoke operations at the communicating partners has to wait for replies before it can continue with its business.

For modeling purpose, an interface will be represented as a class with <<interface>> stereotype. Interface classes may not have any attribute (data member). They can only have static member functions which mean that these classes are static thus can not be instantiate. The interface classes are created for representing concerns with which a particular concern communicates. Therefore they do not instantiate since it would be only one instance for each concern. Further, interface classes must not hold any information about states and objects.

In this study, we extend the notation of executable UML by adding a dotted arrow. The dotted arrow should be supplemented with annotations to describe the flowing messages. This notation represents the flow of messages from and to interface classes where its direction is symbolized by the arrowhead. Close (triangle) arrowhead represents synchronous communication, whereas open (v) arrowhead represents asynchronous communication. In the event of synchronous and asynchronous communications occur between the same entities, the communication symbol will be represented by the synchronous one. The reason is that synchronous communication need more consideration compare to the asynchronous one.

In good modeling practices, each interface class is named in the context of host concern. The name of each interface class should reflect the role of the particular class in the context of concern in which it resides. It is not recommended to name the interface classes in the context of the counterpart concerns to avoid coupling between concerns.

An interface class may only encapsulate one of either provided or required services. It is quite uncommon to encapsulate both types of services within one interface class. Communications in simple bridges mostly occur between one concern as a client and other concerns as servers. Whenever an interface class encapsulates both services in it, this is the sign that the particular concern at the same time plays as both client and server to the concerned that the interface represents. In a simple software structure where simple bridges link the separated concerns, there is a rule called assumptions-requirements pair. A client assumes that its required services are placed somewhere in the servers as their requirements. Consequently, communications between a client to its server are unidirectional (one-way). Hence when a class plays as both client and server at the same time it violates the unidirectional principle of the communications. This condition excludes the reply messages which go back to the originated interface class as a respond of synchronous communications.
Figure 1 depicts a snippet of class diagram from a banking application concern. Requirements of a banking system state that every occurring transaction must be logged for historic record. The requirements also say that every transaction must be charged for any liable fees. Since the banking application concern only specifies the business rules for banking operation, it assumes that the logging and charging mechanisms are specified at other specific concerns. We omit any details such as attributes and operations from the above diagram for brevity.

![Class Diagram](image)

**Figure 1.** Snippet of class diagram for banking application.

Interface classes at figure 1 are a type of required interface. The Transaction_Record class represents a concern which is assumed encloses subject matter that can fulfils the requirements of recording occurring transactions. Meanwhile, the Transaction_Fees class represents a concern which is assumed encloses the subject matter that may fulfils the requirements of calculating transaction fees. Each interface class should be further specified to provide clear description of bridges.

Figure 2 shows the specification for Transaction_Fees interface class. The <<interface>> stereotype at the top compartment (name compartment) states that this class is a type of interface class. Subsequently, class name is given followed by its acronym inside the <…> sign. One or two sentences long description explains the role of this class accompanies the class name statement. The second compartment (description compartment) lists a number of services that are required or provided by this interface. Annotations placed inside the <…> sign following the service statement explain the type of this particular service. In this case, the particular service is a required service with asynchronous communication type. As in the class description, each service statement also accompanied with the descriptions of what this service actually mean. Specifications of operations which have been stated previously are placed into the last compartment (method compartment). In our example, we specify the operation using action specification language. The operations are denoted along with input and (if any) output parameters and accompanied by their type. An acronym followed by double colon (::) state the name of interface class in where a particular operation reside. These specifications may also use any kind of notation as long as it preserves independency from the implementation. The Transaction_Fees class will then be bridged to its counterpart provided class located at the concern which realizes the services. The same case will apply to the Transaction_Record class.

![Interface Specification](image)

**Figure 2.** Specification for Transaction_Fees interface class.

![Interface Specification](image)

**Figure 3.** Specification for Transaction_Record interface class.

At the counterpart side, interfaces classes for those provided services are created at the associated concerns. For example, a fees management concern is developed to specify the mechanisms of liable fees calculation in a banking system. It is assumed that fees management concern can fulfils the requirements of calculating transaction fees. Hence an
interface class which encapsulates a provided service of calculating transaction fees is created. This interface class might also encapsulate other provided services which assumed to be required by the same concern that need transaction fees calculation. Fragment of class diagram for fees management is depicted in figure 4. Same as figure 1, we also omit some details in this figure for brevity.

![Figure 4](image)

Figure 4. Snippet of class diagram for fees management.

Interface class “Client” in figure 4 is a type of provided interface. It represents a concern which is assumed encloses subject matter that need certain services provided by the fees management concern. The specification for “Client” interface class is shown in figure 5.

![Figure 5](image)

Figure 5. Specification for “Client” interface class.

Finally a bridge between interfaces can be specified. A required interface class is going to be connected to a specific provided service interface of another concern within which its required services is assumed to be fulfills. For instance, in our example we will build a bridge between Transaction_Fees and “Client” interface class. It is assumed that the required services stated in Transaction_Fees will be fulfilled by the provided services stated in “Client”. Bridge specification for this example is depicted in figure 6.

![Figure 6](image)

Figure 6. Specification for a bridge between banking application and fees management concern.

A construction for a simple bridge specification as shown in figure 6 comprises of three compartments. The uppermost compartment is called name compartment. It contains the <<Bridge>> stereotype to state that this is a bridge specification. It also contains the names of concerns which are connected by this particular bridge. At the second compartment, called pair compartment, contains the correspondences between required services from the required interface with its associated provided services of the provided interface. The last compartment, called method compartment, contains the bridge operations that realize the communication between a required interface with its associated provided interface. In this example, we use action specification language to state the bridge operations. Any bridge operation statement is preceded with the acronym of its particular concern and interface class’ name which each followed by a double colon (::) to denote in where the operations actually located. Input and (if any) output parameters along with their associated type accompany the operations statements.

Communications between client and server concerns can be illustrated using Concern Collaboration Diagram (CCD). For brevity purpose, some details should be omitted from the diagram. Figure 7 shows the CCD between banking application and fees management concern.

Collaboration between banking application and fees management concern as shown in figure 7 is a specification of communication between these two concerns. A Concern Relationship Diagram (CRD) can be used to lay out the high level view of communications between separated concerns. The fragment of CRD for banking system is depicted in figure 8.

Relation between banking application and fees management concern is represented as a synchronous communication. This relation had been specified in the CCD as depicted in figure 7. Meanwhile, the relation between banking application and logging is represented as an
asynchronous communication unless it will later be found that any synchronous communication occurs between them.

Figure 7. CCD between banking application and fees management concern.

3.2 A Brief Description of Complex Bridges

Complex bridges, as oppose to the simple one, deal with any integration mechanisms between two or more concerns which are complex in nature. For example, it is difficult to specify bridges between application concerns such as banking application with its associated GUI. Few entities within these concerns might be bridged easily. For instance, a button click event in GUI concern can be bridged to a transaction event at the banking application concern. However, most concerns have complex association with the others meaning that the bridges should also be complex in nature. A number of transactions in banking application may correspond to a list of text in GUI. The way to specify this association is complex bridge type.

A concern might have more than one association with others. To complicate the previous example, imagine a banking application that may be related to many other concerns such as GUI, security, fault tolerance, etc. The bridges between these concerns can be inter-related and complex. Figure 8 shows a snippet of typical CRD for a banking system.

A bidirectional (two-way) arrow having closed (triangle) arrowhead at both end symbolizes a complex bridge. This notation is different from simple bridges where the communications between concerns are classified as synchronous and asynchronous by which the symbols are distinctive.

The method to specify complex bridges is quite different from the one for simple bridges. Simple bridge specifications are still rather coupled with their associated concerns. Complex bridge specifications are more loosely coupled (separated) with the concerns which they integrate.

Figure 8. A snippet of CRD for banking system.

Compare to simple bridges, complex bridge specifications are more independent towards their associated concerns. Therefore, complex bridge specifications can be composed separately from their associated concerns. Figure 9 depicts the highest level view of complex bridges.

Figure 9. A snippet of complex bridges for banking system.

Complex bridges can be in the forms of mapping, weaving, or combination of both. Mapping states the correspondences (pairing) between entities of one concern with entities of other concerns. For example, mapping between a number of transactions in banking application with a list of text in GUI. Meanwhile, weaving deal with the mechanisms to weave between entities of one concern with entities of other concerns. For instance, some aspects of security concerns such as authorization, authentication, protected account, encryption, etc must be woven together with aspects in banking application concern.

Mappings and weavings sometimes might occur between a concern with two or more other concerns. In this case, complex bridges become more complex.

In some cases, the specifications of complex bridge are supplemented with architectural or design aspects. These refinement practices may breach our rules of having architecture- and implementation-independent bridges. However, refining bridge specification to be supplemented with architectural or design aspects can help in transformation processes at later development phase.
4 Transformation

We define transformation as an activity which comprises of two portions: refinement and translation. Refinement is a process of refining (supplementing) a concern specification with some design decisions. This process changes the level of abstraction. We call this process as vertical transformation. The process of transforming PIM into PSM is categorized as refinement. Translation is a process of porting a concern specification from one format to another format without adding and/or abstracting any detail. This process does not change level of abstraction. We call this process as horizontal transformation. The transformation from an implementation in C++ into implementation in Java is a kind of horizontal transformation.

In the context of bridging and transformation processes under concern-oriented MDD framework, there are two approaches that can be followed: simultaneous and non-simultaneous transformation. The first approach uses a merge-first-transform-later mechanism. Initially, models of the separated concerns at the same level of abstraction are bridged to get a single combined model. In here, bridges are quite simple. They merely specify the signals to and from entities in external concerns and the invocation of operations (functions/procedures/subroutines) on such entities. Subsequently the combined model will be transformed (refined and translated) into implementation based on rules enclosed in the architecture specification. This approach is suitable for handling relatively simple systems where most of the bridges are simple (simply to developed). Simple bridges are commonly used to bridge a concern with realized concerns and simple service concerns (intermediate abstraction) such as logging, I/O, etc. Currently, CASE tools implementing Shlaer/Mellor (translative) methodology which available in the market is supporting this approach.

Meanwhile, non-simultaneous transformation approach is suitable for handling complex systems where a lot of complex bridges involved. The mechanism used for this approach is bridges artifacts creation. Using this mechanism, bridges for the separated concerns will be specified and modeled. Complex bridges are rather complex since they comprise of specification for mapping and weaving of separated concerns. Furthermore, it is usual for a bridge within complex systems to relate more than two separated concerns. For that reason, complex bridges must be specified separately apart from the concerns they connected. During the bridging processes, the complex bridge specification artifacts are commonly refined with some design decisions. Hence, the transformation processes are different from the previous approach. In here, they proceed in two steps (refinement and translation) instead of performed simultaneously. Some parts of refinements are performed during bridging processes where the results are reflected at the bridge specification. The others are performed during compilation time where the refinements rules are indicated in the architecture specification. The latter are similar with the transformation mechanism within the first approach. Presently, there is no available open source or proprietary CASE tool which supports this approach.

5 Summary and Future Work

A method to specify simple bridges are proposed in this paper. This method is developed in the context of COMDD framework. Bridges are specified at analysis level and they are both architecture- and implementation-independent. They are developed independent from any design aspects therefore could raise the level of reusability and portability.

We plan to extend our study to establish a method for specifying complex bridges. We also want to specifically study the transformation from a single combined system specification which has been bridged down into its implementation.

6 References

A New Approach For Solar Analysis Of Buildings

A. Mojtaba Samimi\(^1\), B. Laya Parviz Sedgby\(^2\), and C. Morteza Adib\(^1\)

\(^1\)Faculty of Architecture and Urban Planning, Shahid Beheshti University, Tehran, Iran
\(^2\)Faculty of Art, Tarbiat Modares University, Tehran, Iran

Abstract – One of the greatest challenges of architects is to design buildings not only receive more from the kind face of the sun; but also protect themselves from the sun’s unkindness. If a building could walk and change its form or skin, the solar problems of architectural design might be solved a little bit easier on the paper; but most of the buildings are standing and static creatures, so the design should be very intelligent to work on both cold and hot times through the year. As the properties of each location like temperature pattern and surroundings geometry affect the design solutions; a proper approach for solar analysis of buildings is needed to answer important questions in solar architecture like the effect of the form and orientation of buildings, the proper amount of openness in each direction and the effect of shading or reflecting surroundings.

Keywords: Analysis, Architecture, Direction, Local, Solar.

1 Introduction

The paper presented here is an effort to eliminate the insufficiencies in architectural solar analysis of the form and orientation of buildings, the proper amount of openness in each direction and the effect of shading or reflecting surroundings.

To achieve that, in the first step the position of the sun in the sky and its estimated direct and diffused radiation in each moment are needed. These values are calculated for more than fifteen major cities of Iran using local parameters of each site. In the second step by defining kindness and unkindness parameters of the sun in each moment from total received radiation and the amount of need to shade or shine, a local face of the sun for each city is modeled. Finally, considering that each face of the building in according to its direction, slope and surroundings, receives different yearly positive and negative scores from the local face of the sun, a proper algorithm has been composed to show final comparable results.

Now we are able to present the solar situation of different building masses with different reflecting or shading surroundings in each city through the year as a new passive solar diagram.

2 The Approach

The sun has two different faces. The kind face of the sun appears in cold times; and its unkind face appears in hot times. In according to the properties of each location, kind and unkind faces of the sun differ from place to place, these local faces of the sun more than whatever depend on two parameters: 1\(^{st}\), the intensity of direct and diffused solar radiation in each moment; 2\(^{nd}\), the amount of need to shade or shine in according to the difference between comfortable temperature and outdoor temperature in each moment. But
first of all a proper algorithm is needed to calculate the position of the sun in the sky.

2.1 The position of the sun in the sky

Knowing the position of the sun in the sky for each location and in each time is the base of all solar studies on the earth. Although solar charts published for different altitudes may answer a part of questions in this field; but doing researches which need detailed solar modeling would be impossible without calculating momentarily position of the sun in the sky.

As the position of the sun in the sky is defined with two azimuth and altitude parameters; some prescriptions are available to calculate these parameters using trigonometry functions; but computer programmers know that using these functions for several reasons is not safe, like applying functions of \( \sin \) and \( \cos \), also dividing variables by variables that may cause division by zero error and etc. So, the computer programmer should control the results using several “if … then … else …” statements that increase the complexity of calculations and still may not work correctly in some cases!

\[
\text{altitude} = A \sin[\sin \delta \sin \phi + \cos \delta \cos \phi \cos(HRA)]
\]

\[
\text{azimuth} = A \cos[\sin \delta \cos \phi - \cos \delta \sin \phi \cos(HRA)] / \cos \alpha
\]

So considering the complexities these prescriptions create\(^1\), here another algorithm has been created which returns three parameters for positioning the sun in the sky. These parameters are x, y and z which locates a point on the sky dome with radius of one unit. This method is more simple than previous methods, and calculates the position of the sun in the sky for each latitude on the earth from +90° north pole to -90° south pole without any bug.

Table 1. SunPosition function

<table>
<thead>
<tr>
<th>Function SunPosition</th>
<th>Latitude</th>
<th>Date_Angle</th>
<th>Hour_Angle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declination = 23.5 * sin(Date_Angle - 180.0)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a = sin(Declination)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b = cos(Declination) * -cos(15.0 * Hour_Angle)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x = cos(Declination) * sin(15.0 * Hour_Angle)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y = -(a * cos(Latitude) + b * sin(Latitude))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>z = -a * sin(Latitude) + b * cos(Latitude)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>return #(x, y, z)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(^1\) The programmers of ECOTECT have pointed on this issue in detail: http://squ1.org/wiki/Solar_Position_Calculator

Figure 2. The sun path on the sky dome for 30°N Latitude.

It is necessary to mention that by applying this algorithm the night position of the sun could be calculated and plotted on the sun charts.

Figure 3. Night and day paths of the sun for 30°N Latitude.
2.2 The sun’s kind and unkind faces

However, the geometry of the sun path in each location plays an important role in solar analysis, there are other factors which have remarkable effects on this matter such as the atmosphere of the earth and the local properties of weather. Considering the effects of these factors; in each location, the sun’s kind/unkind face at each moment depends on two parameters: radiation received and need to shade or shine. When the amount of total radiation doubles, the amount of sun’s kindness is doubled; also if the outside temperature is increased two times warmer, the amount of sun’s unkindness would be increased in the same way. It is necessary to mention that the parameter “need to shade or shine” is about the thermal face of the sun and not about its illuminative face.

As the changes of temperature in two cycles of the year are not the same, two different diagrams are needed to model yearly local face of the sun for each location.
2.3 The method of weighting +/- scores

As each face of the building receives different quantities from sun’s kind and unkind faces in according to its direction through the year, a proper algorithm is needed to weight these yearly positive and negative scores. The first way may be used, is to add positive and negative scores. However this method defines one side of the problem, it does not define the other side. For an example the face that receives 10 positive scores and has no negative score is better than the face with 30 positive scores and has 20 negative scores; however the sum of these cases are the same. That’s why the first case is always good and the other is not. As a matter of fact the other effective parameter is the state of purity. This is the reason why \( \Sigma(3,-1) \) equals to +1!

Table 2. Weight function

<table>
<thead>
<tr>
<th>Function</th>
<th>Weight P N =</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sum = P - N ; Purity = (P - N) / (P + N)</td>
</tr>
<tr>
<td></td>
<td>return (Sum * Purity)</td>
</tr>
</tbody>
</table>

Figure 6. What is the weight of positive and negative scores?

Figure 7. Positive and negative scores for each direction in the city of Yazd collected in different months.

Figure 8. The factors that affect the final comparable result are the sum (above) and the purity (below).

Figure 9. Example cases where a light gray ball = one positive score and a dark gray ball = one negative score.
Scoring all building faces could be resulted by applying this algorithm to the positive and negative scores of each direction and slope.

The below diagrams present important points for the city of Yazd like these:

* Although S.W. faces receive maximum negative scores they are better than N.W. faces with a little positive scores.

* Although +40° tilted S.E. faces receive maximum positive scores, it is much better to use -20° tilted S.E. faces or vertical S.E. faces with small horizontal overhang.

Figure 10. Top: Complex overlay of positive and negative scores for every direction and slope in Yazd. Middle: Two layers of sum and purity created by the algorithm to analyze the complex situation. Below: Simple comparable passive scores for 360° directions and ±90° slopes for the city of Yazd.
Comparable passive scores could be plotted in a full-spherical diagram to show the effect of all possible directions and slopes of external building faces. For an example by taking a look at the below diagram the architect understand that in the city of yazd directions from south to east are the best choices; also it is obvious that it is better to use -20° slope or 20° horizontal shading device in these directions. On the other side the worst condition of N.W. faces especially with +60° slope is prompted.

By repeating the process presented above for fifteen other cities of Iran, the table below is resulted which contains the passive scores of different directions in each city. The quantity of these scores shows total kindness or unkindness of the sun to each of these directions through the year.

### Table 3. The passive scores of eight directions in Iran.

<table>
<thead>
<tr>
<th>City</th>
<th>N.</th>
<th>N.E.</th>
<th>W.</th>
<th>S.W.</th>
<th>S.</th>
<th>S.E.</th>
<th>E.</th>
<th>N.W.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Urmia</td>
<td>29</td>
<td>76</td>
<td>104</td>
<td>81</td>
<td>104</td>
<td>76</td>
<td>29</td>
<td>76</td>
</tr>
<tr>
<td>Tabriz</td>
<td>3</td>
<td>21</td>
<td>72</td>
<td>89</td>
<td>61</td>
<td>21</td>
<td>3</td>
<td>21</td>
</tr>
<tr>
<td>Zanjan</td>
<td>3</td>
<td>3</td>
<td>25</td>
<td>76</td>
<td>105</td>
<td>79</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Ramsar</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>16</td>
<td>17</td>
<td>10</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Bojnoord</td>
<td>0</td>
<td>-1</td>
<td>8</td>
<td>45</td>
<td>67</td>
<td>50</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>Mashhad</td>
<td>0</td>
<td>-5</td>
<td>-1</td>
<td>34</td>
<td>56</td>
<td>43</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>Kermanshah</td>
<td>0</td>
<td>-9</td>
<td>-5</td>
<td>28</td>
<td>59</td>
<td>51</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>Hamedan</td>
<td>2</td>
<td>0</td>
<td>13</td>
<td>58</td>
<td>88</td>
<td>69</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>Tehran</td>
<td>-1</td>
<td>-11</td>
<td>-7</td>
<td>0</td>
<td>13</td>
<td>18</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>Tabbs</td>
<td>-12</td>
<td>-45</td>
<td>-60</td>
<td>-38</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>-7</td>
</tr>
<tr>
<td>Birjand</td>
<td>-1</td>
<td>-18</td>
<td>-16</td>
<td>-1</td>
<td>19</td>
<td>49</td>
<td>39</td>
<td>9</td>
</tr>
<tr>
<td>Espahan</td>
<td>-1</td>
<td>-14</td>
<td>-9</td>
<td>0</td>
<td>25</td>
<td>45</td>
<td>33</td>
<td>6</td>
</tr>
<tr>
<td>Yazd</td>
<td>-6</td>
<td>-38</td>
<td>-40</td>
<td>-13</td>
<td>8</td>
<td>24</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>Kerman</td>
<td>0</td>
<td>-16</td>
<td>-14</td>
<td>-1</td>
<td>21</td>
<td>54</td>
<td>48</td>
<td>15</td>
</tr>
<tr>
<td>Shiraz</td>
<td>-3</td>
<td>-23</td>
<td>-22</td>
<td>-5</td>
<td>11</td>
<td>32</td>
<td>25</td>
<td>4</td>
</tr>
<tr>
<td>Bushehr</td>
<td>-15</td>
<td>-42</td>
<td>-64</td>
<td>-55</td>
<td>-17</td>
<td>-11</td>
<td>-13</td>
<td>-18</td>
</tr>
</tbody>
</table>

#### 2.4 Complex analysis

However in diagrams and tables such as above you can see the effect of different directions which could be used in basic architectural studies and design; for progressive design a computer program is needed to analyze the situation of every point of building skins in according to shading or reflecting effects on each other in the whole cycle of the year. Also it could be used to show the effect of any kind of shading devices such as horizontal or vertical louvers, overhangs and etc. As the colorful output of this program would be presented in one or more parallel/perspective views; it would used easily by the architects, urban planners and landscape architects to understand what the situation is and if the decisions taken are correct or not.
3 Conclusions

One of the greatest challenges of architects is to design buildings not only receive more from the kind face of the sun but also protect themselves from the sun’s unkindness. We wish “A new approach for solar analysis of buildings” would be answered to the important questions in solar architecture like the effect of the form and orientation of buildings, the proper amount of openness in each direction and the effect of shading or reflecting surroundings. Although it may be considered that this paper analyzed only the static state of buildings, but the method could be applied to the flexible and dynamic buildings of course. It is necessary to mention that several computer programs which were designed to make this happen for sixteen cities of Iran, could be used for solar analysis in other cities of the world.

4 References


SESSION

WEB APPLICATIONS

Chair(s)

TBA
ABSTRACT
System development methods and frameworks tend to provide a thorough and optimum solution to a given real-life problem. Classical models such as SDLC and its phases provide a common roadmap to tackle complicated problems in our world. But, such methods do not take into considerations the complex hidden angles of real-life and user’s demands. In other words the most emphasis is on the predefined and well-defined “order”’s and “conflict”’s are not addressed systematically. Particularly, in web application development, using only certain objective criteria is not sufficient in providing a thorough list of the requirements of the application. In other words, “subjective” factors needs to be addressed simultaneously in conjunction with objective ones and within the framework of order/conflict realization. This work addresses web application development from a different perspective. The “Do”’s and “Do not”’s are realized together with subjective-objective factors in a unified framework for practical and easy-to-do web application development.

Keywords: web application, framework, SDLC

1. INTRODUCTION
System development methods and frameworks tend to provide a thorough and optimum solution to a given real-life problem. Classical models such as SDLC and its phases provide a common roadmap to tackle complicated problems in our world. SDLC and its other alternatives such as Rapid Application Development (RAD) and Joint Application Development (JAD) have different perspectives toward the description of user requirements, though they all suffer from a common flaw that is, they do not take into considerations the complex hidden angles of real-life and user’s demands.

Web as a ubiquitous communication medium could handle sophisticated applications. General guidelines for any web application development which takes into the account the special characteristics of each application are required. For example, e-commerce sites have to support a special form of human-computer interaction (HCI) which could be very different from those of an online library. Another example is that, access to information on web has to be convenient for non technical users and general public Therefore; a general model should be able to formulate such characteristics. The nature of web applications is far beyond static interfaces. Information managed by applications changes rapidly, may be stored in many places and may be in variety of formats. Hence, web applications should be designed for change regarding content, requirements and architecture [5].

Considering web and its applications as systems, we found out that various web application frameworks were originated from that of system development. So the pitfalls of classical system developments frameworks are automatically reflected in web application frameworks. Most significant studied works include AutoWeb and Process model that will be discussed later in the paper.

To fix these problems, various frameworks and guidelines for system development were studied. We observed that in those frameworks the most emphasis was on the predefined and well-defined “order”’s and “conflict”’s are not addressed systematically.

Another real-life perspective absent in most frameworks was subjectivity and its elements. In fact, using only certain objective criteria is not sufficient in providing a thorough list of the requirements of the applications. In essence, “subjective” factors need to be addressed simultaneously in conjunction with objective ones and within the framework of order/conflict realization. This is particularly crucial for web application development that is the topic of this paper.

Fortunately, Hirschheim et al. have identified two dimensions regarding information system development including Subjectivist-Objectivist dimension and Order-Conflict dimension [1]. The concepts of orders and conflicts are discussed and tailored for web application development later in this paper. Inherently, a radically forgotten truth is that, every system, not only should be developed in a way that fulfills what it has to perform, i.e. requirements, but also, it has to avoid tasks lead to inappropriate results of the system. In addition, different elements of a system can be easily formulated and evaluated objectively if the associated objective criteria are available. Formulation of subjective elements though is the challenging part, since subjectivity inherently lacks firm certain criteria to rely on. Thus, the necessity of integration of objective and subjective factors in a system and its influence on more efficient web application development along with the methods and attitudes toward subjectiveness like expert systems are addressed as well.

Based on the insight gained from this and some other researches, some related works in the context of web was studied. Three works of Zahedi et al [3, 4, 9 and 10], regarding subjective and socio-cultural factors influencing web sites were among most inspiring ones. These works, raises the idea of categorizing subjective viewpoint into cultural (e.g. individualism vs. collectivism) and individual factors (e.g. demographics) and this can apparently illustrate the subjective angles of web playing an important role in the success of web applications.

This paper consists of four main sections. Section 2 will have an overview first on the comparison of popular system development frameworks such as SDLC, RAD and JAD and common features of all these classic methods are found out. Then it will take into consideration some of the web application frameworks have been proposed so far. This will include AutoWeb by Atzeni et.al [5], which proposes a methodology and a development environment for data-intensive websites and a process-based model that uses the generic product line lifecycle[6,7]. In section 3, requirements and design are found to be two important issues need reconsideration based on the results achieved through previous sections. Both order-conflict and subjective-objective dimensions are explained and exemplified. Also, for design phase, some comparisons are made between structured, object oriented and
agent oriented methods. Then finding the best design solution is discussed regarding associated paradigms, approaches and methodologies according to [2]. Section 4 will conclude over the whole work.

2. Overview

2.1 System development frameworks:

SDLC: developed in 1960's, System Development Life Cycle (SDLC) is a systematic approach to problem solving. SDLC classically is made up of several "waterfall" phases as: requirement definition, requirement analysis, design, implementation and maintenance. The SDLC is inherently linear, with each phase having specific outcomes and deliverables that forward important information to successive steps (Hoffet et al., 1999)[8]. Signing off on one phase to start another one is a must in SDLC and the deliverables produced from each phase include substantial amounts of documentation and specification. Therefore, once one phase is finished, it is literally impossible to return to earlier stages for making modifications due to probable changes to conditions and requirements. This prevents SDLC from adjusting to dynamic settings.

RAD: As inferred from the previous part, a main pitfall of the SDLC is that the user should wait until the end of the project in order to present a feedback. Adopting a collaborative approach, Rapid Application Development (RAD) was introduced in 1991 by James Martin. "The fundamental principal of RAD is to remove fixed milestones from the project by delaying the production of detailed system design documentation and deliverables until user requirements are fully clear. This is done using prototyping"[8]. Szekely identifies prototyping as "a small scale version of a complex system in order to acquire critical knowledge required to build a system" [8]. Continuous improvement of the prototype by the user and the developer results in a more tangible system development project that can respond to user requirements more quickly and easily. In other words, armed with prototyping, RAD can better cope with real-life issues and hence user requirements are satisfied to a higher degree than within a SDLC framework.

On the other hand, several weaknesses of the prototyping approach can be identified, such as inconsistencies between system models, non-compliance with standards and the lack of reusability of system components. Also, for some phases, signing off is difficult to be performed in a timely manner due to the cyclic, iterative process which affects resource planning. User-centric nature of prototyping may also increase the project time span and there is a risk in RAD that developers are too confined to satisfy documentary requirements.[8]

JAD: another alternative to traditional SDLC is Joint Application Development (JAD) which brings users into the development process as active participants (wikipedia). It uses data models to provide requirements definition; therefore, JAD allows key users to take part in the requirements modeling process. Such data modeling can produce thorough system specifications more quickly than SDLC, especially by the use of computer-aided software engineering (CASE) tools [8]. So, JAD can result in a more accurate statement of system requirements, a better understanding of common goals, and a stronger commitment to the success of the new system. However, JAD may seem more expensive and can be problematic if the user group is too large relative to the size of the project. (Wikipedia)

2.2 Web application development frameworks

Web can be considered as a system. It receives some data or information as its input and send out the output after processing that. Therefore, it makes sense to shape the web applications development frameworks according to that of system development.

Considering mentioned system development frameworks, we may conclude that they all attempt to capture user requirements or system functionalities as early and accurate as possible. For example, The ability to treat user requirements as a fluid, dynamic entity means that RAD development is well suited to Web application design, where user needs and commercial environment are often considerably more volatile than met in conventional IS design [8]. However, they are not typically used to capture Human Computer Interaction (HCI) factors that affect user interactions [13]. Particularly, far more importance is attached to this flaw, In the case of web applications, that interaction between system and users is of major vitality. Returning to the same example, a central aim of RAD methodology is to improve the communication between the development team and the user, ensuring that the final product meets user requirements more closely (Martin, 1991). In e-commerce design, however, the end user is not the client, but a distant customer connected via the Internet. A central weakness of requirements analysis via prototyping is therefore that the client must often put themselves in the shoes of the end user. User centered design is a specialist area and the client does not usually have the requisite knowledge to make an effective decision, but will instead act to define requirements tuned to their own subjective, needs and opinions [8]. Some of the studied works are discussed below:

a) AutoWeb: A model-driven development of web applications was presented in [5] and the AutoWeb system was described and examined by real-life experiences. In AutoWeb model, the initial step is the collection of requirements and their formalization. This phase is human-intensive it is supported by a tool of the Autoweb System called Visual HDM. The second phase is the generation of the supporting database that results in a relational database, which will support the application at runtime. This phase is totally automated by a tool called VHDM Database Schema Generator. The last phase is the implementation and deployment of the Web application. There are two scenarios for this phase:

1) the application content does not exist, so the empty database produced by Autoweb is filled with structured application content, possibly linked to unstructured data (e.g., multimedia files). This operation is almost totally automated and the application database can be filled via a Web interface automatically produced by the Autoweb Data Entry Generator.

2) The application content (or part of it) already exists and is stored in a legacy system (e.g., a relational database): in this case, the database schema produced by Autoweb must be mapped onto the legacy data sources, in order to integrate the existing content into the page production process.

b) Process model: Process-based model for web application development was another work proposed in [6] with the purpose
of generating ready-to-run web applications for business users. This model has got the idea from generic product line engineering life cycle using [7]. It splits up the software engineering process into two phases called the domain engineering and the application engineering parts. During the domain engineering part, the common domain for all software products enclosed with the product line is analyzed, designed and implemented. This comprises the identification of requirements, the elaboration of design artifacts to form a foundation common to the whole product line and finally the implementation of reusable software components that can be composed to form a complete software product. The application engineering phase consist of application analysis, application design and application implementation.

The fact is that, the frameworks we have studied, more or less were inspired by classic system development methods. Even if those methods were not directly addressed, the concept could be easily observed.

3. Two issues that need reconsideration
We have come up with the idea so far that classical methods as a foundation for various web application development frameworks provide a straightforward roadmap to solve problems. But, such methods lack the considerations of the complex hidden angles of real-life and user's demands. Reviewing the mentioned frameworks, we can address two general issues subjected to reconsideration in this paper: 1) requirements, 2) design. Each has some sub categories stated as following:

3.1 Requirement
Apparently, requirements provide the system developer with a base, essential for development job to proceed. For instance, phases 3 to 5 of SDLC are crucially depended on requirements definition phase (phase 1) as well as requirement analysis phase (phase 2). System developers base the development on the requirement more or less corresponding to those invoked by stakeholders. For instance, this fool-proved fact that a health application is developed differently from flight application exemplifies the fundamental influence of specific requirements demanded by prospective users of each application on the output of the system or the ultimate system-to-be. However, we hold the opinion that "requirements" should be looked at in an innovative manner. We divided this new approach into two dimensions as following:

3.1.1 Order-conflict dimension
"Requirement investigation" is the traditional view for elicitation of system's characteristics. Inherently, a radically forgotten truth is that, every system, not only should be developed in a way that fulfills what it has to perform, i.e. requirements, but also, it has to avoid tasks leading to inappropriate results of the system. An outstanding example for that is the Internet. During the Internet evolution, merely its desirable functionalities were taken into consideration, while within that era, no one wondered "what if an intentional modification made by an unauthorized user, i.e. a hacker?" or "what if some intruders misuse the Internet environment and more other "what ifs" like that. Consequently, Internet corruptions were encountered and experts were invited to come up with some solutions.

This argument encourages the idea that "Do s" of a system have to receive the same attention as "Don't s" do. The former are called "conflicts" and the latter are called "orders". Conflict analysis mostly deals with security, safety and privacy issues; however other problems may be addressed as conflicts according to developer's experience or predictions of possible risks. Table 1, shows that how traditional view is replaced by our proposed innovative view for first two phases of classic SDLC regarding this dimension.

<table>
<thead>
<tr>
<th>Table 1 – Traditional vs. Innovative view to development</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Traditional view</strong></td>
</tr>
<tr>
<td>1. Requirement definition</td>
</tr>
<tr>
<td>2. Requirement analysis</td>
</tr>
<tr>
<td>4. Conflict analysis</td>
</tr>
</tbody>
</table>

According to the scope of this paper, orders and conflicts of web applications should be highlighted. To realize that, we shed light on ecommerce websites as an important web application. Orders can be regarded as general and specific ones. General orders are mainly whatever is known as "requirements" in many literatures. Good presentation, fast download, flexibility, security, accessibility, usability and proper navigation, etc are some general orders.

On the other hand, conflicts of an ecommerce website can be identified as:

- The website must not allow accidental or intentional access to and manipulation of information by unauthorized users.
- The website must not broadcast or sell personal and contact information of its users without their awareness.
- The website must not aggregate certain culture, race or religion.
- The website must not collapse due to a failure in one or more of its functions.
- The website must not be very complex and confusing for the target audience.
- And many other conflicts that should be investigated according to the application.

As it is clear from the list of conflicts, it can include issues ranging from socio-cultural to ethical to technical ones.

3.1.2 Objective-subjective dimension
Both orders and conflicts mentioned above, in turn, are obtained regarding subjective-objective dimension of information systems.
Objectivity is the property of scientific evaluation that can be performed independent from the individual scientist (the subject) who proposes them. Scientists strive, where possible, to eliminate human senses by use of standardized criteria. However, some aspects of every system like web may not correspond to such an objective view. In other words, their evaluation is idiosyncratic- structural or behavioral characteristic peculiar to an individual or group. Hence, subjectivity issue remarkably matters in information system development in general and web development in specific.

In fact, objectiveness is the main behavior of a system, whilst subjectiveness may be regarded as the initial condition assigned to a system that leads to various outputs when different people (subjects) work on that system. As a result, one may think of dividing each of four items of above table into two subsets based on objective vs. subjective view, for instance, divide order definition into order-objective definition and order-subjective definition and do the same for other items. Such categorization may seem reasonable at the first glance. Nevertheless, the fact is that these two dimensions, i.e. order-conflict and subjective-objective, work together rather than individually. Thus, if the definition and analysis of an order/conflict is desired, both subjective and objective sides need to be taken into account. Different elements of a system can be easily formulated and evaluated objectively if the associated objective criteria are available. Formulation of subjective elements though is the challenging part, since subjectivity inherently lacks firm certain criteria to rely on. In this area, Zahedi et al. ran a large multistage empirical study of user satisfaction and effectiveness of various web design based on cultural and individual factors [3, 4]. Before that, few studies had mainly discussed and investigated the existence of such socio-cultural dimensions in information system including Web sites. These works, raises the idea of categorizing subjective viewpoint into cultural and individual factors as following:

Cultural factors may include:
- power distance
- individualism vs. collectivism
- anxiety avoidance
- long-term vs. short term orientation
- polychronic vs. monochronic time orientation
- masculinity vs. femininity

Individual factors include:
- demographics
- professional knowledge
- information technology knowledge
- flexibility
- information processing abilities
- cultural knowledge
- Hermeneutics (this may be described as the development and study of theories of the interpretation and understanding of texts.)

A better understanding of subjective factors and their influence on web sites is provided in [4] through discussing how online shoppers are influenced by the exposure to the web design elements-the features, components and information used in developing ecommerce websites. In this study, Song and Zahedi addressed the prediction of e-shoppers' reaction to the website according to web design elements by answering this question: “What is the process by which websites may impact web customers' beliefs and behaviors?” A conceptual framework for explaining such a process is called Belief Reinforcement Model or BRM. To test such a model, some categories of web design elements have been figured out as promotion, service, external interpersonal, ease of use and navigation and purchase intention. The results indicated that the categories of web design elements influence web customers' beliefs, which in turn impact purchase intention by changing their attitudes. This shows that web design elements play an important role in shaping web customers' attitudes and purchase behavior.

On the other hand, we expanded our studies to other papers discussing the similar issue by searching keywords such as web design elements, web design evaluation criteria, ecommerce websites and the like. As a result, we have come up with bunch of elements that could play a part in an ecommerce website. Some of those elements were of the same meaning but represented differently in various papers. Some categories of elements could include another category too and many other meaningful relationships could be found among those results. Therefore, we tried to reorganize and re-categorize the obtained elements that are illustrated in Table 2.

Referring again to the list of orders and conflicts, we can verify both objective and subjective aspects of each one. For example, loading time should be reasonably fast to satisfy users. One may link loading time merely to technical issues such as use of graphics in the design of the website or connection speed of the end user, etc. However, quickness also can be interpreted differently by users with different response time. In other words, human characteristics are crucial here. This example states that how objective aspects of a web application is woven into its objectiveness.

Now, the question is that how subjective concepts can be expressed for a system development? In this case, how the elements of ecommerce websites can be mentioned in terms of subjectiveness. There are many different factors forming such concepts. The problem becomes even more complicated when we consider the existing mutual effects of those factors. For instance, the more we try to provide methods to increase trustworthiness in a website, the more delay occurs that can result in customer dissatisfaction. Hence, a compromise between trustworthiness and response time has to be maintained to balance the effects of each one properly. Expert system, as explained in the following part may help developers in this field.

**Expert System and subjectivity**

An expert system, also known as a knowledge base, is a system that contains the knowledge and analytical skills of one or more human experts, related to a specific subject (wikipedia). Refocusing on this definition, we could figure out the inherent subjectivity issue beneath the concept of an expert system. In other words, the subjectivity, these days is equivalent of an expert system and most of modern information systems include knowledge base. Therefore, a regenerative knowledge-base can be added to the system that follows certain characteristics. Suppose that we want to design a very fast website. We can define the speed in terms of the response time of the user or the web server or both. This can be put in a knowledge base. In fact, ultimate
design needs a proper knowledge base to hold the subjective paradigms.
Nevertheless, developing such a system is beyond the scope of this paper.

Table 2: Elements of an E-commerce Website

<table>
<thead>
<tr>
<th>Name</th>
<th>Subcategories</th>
</tr>
</thead>
<tbody>
<tr>
<td>Presentation</td>
<td>First impression</td>
</tr>
<tr>
<td></td>
<td>Layout consistency</td>
</tr>
<tr>
<td></td>
<td>Text color/format</td>
</tr>
<tr>
<td></td>
<td>Background color/pattern</td>
</tr>
<tr>
<td></td>
<td>White space</td>
</tr>
<tr>
<td></td>
<td>Graphical/tabular presentation</td>
</tr>
<tr>
<td></td>
<td>Detailed description of product/service images</td>
</tr>
<tr>
<td></td>
<td>Payment option</td>
</tr>
<tr>
<td></td>
<td>Shipping option</td>
</tr>
<tr>
<td></td>
<td>FAQ</td>
</tr>
<tr>
<td></td>
<td>Help/instructions</td>
</tr>
<tr>
<td></td>
<td>Shopping cart</td>
</tr>
<tr>
<td></td>
<td>Terminology consistency</td>
</tr>
<tr>
<td></td>
<td>Vocabulary appropriateness</td>
</tr>
<tr>
<td></td>
<td>Range of multilingual options</td>
</tr>
<tr>
<td>Accessibility</td>
<td>domain name</td>
</tr>
<tr>
<td></td>
<td>download time</td>
</tr>
<tr>
<td></td>
<td>disabilities</td>
</tr>
<tr>
<td></td>
<td>Structure type</td>
</tr>
<tr>
<td></td>
<td>Links</td>
</tr>
<tr>
<td></td>
<td>Navigation bar</td>
</tr>
<tr>
<td></td>
<td>Search engine</td>
</tr>
<tr>
<td></td>
<td>Site index</td>
</tr>
<tr>
<td>Privacy and Security</td>
<td>Privacy policy</td>
</tr>
<tr>
<td></td>
<td>Security mechanisms and warnings</td>
</tr>
<tr>
<td></td>
<td>Security certificates</td>
</tr>
<tr>
<td></td>
<td>advertisements</td>
</tr>
<tr>
<td>Marketing factors</td>
<td>Customization/personalization</td>
</tr>
<tr>
<td></td>
<td>After sales relationship building</td>
</tr>
<tr>
<td></td>
<td>Quick reply to email enquiries</td>
</tr>
<tr>
<td>Usability</td>
<td>Ease of learning</td>
</tr>
<tr>
<td></td>
<td>Ease of use</td>
</tr>
</tbody>
</table>

3.2 Design

3.2.1 Popular design methods
There are many different methods model and design system requirements. Structured analysis and design is a classic, straightforward method. Mainly concentrated on processes, It is pretty easy to deal with, but, it seems to be suitable for simpler systems with less complex processes. So, it is not recommended for developing complicated systems like web applications.

Object-oriented method, on the other hand is focusing on the objects rather than processes. Therefore it would be more appropriate for web applications. But it fails to deal with issues such as autonomy, proactiveness and sociality [11].

Agent/goal oriented seems to be an alternative to structured and object orientation for various reasons. It can provide robust and autonomous systems and specifically web applications, capable of serving end users with a minimum of overhead and interference and coping with unpredicted situations [11, 12].

3.2.2 Paradigms, approaches and methodologies:
In fact, designing a system that accomplishes our goals is a cumbersome task. A general belief states that the higher quality of information systems is achieved through a more disciplined development process. So, design receives a noticeable attention and it seems that information system development methodologies (ISDMs) have the main role toward this target [2]. To start a discussion over this topic, four terms should be clarified in advance. These terms are defined from more specific to more general ones.
Methodology: codified set of goal-oriented "procedures" which are intended to guide the work of the cooperation of the various parties (stakeholders) involved in the building of an information system application. Typically, these procedures are supported by a set of preferred techniques and tools and guiding principles [2].

Approach: class of specific ISDMs that share a number of common features. four features proposed in [2] are Goals, Guiding principles and beliefs, Fundamental concepts and Principles. Based on the definition of Information System Development Approaches (ISDAs), it makes sense to compare different ISDAs regarding their number of instances [2].

Paradigm: paradigm consists of assumptions about knowledge and how to require that (epistemological assumptions), and about the physical and social world (ontological assumptions)[1]. Two types of assumptions yield two dimensions discussed earlier in this paper:

1. Subjectivist-Objectivist dimension
2. Order-Conflict dimension.

With combination of these two dimensions, four paradigms are identified:

1. functionalism (objective-order)
2. social relativism (subjective-order)
3. radical structuralism (objective-conflict)
4. neohumanism (subjective-conflict)

Theses inter-related terms shape a hierarchy with paradigm in the highest and methodologies in the lowest position. Approaches also link methodologies to these paradigms and the underlying assumptions. Ivari, Hirschheim and Klein have proposed a framework in such a hierarchical model in more details and hold the idea that one should not strongly stick to some specific ISDMs. Instead, more fundamental and general Information system development approaches (ISDAs) need to be focused on. "Those ISDAs can be assumed as prototypical classes that share number of features with their member (instance) ISDMs" [2]. It is also perceived that in practice, ISDAs are influenced by assumptions from more than one paradigm, but one paradigm is dominant. This mixture can lead to more creative solutions to system development [1, 2]. As a result, to use this framework one should be aware that although, most researches are focused on the first paradigm consisting of objective and order issues, i.e. functionalism, particularly in web application development as a multi-dimensional system, it is observed that two or more paradigms are working together. Based on the aspect through which we are looking at the web application, it reflects a different paradigm. Because of that, more than one approach will be take part in the development process. As an example, the concept underlying functionalism paradigm is close to the definition of straightforward requirements and objective modeling and assessing of a system. These in turn remind us of structured approach and methodologies. To put it simply, developer may be encouraged to exploit structured techniques in this example since the dominant paradigm is recognized as functionalism. However, we face the subjectivity and evolutionary nature of requirements that inevitably affect the output of system. So, functionalism is not sufficient. Other paradigms need to take part and hence other approaches and methodologies will be involved. In other words, finding the most appropriate methodology according to multiple paradigms and methodologies requires far more sophisticated model.

4. Conclusion
All in all, this work addresses web application development from a different perspective. Requirements and design issues in system development and particularly web application development are reconsidered here. For that, requirements are transformed into the “Do’s” and “Do not’s” realized together as orders and conflicts with subjective-objective factors. Also, design solution is decided based on the involved paradigms, approaches and finally methodologies. All of the mentioned issues eventually should be integrated into a unified framework for practical and easy-to-do web application development.

5. REFERENCES
Using agent to coordinate web services

C. H. Liu¹, Y. F. Lin², and J. Y. Chen³
Department of Computer Science & Information Engineering
Nation Central University
Jhong-Li, Taiwan 32099

Abstract -Traditionally, agent and web service are two separate research areas. We figure that, through agent communication, agent is suitable to coordinate web services. However, there exist agent communication problems due to the lack of uniform, cross-platform vocabulary. Fortunately, ontology defines a vocabulary. We thus present the web ontology language (OWL)-based operational ontology that defines a declarative description, which can be accessed by various engines to facilitate agent communication. And, we define mental attitude of agent to be shared among agents. Our architecture enhanced the 3APL agent platform, and it is implemented as an agent communication framework. Finally, we extended the framework to be compatible with the web ontology language for service (OWL-S), and then develop a movie recommendation system with four OWL-S semantic web services on the framework. The benefits of this work are: 1) dynamic web service coordination, 2) ontological reasoning through uniform representation, namely, the declarative description, and 3) easy reuse and extension of both ontology and engine through extending ontology.

Keywords: agent communication, semantic web service, agent mentality layer.

1 Introduction
Traditionally, agent and web service are two separate research areas. The research on agent focuses on problem solving mechanisms in distributed environment. A widely known agent model is the belief, desire and intention (BDI) model. On the other hand, the research on web service concentrates on distributed technique and standard such as web service description language (WSDL), simple object access protocol (SOAP) and universal definition and discovery integrated (UDDI). However, web services are not reliable and easy-to-use due to the fact that they are at remote sites that a user has no control over them.

Incidentally, an agent is able to proactive request other agents to achieve a goal, and then the requested agents respond reactively through agent communication. Each agent has its capability (actions). Also, each has its mental attitude (such as what it believes). Through communication mechanisms, the agent decides the proper agents to execute their actions, which are implemented as semantic web services that in turn are annotated web services in this paper. Thus, the web services can be coordinated through the agent communication.

A problem with agent communication is that there is no cross-platform, uniform vocabulary to identify concepts used in various agent programs. This causes difficulty in communication. Fortunately, ontology is a document that formally defines a vocabulary. The web ontology language (OWL) is a popular language to describe ontology, and it allows people to utilize the uniform referential identifier (URI) to give every concept a specific term such as those used in agent communication. It also defines semantics of the terms, and organizes all kinds of terms by using relations among the terms. The terms can be shared among agents. A special ontology is called operational ontology, in which the operational concepts are represented in a declarative description. By using OWL to describe the operational ontology [1], this paper solves the problem in a novel way.

This paper presents an agent communication framework with operational ontology, as well as the mental attitude of agent, to facilitate agent communication. This paper is organized as follows. Firstly, we depict the agent communication layers stack and the ontology associated with each layer in section 2. Secondly, we explain the agent communication architecture that consists of the declarative descriptions for ontology and the engines to access them. Next, the architecture is implemented to a framework by enhancing the 3APL agent platform in section 3. Then, the framework is extended to be compatible with the web ontology language for service (OWL-S). Further, we develop a movie recommendation system with four OWL-S services on our framework in section 4. Lastly we give the conclusion in section 5.

2 Agent Communication and Its Ontology
Ontologies are vocabularies that should be obeyed when agents are communicating with each other. For each agent communication layer in Figure 1, a corresponding operational ontology is defined in OWL, which is too detailed to show in this paper.
In Figure 1, the physical layers deal with actual message and message format. Note that the message transport layer is not covered in the paper. In the content language layer, the content language ontology defines the message representation. On the other hand, the abstract layers deal with high-level communication. In the agent mentality layer, the proposition ontology is used to define the representation of mental attitudes of agents, which is the abstract concept of agents and is represented by property “belief”. Also, in this layer, action and capability of agents are defined by action ontology. In the performative layer, communicative act ontology defines formal semantics of every communicative act. As for the highest interaction protocol layer, the interaction protocol ontology defines the state shift during communication. It constitutes all performatives and thus forms a conversation pattern, which then becomes a restricted standard of communication amid agents. Briefly summing up, the five operational ontologies are collectively called the agent communication ontology. And, the documents of these ontologies are referred to as the agent communication description.

The relationship of the documents is shown in Figure 2. Every description in Figure 3 refers to one another via a URI reference. Every interaction protocol description refers to multiple communicative act descriptions. Similarly, every communicative act refers to an action description as well as a proposition description in the content language description.

3 Agent Communication Architecture

As shown in Fig. 3, our architecture contains the Mental Model, Action Engine, Proposition Engine, Communicative Act (CA) Engine, and Protocol Engine. Additionally, the architecture also contains 3APL agent platform as the planner, and Data Model as the ontology storage.

Next we will dilate each component in the architecture.

1) Protocol Engine: The protocol engine processes interaction protocol description. It identifies the agent who participates in communication, records the communication state, and then invokes CA Engine to transform the state.

2) CA Engine: The CA engine processes communicative act description. It evaluates the feasibility precondition, updates the mental model by rational effect. If action exists in the description, the action engine will be invoked.

3) Action Engine: The action engine processes the action description. It decides the way to execute actions such as sequence, concurrency, alternative and iteration. Moreover, the actions’ precondition will also be evaluated before execution, and then the actions’ effect will be updated in the data model after execution.

4) Proposition Engine: The proposition engine processes the proposition description. It refines a statement in data model as a proposition in mental model. It allows a statement to have different manifestations if needed.

5) 3APL Agent Platform: A 3APL is a Prolog-based agent platform. It contains a goal base, a belief base, and a plan base that stores goals, beliefs, and plans, respectively. Besides, it contains plan rules to infer plans and an mProlog-based engine [2] to deliberate these rules. Our work tries to extend this platform with subscribing the proposition in the mental model as data source of belief base.

6) Mental Model: The mental model stores the reified statements i.e. propositions. It provides a belief
algorithm [3] to appropriately allocate the proposition as an actual world or imaginary worlds.

7) Data Model: The data model [4] stores domain ontologies and statements which are generated form these ontologies. The ontologies are regarded as data types of actions’ input/output and generate statements as their values.

3.1 Executing Agent Communication Description

Our agent communication is based on the Foundation of Intelligent Physical Agents (FIPA) formal semantics, which is questioned by some researches [5][6] who point out that an agent is unable to predict mental attitudes of other agents by checking its own attitudes. Thus, if we regard these attitudes as precondition for communication, they should be checked. Further, if we consider them as effect of communication, they probably cannot be determined by the sender agent, because the effect is administered by the recipient agent. Therefore, we would say an agent itself cannot predict an intended purpose.

In our architecture (see Fig.4), the mental model is shared among agents. And, the data model saves an action’s effect and output executed by an agent, which is sharable too. Thus, there is no above mentioned problem.

Figure 4: Agent Communication Framework

The execution process of the engines and the declarative descriptions is shown in six steps below:

Step 1: The protocol engine will read the agent communication description. It will choose and read the section of the interaction protocol description and then confirm participative agents by using the initiator and the participant. Afterwards, it will put decomposition rules (plan rules) [7] owned respectively by agents into plan bases of the 3APL agent platform.

Step 2: The protocol engine then enters a certain stage and orders the CA engine to perform the transition. Then it considers the internal communicative act as a parameter, which will be transmitted into the CA engine with the initiator and participant.

Step 3: By viewing the agent mental states in the mental model, the CA engine can evaluate the feasibility precondition (FP) of this CA. If there is no state left, this algorithm terminates. In contrary, it is necessary to meet the demand of the FP and insert all parameters, including the initiator, the participant (come from the protocol description, and they are necessary), the action and the proposition (come from the content language description, and they are not necessary) into it. Then, it will call respectively the action engine and proposition engine to execute processes depicted below. If not, the interaction will come to an end.

Step 4: By viewing the domain knowledge in the data model, the action engine can evaluate the precondition of the action. If it forms, this action will be executed, and the effect of the execution will be recorded in the data model. At the same time, if there has an output which will be transmitted to the propositional engine and be reified to mental model. By contraries, if it fails to form, the execution of the action will be called off. And the explanation of this cancellation, such as the precondition and the input bringing about the failure, will be transmitted to the propositional engine and be reified mental model.

Step 5: When the execution of the action is finished, the CA engine can store outputs and reasons that are reified from the propositional engine. Those outputs and reasons are very useful when we are ought to judge the CA next time.

Step 6: Go back to Step 2 and enter the next stage.

3.2 Selecting Agent Communication Description

In section 3.1, we described how agents communicate through the agent communication description. Here, we are going to elaborate how agents select appropriate agent communication description before communication. The selection process and the interaction of engines will be discussed next.

If the planner infers plans to be “external actions” (not agent capabilities by itself), the planner knows the agent itself cannot complete a certain task, it will seek help from external agents to move closer to the goal. The agent can coordinate or cooperate with external agents for that. The difference between coordination and cooperation is that if descriptions are protocols [8][9] of contract net family, the agent uses coordination; inversely, if they are protocols [10][11] of request family, the agent uses cooperation. Certainly, there are also protocols [12][13] of broker family. However, that involves interaction of all agents, and it is not discussed here.
The process of selection can be divided into three steps: 1) external action selection and 2) performative selection and 3) interaction protocol selection.

Step 1: An external action selection. Agents would query the action registry about the location of the agents with the capability of executing this external action. If they find out an agent, they will send the action description into the action engine; if not, they will terminate.

Action engine evaluates effects in action description with data model as the information source to supply necessary parameters such as output or other local variables. If evaluations succeed, proposition engine will reify and transmit the effects to mental model where they can be preserved. On the contrary, if evaluations fail, it will terminate. Note if this action description was ever executed by agents offering actions, output will remain to exist in data model. Thus, we have to make agents be able to share data model and mental model with each other.

Step 2: A performative selection. The CA engine use external actions and output of mental model as parameters to evaluate rational effect of a number of the FIPA 22 CAs descriptions [14]. If evaluations are successful, it will choose and send this CA description to interaction protocol engine; if not, it will go back to the 1.

Step 3: An interaction protocol selection. After the protocol engine receives CA descriptions, with which it will choose the interaction protocol description. Later, the chosen protocol description will replace the original external action. Then, it will be finished.

Our action communication description is the description which linked the action description with the CA description as well as the interaction protocol description.

3.3 Agent Communication Enhanced 3APL Deliberation Cycle

By using the selection of agent communication description just mentioned, the 3APL deliberation cycle is enhanced, as shown in Figure 5.

Figure 5: Agent Communication Enhanced 3APL Deliberation Cycle

The agent chooses one goal in goal base, and judge whether it is a composed action or not. If yes, the agent will choose the plan rule to decompose and execute it (the condition of choice is that if the current state of brief case satisfies the guard of the plan rule). After the execution, the newly-produced plan will be stored in the plan base. At the same time, for the purpose of completing sub-goals decomposed by the plan rule, the agent will choose a new plan rule to carry out.

If this goal is a basic action, the agent will make a judgment if this action is external or not. If not, the agent will find the capabilities in the belief base. If they are found, this action will be carried out.

If the action is external, it means that the agent itself cannot carry it out. At this point, the agent will question the matchmaker: “Which agent is able to execute this external action?” Subsequently, the agent will decide on one interaction protocol by searching for proper communicative act and follow this protocol to coordinate or collaborate with another agent that will execute this external action.

3.4 Agent Communication Framework for Semantic Web Services

We extend the framework in Figure 4 to be compatible with the OWL-S, as shown in Figure 6. As mentioned earlier, the action ontology is defined in the agent mentality layer. In Figure 5, we extended the action ontology to generate the declarative description, which is represented by OWL-S Service description. Then, we can reuse OWL-S’s process engine [15] and OWLS/UDDI Matchmaker [16].

Figure 6: Agent Communication Framework for Semantic Web Services

On this extended architecture, we added: 1) Web service, 2) OWL/UDDI Matchmaker, and 3) OWL-Process Engine. We analyze the three components from the viewpoint of OWL-S: 1) they can be bound to a web service through OWL-S grounding, and they can be used by agents. 2) The
OWL/UDDI Matchmaker provides the platform for the OWL-S Profile to register and match. 3) The OWL-Process Engine allows agent to invoke web services according to the OWL-S Process.

And, our Action Engine is responsible for controlling these components. In reading action, it will store input into data model as parameters and use actions’ capabilities to inquire OWLS/UDDI Matchmaker, attempting to find proper OWL-S service description, which will be read and executed by the OWL process engine. Under three conditions: 1) before evaluation, 2) after updating precondition and effect and 3) after execution, the action engine will check the semantics of the conditions by using data model. The output of OWL-S will be stored into data model when 1) OWL-S process engine ends execution and 2) the action engine retrieves and sends the output back to the agent.

4 An Example

We develop a movie recommendation system on the extended framework. In this system, there are four semantic web services: 1) Information Retrieval Service, 2) Recommendation Service, 3) Video Abstract Service and 4) Video Broadcast Service. Agents utilize cooperation and coordinate to interact with the semantic web service. With the cooperation, agent S queries the OWL-S/UDDI:

“Which agent(s) offer the abstract video service for “Brokeback Mountain”?”

And it receives the response:

“Agent T offers the service”.

Then, agent S decides the CA to interact with agent T according to agent attitude. When it receives the request, it will judge FP of CA to response “agree” or “refuse”. If agreed, it will execute the abstract video service. The process is shown in Figure 7.

Figure 7: Cooperation with Semantic Web Service

When an agent queries OWL-S/UDDI and receives the response that involves multiple agents. The agent will use coordination to pick a proper agent. Assume agent S queries:

“Which agents provide video broadcast service to show the Brokeback Mountain?”

Then it receives the reply:

“The agent A, B, C, and D offer this service.”

Then, it will use Contract Net Protocol to interact with them. Among them, agent c and d have difference condition bandwidth greater than 1Mbps and bandwidth greater than 2Mbps, respectively. Next, the agent s accepts the proposal of agent d. Finally, agent d will execute the video broadcast service. The process of coordination is shown in Figure 8. And the execution result of the movie recommendation system is shown in Figure 9.

Figure 8: Movie Recommendation System

Figure 9: Movie Recommendation System

5 Conclusions

This paper defines the operational ontologies for agent communication program, along with the engines to interpret the ontology. This facilitates declarative development of agent programs. Its advantages are:
1) Dynamic Semantic Web Service Invocation: Operational ontology enables the calling agent communication to comprehend the meaning of the called. As a result, the calling agent communication can dynamically invoke the accurate semantic web service in accordance with semantics of operational ontology.

2) Ontological Reasoning with Uniform Representation: The operational ontology and domain ontologies stand for programs and data, respectively. They adopt the same representation format, namely declarative description. Finally, program and data on the semantic web can be inferred according to the declarative description.

3) Easily Reuse and Extendable through Ontology Extension: When a new ontology is needed, the ontology defined before can be extended by OWL. As for the new engines for the new ontology, we can also reuse the engines developed before.

6 References


Remote sensing and control by means of the parallel port, java and recycled e-waste


1Centro de Investigación en Ingeniería y Ciencias Aplicadas, Universidad Autónoma del Estado de Morelos, Cuernavaca, Morelos, México
jose_hernandez@uaem.mx

2Universidad de las Américas, A.C. Cd. de México, México, D.F.

3Instituto de Ingeniería y Tecnología, UACJ, Cd. Juárez, Chihuahua, México

Abstract - We discuss a prototype multiplatform system to perform remote sensing and control of physical variables i.e. temperature, based on the use of the parallel port, application software developed on java and gnu software, working on old recycled computers Pentium II (now considered as computer garbage or e-waste). In this paper, we analyze the electronic circuit that allows sensing until sixteen multiplexed different signals, and its introduction to computer by means of parallel port, we analyze the program on java that makes possible the execution of remote sensing and control over the Internet, as well as the system’s implementation on old computers.

Keywords: Remote sensing, Java programming, computer recycling, e-waste.

1 Introduction

Remote Sensing in the most generally accepted meaning refers to “instrument-based techniques employed in the acquisition and measurement of spatially organized (most commonly, geographically distributed) data/information on some property(ies) (spectral; spatial; physical) of an array of target points (pixels) within the sensed scene that correspond to features, objects, and materials, doing this by applying one or more recording devices not in physical, intimate contact with the item(s) under surveillance (thus at a finite distance from the observed target, in which the spatial arrangement is preserved); techniques involve amassing knowledge pertinent to the sensed scene (target) by utilizing electromagnetic radiation, force fields, or acoustic energy sensed by recording cameras, radiometers and scanners, lasers, radio frequency receivers, radar systems, sonar, thermal devices, sound detectors, seismographs, magnetometers, gravimeters, scintillometers, and other instruments” [9].

On this paper we will focus on remote sensing of temperature by means temperature sensors, gnu free software and old recycled computers (considered as e-waste). In [1] and [4] is discussed the way to deal with electronic waste (e-waste), derived from domestic and office appliances like refrigerators, micro ovens, printers and computers, now considered as a health risk by the way these appliances are finally disposed in third world countries like India, several at Africa, China and Latin America, where mostly of e-waste is burned or recycled in uncontrolled environments contaminating the air, soil and water, with high contents of mercury and acids; given the high toxicity of these pollutants is considered as an emerging problem. But we prefer to see this problem as in [5] like an opportunity, not merely by the valuable materials in them like iron, copper, aluminium, gold and other metals, but specifically in the case of computers, like reusable components in useful applications.

Lets suppose we require automating the lecture of variables involved in some process, a common task in several laboratories, by force brute is required the waste of valuable time of high skilled human resources that spent most of their time in “dynamics” on which is required to measure each specific xi interval of time several variables. Now consider extremely dangerous process difficult to measure like the resulting of acids manipulation or reactions producing high temperatures. These above described situations make us think about the quality of the lectures, the human error and the heavy and not value-added task that this activity can result for some researchers. Although there exists proprietary software (i.e. Labview) that allows the control of such process, the solution can result very expensive and for some research projects prohibited due high license and training costs.

If we analyze related work focused on the measure of single variables like temperature, we could find at the internet some interesting approaches that deal with this problem. In [3] is discussed the use of GP-3 board, circuitry and C# to measure and control temperatures over the internet, but the system requires relatively new computers, third party hardware and .NET to work. In [7] are used recycled old computers, a multi meter and Linux, to treat with the same problem, despite code is multiplatform and uses gnu software like PERL, reach is limited by the use of single purpose multi meter that read merely temperatures and display lectures in a tiny web browser over a LAN.
1.1 Problem at hands

Considering the above exposed situation, is our desire to contribute with the design and implementation of a low cost hardware and software application that allows monitoring and control of several digital signals (i.e. temperatures) by means of the parallel port -we remark that this port is available in most of old computers due it was used to send data to printers- and free gnu software, specifically Linux, Netbeans, Java and the necessary drivers to manage this port.

On the first section of this paper, we discuss the proposed Client-Server technology by means of block diagrams; the electronic circuit that will multiplex the signals coming from different temperature sensors, we analyze the electronic diagram, and discuss its basic performance. Then we analyze the class diagram to manage the parallel port, to read the information from sensors as well as the writing of control sequences, in this section, we present the Java code employed and created ad-hoc for the application, at the end on this section we describe specific details about implementation (hardware and software) on old computers and the tuning of the application to work over the internet. We present our preliminary results and finally our conclusions.

2 The Client-Server technology

Java is actually quite extensive including capability for http, ftp, and tcp. The part of the Client-Server architecture that we will be using in this paper takes advantage of tcp communication and consists of two pieces, a server piece that reads the temperature (until sixteen different sensors) at the location we are interested on (by means of a remote IP address), and a client piece that will remotely access and -if desire- to control the temperature.

Fig. 1. Block Diagram of the proposed Client-Server Technology

The client piece in our application is generated by means of a remote terminal that runs a remote .JAR program which uses a timer that requests a temperature reading every 1 second from different sensors located at Server side. Depending of Network configuration at Client Side, it uses a proxy to call the remote object on the Server side and the call is made as if the remote object were living on the same machine as the Client. The remote object in our example takes the form of an instance of a class. The server job is simply to register and run the content of the remote object (a .JAR program) and provide a channel (via SSH) to the remote object so that the Client knows about them. The Remote Object does all the important work by controlling and sensing the hardware through the circuit that we designed and implemented. To get a little more detailed idea of the architecture for the current project, take a look at the UML diagram in figure 4 showing the three individual assemblies needed to complete the remote picture. Note that the Client piece (on the left side) lives on the Client machine and the remote object and the server piece live on the Server machine.

2.1 Temperature sensing circuitry and the parallel port of the computer

Figure 2, shows the low cost implementation circuitry we designed, at left hand is located the sensors array, for simplicity purposes we use temperature sensors LM135 (-55ºC to +155 ºC temperature range), at the center, the multiplexing module consisting of a HEF4067B, sixteen signals multiplexer; in the center at top, an eight bit resolution ADC (Analog to Digital Converter) 0831CCN connected to a voltage regulator LM317 that provides a voltage of reference Vref of 2.5 Volts. Finally at left hand the parallel port of the computer represented by a DB25 connector, lines are used to represent the cable bus required to monitor and control circuitry.

Fig. 2. Proposed circuit to read sixteen temperatures and control parameters by means of parallel port

The array of sensors provides the sixteen input signals for the HEF4067B (Y0 to Y15) meanwhile parallel port controls the selected multiplexed signal A2 to A5 (pins 4 to 7), the multiplexed signal is selected as the Vin to the ADC0831, which is going to be converted to an eight bit serial data, considering a Vref of 2.5Volts provide by LM-327 Voltage regulator. The ADC CS signal is enabled by the parallel port DO (pin 2), and the bit selection is controlled by the parallel port D1 (pin 3) by generating the clock for the ADC. Reading of converted signal ADC’s DO is done at
parallel port’ ACK (pin 10). Ground is referenced at parallel port GND (pin 25). Control signals are provided by D6 y D7 (parallel port’s pins 8 and 9 respectively).

2.2 The application program

Figure 3 shows the implementation of the before analyzed circuitry implemented on a proto board, as can be appreciated implementation is very simple.

2.2.1 The Java code

As we stated in previous section, the application consists of a Server, a Client and the parallel port class. First, we will analyze the parallel port class.

The Parallel Port class

This class is used in several applications due its simplicity and usability regarding Java libraries [6, 8]; it has been used to control five axis arms for robots, remote control of cars, and to interface controllers and PC’s [2]. This class is available to work in Windows and Linux environments. The code and libraries required are free to download from Internet. Below we show a fragment of the class we used to read and write information from to and the parallel port.

```java
public class ParallelPort {
    /** The port base address (e.g. 0x378 is base address for LPT1) */
    private int portBase;
    /** To construct a ParallelPort object, */
    // you need the port base address */
    public ParallelPort (int portBase) {
        this.portBase = portBase;
    }

    //** Reads one byte from the STATUS pins of the parallel port.
    The byte read contains 6 valid bits,
    corresponding to 6 pins of input from the STATUS pins
    of the parallel port (the STATUS is located at
    "portBase + 1", e.g. the STATUS address for LPT1 is
    0x379)
    *
    * This diagram shows the content of the byte:
    *
    * Bit | Pin # | Printer Status | Inverted
    * --- | ------ |---------------|---------
    * 7   | 11     | Busy           | Yes     
    * 6   | 10     | Acknowledge    |         
    * 5   | 12     | Out of paper   |         
    * 4   | 13     | Selected       |         
    * 3   | 15     | I/O error      |         
    *
    * Note that Pin 11 is inverted, this means that "HI" input on pin means 0 on bit 7, "Low" input on pin means 1 on bit 7.*/
    public int read() {
        return ParallelPort.readOneByte(this.portBase+1);
    }
}
```

On previous figure are shown, the hexadecimal address (0x378) to access the parallel port lpt1, the parallel port description of the status pins and its respective hexadecimal status address (0x379). At the end of this figure is the code to read one byte from the parallel port.

On figure 6 we is shown the core code for the Server application. This code uses the parallel port class, and creates the GUI to interact with the user; this GUI contains the timers that make possible remote monitoring and control of temperatures. For simplicity, we show just the fragment of code where is possible to review the interaction of the interface with the parallel port class.
Sensors information is read one by one, first is written a set of instructions (888 decimal port address) to select the sensor from the multiplexer, then information provided by the sensor is loaded on the 8 bits length Analog to Digital Converter, once there, information is read bit by bit from the parallel port (889 decimal port address). Later information is converted to decimal, and then to a temperature (considering a reference voltage of 2.5 V). Information can be expressed on centigrade or Fahrenheit grades depending of user’s setup.

2.2.2 The implementation

To implement the Server System we used a recycled laptop Pentium II, with the following characteristics: 196 MB in RAM Memory, 5 GB in HD, a network interface card running at 100 Mbps and a Modem ADSL for Broad Band Internet connection. We installed Linux Red Hat 9.0 software for the Server side (we used open Suse 10.2 and Fedora 7.0 installed in some recent computers to compile and generate the .jar file), we enabled the http and ssh servers (web and secure shell servers) to make possible the usage of web pages and remote desktops, we also installed the Java Virtual Machine 6.1 for Linux, the parport class and its respective library for Linux. Finally we installed the resulting .JAR file containing the GUI generated in Netscape 5.5.1 to perform remote control and monitoring of temperatures.

To achieve results shown on figure 7, the client must open a session at Server side from the Client side, by means of SSH and Xming software to emulate a graphic terminal. The client will be asked for an specific user and password, once is introduced this information, to start the application, the Client must open the remote desktop and execute the JAR application by means of double click over the cup of coffee icon, or by typing the command java –jar GUIFormExamples.jar using a terminal. The application will start the control and monitoring of temperatures at Server Side, but information is displayed at Client Side.

Sensors information is read one by one, first is written a set of instructions (888 decimal port address) to select the sensor from the multiplexer, then information provided by the sensor is loaded on the 8 bits length Analog to Digital Converter, once there, information is read bit by bit from the parallel port (889 decimal port address). Later information is converted to decimal, and then to a temperature (considering a reference voltage of 2.5 V). Information can be expressed on centigrade or Fahrenheit grades depending of user’s setup.

2.2.3 Details for a successful implementation

We installed the software in Windows and Linux environments successfully. For Windows environments is mandatory to install the library –included on the download package from internet- parport.dll on the System32 directory. For Windows NT, Windows 2000 and XP, is necessary to install the useport.dll library [10] on the System32/drivers directory. In other hand, on Linux environment is mandatory to install the libparport.so on the /lib directory as the root user. Finally, is necessary to calibrate each sensor individually by means of a device considered like a standard (i.e. other temperature sensor from a well known supplier).
3 Preliminary results and discussion

We verify the performance of the application at local area network and remote locations and the application performance is OK, regarding the works of [2, 3, 7] our system is better, first because is possible to read sixteen different temperature signals, second provides a multiplatform environment (runs perfectly well in windows based systems as well as Linux based systems); third, we reuse recycled old computers contributing to diminish pollution of environment; four, we use gnu software which provides a low cost solution; five, is possible to control remote process, and finally we provide an open architecture that can be modified for a myriad of applications.

Of course we are aware of the system’s limitations like the requirement of the internet, but the question is where is not available a kind of internet connection today? Another limitation is the maximum distance of the sensors from the parallel port, but we can solve this problem by using a different standard.

3.1 Further work

We are working on the improvement of the graphic user interface (GUI), we are considering the use of wireless sensors for monitoring temperature, relative humidity, and luminescence, and we would like to use our system in real world applications like green house monitoring and control, and the generation of expert systems with sensing capabilities for the improving of diagnosis for human and plant diseases.

4 Conclusions

The above studied technology is useful, provides an open architecture to deal with different problems of remote monitoring and control, besides provides a low cost alternative thanks the use of gnu software and multiplatform code generated on Java, and, maybe more important, contributes to diminish earth pollution by recycling e-waste from computers.

5 Acknowledgments

We want to thank to CONACYT due part of this research is supported by project 47220.

6 References


Managing Requirements for e-Learning Platforms

D.D. Burdescu¹, M. C. Mihăescu¹, and B. Logofătu²
¹Software Engineering Department, University of Craiova, Romania
²CREDIS Department, University of Bucharest, Romania

Abstract - Currently, there are many open source and proprietary e-Learning platforms. One of the big issues when developing such systems is requirements engineering. This is due to the large number of facilities that need to be implemented and the large number of flavors in which facilities themselves may be presented to end-users. This paper presents a custom solution for extracting and managing requirements. This step of the overall software engineering process becomes even more critical when the development process is to become a global one. The main outcome of this paper is an approach for passing from ad hoc requirements to systematic requirements management. This step is also needed due to the increasing scale of software system and increasing globalization. The study is presented for an e-Learning platform that has been developed and reached a certain maturity level such that these kind of activities are needed. The proposed solution in the paper allows centralization of requests among involved parties (developers and beneficiaries). It is proposed a custom structure for requests that has at its basis the very specificity of developed application. The structuring of requirements is customized for platform’s functionality such that the process which determines what functionalities may be outsourced is improved.

Keywords: requirements engineering, e-Learning, software development.

1 Introduction

The success of a software system depends on how well it fits the needs of its users and its environment [1, 2]. Software requirements comprise these needs, and requirements engineering (RE) is the process by which the requirements are determined. Successful RE involves understanding the needs of users, customers, and other stakeholders; understanding the contexts in which the to-be-developed software will be used; modeling, analyzing, negotiating, and documenting the stakeholders’ requirements; validating that the documented requirements match the negotiated requirements; and managing requirements evolution.

Requirements engineering is not an easy task. There are many reasons that for this situation. Ideas may be ill defined or even conflicting. This situation usually has to shift towards a single, sound, detailed technical specification of the software system. The requirements problem space is usually not that strictly defined. In complex systems there are many too many options to consider. That is why management of large collections of proposed requirements, prioritization, specification of system boundaries, negotiating resolutions to conflicts, setting objective acceptance criteria [3] are processes that usually need to be carried out. Reasoning and having conclusions about the software that is to be designed and developed includes identifying not only assumptions about the general normal behavior of the environment represented by main scenario use cases, but also about possible threats or hazards that may appear in the system. The obtained top level requirements have to be understood and usable by all involved persons: developers and beneficiaries. Thus, requirements notations and processes must maintain a delicate balance between producing descriptions that are suitable for beneficiaries which may be represented by non-computing audience and producing technical documents that are precise enough for domain experts and downstream developers.

Due to above presented reasons, requirements engineering activities, in contrast to other software-engineering activities, may involve many more players who have more varied backgrounds and expertise, require more extensive analyses of options, and call for more complicated validations of various components such as software, hardware or even human.

Regarding requirements engineering there are several main activities that need to be taken into consideration: elicitation, modeling, analysis, validation/verification and management. Requirements elicitation regards activities that enable the understanding of the goals, objectives, and motives for building a proposed software system. Elicitation also involves identifying the requirements that the resulting system must satisfy in order to achieve these goals. This activity rises problems regarding stakeholders’ identification [4], contextual and personal requirements engineering techniques [5, 6]. In requirements modeling, a project’s requirements is expressed in terms of one or more models. Modeling strategies provide guidelines for structuring models. For example, requirements engineering reference models [7, 8, 9] decompose requirements-related descriptions into the stakeholders’ requirements, the specification of the proposed system, and assumptions made about the system’s environment. Problems regarding requirements analysis are mainly linked on the techniques employed for evaluating the quality of gathered requirements. Some analyses look for well formedness
errors in requirements, where an “error” can be ambiguity [10, 11], inconsistency [12, 13], or incompleteness. Other analyses look for anomalies, such as unknown interactions among requirements [14, 15], possible obstacles to requirements satisfaction [16, 17], or missing assumptions [18]. Requirements validation make sure that obtained models and documentation express as possible as accurate the needs of involved persons, beneficiaries or developers. Validation usu-ally requires that beneficiaries and developers to be directly involved in reviewing the requirements [19]. Main issues in this area regard improving the information provided to the involved parties including animations [20] or simulations [21].

Requirements management is a general activity that regards tasks related to the management of requirements, including structuring and evolution over time and across product releases. Currently there are used specific tools that partially automate different tasks (e.g. identification, traceability, version control) [22, 23]. There may be also employed tools that estimate the maturity and stability of elicited requirements, so that the requirements most likely to change can be isolated [24]. Organization and structuring of large numbers of requirements [25] that are globally distributed [26], and that are at different phases in development in different product variants [27] are current issues that appear in complex and large systems.

This paper presents advances made within an e-Learning platform called Tesys [28] regarding requirements engineering. This platform has initially been designed and implemented only with core functionalities that allowed involved people (learners, course managers, secretaries) to collaborate in good conditions. The requirements engineering followed an ad-hoc process that informally followed the classical life-cycle: elicitation, modeling, analysis, validation, verification and management. The involved parties were represented by three parties: development team, beneficiaries and end-users. Firstly, a prototype that implemented main functionalities has been developed. The requirements were elicited and negotiated between development team and beneficiary. After prototype has been deployed the e-Learning system has been populated with data and users. The beneficiary was the one that kept a close relation with end-users and closely looked the effectiveness of the platform.

The features and functionalities rapidly grow, such that in less than an year the development team faced a large scale software system. Under these circumstances there had to be found specific solutions to encountered problems regarding requirements engineering. The software system became large in size. The reason for calling it a large-scale system do not necessarily refer to significant size regarding lines of code. Scale factors also include business logic implemented complexity or degree of heterogeneity among assets. Another important scale factor is variability, as software system needed to accommodate increasingly larger sets of requirements that vary with respect to changes in the software’s environment. Requirements started to come from many different involved persons (secretaries, professors, students), involve multiple aspects (e.g., need for additional functionality, modify existing functional-ity, implement more complex activities). There was discovered the need for increasing the reliance and self management of the environment. Bringing around new teams of developers and the need to keep them closer with the beneficiary was the decisive step in going towards global software development. Global software development is an emerging paradigm shift towards globally distributed development teams [29]. The shift is motivated by the desire to exploit a 24- hour work day, capitalize on global resource pools, decrease costs, and be geographically closer to the end consumer [26]. The downside is increased risk of communication gaps. For example, elicitation and early modeling are collaborative activities that require the construction of a shared mental model of the problem and requirements. However, there is an explicit disconnect between this need for collaboration and the distance imposed by global development. Decisions regarding requirements engineering in this direction are having a huge impact regarding future possibility of outsourcing. In this context, outsourcing may be seen as a particular case of globalization.

Globalization raised two main problems for involved people. First, new or ex- tended requirements engineering techniques are needed to support development tasks, such as design, coding, or testing. Since geographical distance aggravates the gap between teams (e.g. development, beneficiaries, requirements, etc.), particularly if the teams are from different organizations, have different cultures, or have different work environments. In particular, because geographic distance reduces team communication [30], ill-defined requirements are at risk of ultimately being misinterpreted, resulting in a system that does not meet the envisioned needs. As a preliminary effort to narrow communication gaps, Bhat et al. [26] have proposed a framework based on a people process- technology paradigm that describes best practices for negotiating goals, culture, processes, and responsibilities across a global organization.

The second problem is to enable effective distributed requirements engineering. Future requirements activities will be globally distributed, since requirements analysts will likely be working with geographically distributed stakeholders and distributed development teams may work with in-house customers. As such, practitioners need techniques to facilitate and manage distributed requirements elicitation, distributed modeling, distributed requirements negotiation, and the management of distributed teams – not just geographically distributed, but distributed in terms of time zone, culture, and language.

2 Tesys Application Platform

An e-Learning platform that represents is a collaborative environment for students, professors, secretaries and administrators has been designed and developed. Secretary
users manage sections, professors, disciplines and students. The secretaries have also the task to set up the structure of study years for all sections. The main task of a professor is to manage the assigned disciplines. The professor sets up chapters for each assigned discipline by specifying the name and the course document, and man-ages test and exam questions for each chapter. The platform offers students the possibility to download course materials, take tests and exams and communicate with other involved parties like professors and secretaries.

All users must authenticate through username and password. If the username and password are valid the role of the user is determined and the appropriate page is presented.

A message board is implemented for professors, secretaries and students to en-sure peer-to-peer communication. This facility is implemented within the platform such that no other service (e.g. email server) is needed.

From software architecture point of view, the platform is a mixture of data access code, business logic code, and presentation code. For development of such an application we enforced the Model-View-Controller [31] (MVC for short) design pattern for decoupling data access, business logic, and data presentation. This three-tier model makes the software development process a little more complicated but the advantages of having a web application that produces web pages in a dynamic manner is a worthy accomplishment.

From the software development process point of view we enforced the cyclic software development with project planning, requirements definition, software architecture definition, implementation, test, maintenance and documentation stages. Software development makes intensive use of content management through a versioning system, testing and continuous building infrastructure.

3 Improvements in Requirements Engineering Process for Globalization

Requirements engineering is the first step in making the development process a global one. Analysis and changes in this direction may have good implications regarding the possibility of future outsourcing of specific activities. Proposed solutions are specific for e-Learning environments in general and to Tesys e-Learning platform in particular.

3.1 Structuring Requirements on User Groups

The first decision is to structure requirements based on user groups. Within Tesys there were defined three main roles: Secretary, Professor and Student. Each user that accesses the platform has to have one of these roles. During prototype development phase the requirements elicitation and negotiations between development team and beneficiary were always carried out for specific role functionality. This was mainly due to disjunctive implemented functionalities for these roles.

This decoupling is very natural due to employed software architecture presented in chapter three. The decoupling may be seen at all three MVC levels: view, business logic and model. The functional requirements are very linked with the view tier. That is why the templates that represent this tier are grouped in specific folders according to the user role that will display them.

This decision may have important impact in globalization or even outsourcing since a new group of developers may start working on adding functionality.

Under these circumstances, a requirement started to become an object with his own status and life cycle. The status is determined by the set of values of fields. There has been defined the following set of fields:

- **Id** – uniquely identifies the requirement;
- **Role** – defines the role to which the requirement addresses;
- **Activity** – defines the activity to which the requirement addresses;
- **Status** – there were defined three states: INWORK, SOLVED, VERIFIED;
- **Solver** – person responsible for implementing the requirement;
- **Memo** – text that represents a short summary about the requirement.

Firstly, the requirement has to be signaled from the beneficiary. The beneficiary may have the requirement either simply as a need for more functionality or as a bug from a user (secretary, professor or student). The requirement is negotiated with development team and a conclusion is reached. At this phase there are identified all values of fields, which means there is identified the Role and Activity, it is set the Solver and the Memo, and it is assigned an Id and Status is set to INWORK. Basically, from this moment the Solver is announced that he has some work to do. As soon as the Solver accomplishes his job he will signal this by changing the status of the Requirement to SOLVED. This means that the Quality Assurance people may start testing the functionality. At this step they will be aware of the exact functionality it refers from Role, Activity and Memo fields.

At this point there are obvious the advantages of this way of structuring requirements. If Quality Assurance people do not agree with implemented solution they may change the status back to INWORK such that the Solver will be announced that his solution does not meet the required quality. In this situations there is advisable that additional communication has to take place between management and Solver.

3.2 Determining the Benefit of Requirements Management

When the problem of globalization appears, the next question is: “What?”. In the next table [42] it is presented the percentages of outsourced and in house activities.
Table 1. Percentages of outsourced and in-house activities

<table>
<thead>
<tr>
<th>Activity</th>
<th>Outsourced</th>
<th>In-house</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Management</td>
<td>30%</td>
<td>90%</td>
</tr>
<tr>
<td>Requirements Gathering</td>
<td>17%</td>
<td>89%</td>
</tr>
<tr>
<td>Architecture</td>
<td>19%</td>
<td>88%</td>
</tr>
<tr>
<td>Research and Development</td>
<td>25%</td>
<td>78%</td>
</tr>
<tr>
<td>Business Integration</td>
<td>16%</td>
<td>76%</td>
</tr>
<tr>
<td>Design</td>
<td>51%</td>
<td>77%</td>
</tr>
<tr>
<td>Systems Integration</td>
<td>35%</td>
<td>76%</td>
</tr>
<tr>
<td>Deployment</td>
<td>26%</td>
<td>75%</td>
</tr>
<tr>
<td>Testing</td>
<td>74%</td>
<td>71%</td>
</tr>
<tr>
<td>Modeling</td>
<td>26%</td>
<td>69%</td>
</tr>
<tr>
<td>Maintenance</td>
<td>53%</td>
<td>65%</td>
</tr>
<tr>
<td>Code migration</td>
<td>54%</td>
<td>42%</td>
</tr>
<tr>
<td>Coding</td>
<td>94%</td>
<td>41%</td>
</tr>
<tr>
<td>Internationalization</td>
<td>39%</td>
<td>34%</td>
</tr>
</tbody>
</table>

From the above highlighted percentages it became obvious that in requirements gathering is not an usual activity that is outsourced. Still, coding is by far the most outsourced activity. Since coding is the most likely activity to be outsourced it means specific measures are to be taken regarding structuring of assets. Code, along templates, data models, data itself, etc. represent the assets of the e-Learning platform. Requirements gathering and management and coding have to be very well coordinated such that the productivity is maximized. The structure presented in previous chapter is very well suited for determining groups of activities that belong to a certain role and may be outsourced.

An obvious scenario is when many requirements appear for the same role and same or related activities. In this situation there may be decided that the corresponding piece of software to be outsourced in the effort of globalization.

3.3 Benefits Regarding Verification and Validation

Requirements engineering relies fundamentally on verification and validation as a way of achieving quality by getting rid of errors, and as a way of identifying requirements. One benefit from structuring requirements is the use of automation for verification of requirements. The requirements may be inspected such that verification is performed by using well established checklists. The checklists are applied to the requirements by a well established process.

Modeling requirements in a custom structured form provides the opportunity for analyzing them. Analysis techniques that have been investigated in requirements engineering include requirements animation, automated reasoning, consistency, and a variety of techniques for validation and verification that are further discussed.

Validation is the process of establishing that the requirements and derived structures provide an common and accurate base for involved persons (developers and beneficiaries). Explicitly describing the requirements is a necessary precondition not only for validating requirements, but also for resolving conflicts between developers and beneficiary.

Difficulty of requirements validation comes from many sources. One reason is the problem itself is philosophical in nature. This makes the formalizing process hard to define. On the other hand, there is a big difficulty in reaching agreement among involved persons due to their conflicting goals. The solution to this problem is requirements negotiation. These will attempt to resolve conflicts between involved parties without necessarily weakening satisfaction of each person’s goals.

Structuring requirements brings a big advantage for validation and verification in case of changing requirements. As all successful systems, our e-Learning platform evolves. This means that when a functionality changes because of beneficiary and developer negotiated such a change, this transition needs to be done with minimum of effort. For this, requirements have to be traceable and this feature is accomplished by proposed structuring.

3.4 Employing a systematic process of requirements engineering

Lack of systematic process in requirements engineering is one of the critical problems. The proposed way of structuring requirements defines some steps regarding the process of requirements engineering. These steps are:

1. Define the requirement – this is accomplished by the beneficiary in negotiation with the development team. The accomplishment of this step is marked by status INWORK.
2. Work the requirement – this is accomplished by the development team. The development team changes the status of the requirement from INWORK to SOLVED when the solution is implemented and thus it may be tested.
3. Test the functionality – this is accomplished by a test team or beneficiary. It checks whether the implemented solution is acceptable or not. If it is acceptable, the status will be changed to VERIFIED. Otherwise, the status will be reset to INWORK, marking thus the fact that the implementation is not satisfactory. In this situation an explanatory text is needed such that the development team is aware appeared complaints.
4 Conclusions

In this paper, there were presented the main challenges in requirements engineering. All of the problems described in introduction (elicitation, modeling, analysis, validation, verification and management) require substantial effort in order to make progress towards effective solutions. In this general context there was presented Tesys e-Learning platform from functional, software architecture and software development point of views. It has been presented the initial requirements engineering process that was used when the proto-type has been developed. Currently, there is big effort for globalization of software development process since the application is rapidly growing in size. More than this, the business logic complexity, degree of heterogeneity among assets are increasing.

From requirements point of view there were adopted two solutions. One regards the custom and proper structuring of requirements. This was accomplished according with the nature of the application, in our case represented by an e-Learning platform.

The benefits from this approach are multiple. Centralization and proper structuring of requirements had a big impact in management activity of the project. Although managing the effort of centralization is big at beginning, for future development it is supposed to have a good return of investment.

Other benefit is that there may be created pools of requirements based on functionality at role level and even with a higher granularity at activity level. This will have a big impact on future decisions regarding what parts of software to be out-sourced in the effort of globalization.

Finally, there presented the benefits brought by our structuring to verification and validation processes. The proposed structure ensures traceability of requirements, such as the system evolves the requirements are still properly managed. More than this, there presented the benefits regarding verification and validation of requirements and the proposed process of requirements engineering. The process has three states and is simple enough such that it needs minimal managerial effort.

4 References

Object-Oriented Hypermedia Design and J2EE Technology for Web-based Applications

Habib Karimpour¹, Ayaz Isazadeh², Mohsen Heydarian³
¹Faculty of Computer Education, Islamic Azad University of Maimaghan, Maimaghan, Iran
²Faculty of Computer Science, University of Tabriz, Tabriz, Iran
³Faculty of Computer Education, Islamic Azad University of Maimaneh, Maimaneh, Iran

Abstract - Web-based application development is a difficult task, since these applications include various features, like graphical interfaces, navigational structures, business models, and wireless communications, as well as other issues, such as serving a multitude of users, and the need for shorter development time. To overcome these complexities, it is indispensable to use web-based application designs and software architectures. In this paper, we study the Object-oriented Hypermedia Design Methodology for the design of web-based applications. We use a multi-tiered, component-based software architecture based on the J2EE technology. Unfortunately, the J2EE technology lacks rich hypermedia and navigational structures, and the mapping of prepared designs to components is not carried out properly. For this reason, in this article, a combined architecture called OOHDM-J2EE is defined which uses the benefits of both schemes and solves the mentioned problems, provides the possibility of reuse, and further reduces the development time.

Keywords: object-oriented hypermedia design, navigation, decomposition, component-based architecture, multithreading.

1 Introduction

Nowadays, the web is one of the most important communication channels which can be accessed on various platforms around the clock all over the world. The internet encompasses more than 10 trillion pages and has about 1 billion users. While the nineteenth century was the pinnacle of industrial revolution, the beginning of the current century is characterized by the information revolution, and the web is the principal engine of this revolution. Noting the ubiquity of the web, many information systems are accessible via the web, and they are called Web Information Systems (WIS).[2]

Modern WIS applications present the information in meticulously crafted, multimedia-enabled texts. This system is not sufficient for the ever-growing need of publishing high volumes of information over the web. Information-intensive web features, like electronic cities, organizational portals, social engineering sites, e-shopping centers, digital libraries, and the like, are present in many web-based applications.

1.1 Operational Terminology

Navigation: guiding the users from one place to another. Even though the web is not a physical space, navigational procedures are necessary, so that the users can move in the information space.

Context: a set of navigational nodes with a common property.

Container: a place on the server where application components are situated and are executed therein. Containers provide middle-tier services for the components.

1.2 Problem Description

Web information systems allow a user to search in the plethora of information in the internet and perform tasks like research and updating. Some of the specific features in these programs are as follows:

1. They must be developed with high confidence in a short period of time, because the time to market is a critical factor in the success of web-based applications.

2. They must be able to guide the users in the application space in a satisfactory manner.

3. They must be able to respond to large numbers of users.

4. They must integrate with modern telecommunication technologies, in order to support access through mobile devices.

5. They must support a high degree of reusability.

Therefore, in this article, we are going to find a suitable solution for designing applications with these features.

1.3 Theoretical Background

We use the component-based architecture of J2EE (Java 2 Enterprise Edition) to present solutions for reusability, rapid development, multi-user service, and rely on the object-oriented hypermedia design paradigm for navigation, graphical interface design, and reusability. We believe that the combination of these two methodologies maximizes the
capabilities for reusability and rapid development for web-based applications.

1.4 Objective

Separation of various parts of web development paves the way for reusability and parallel development, effectively shortening the application development time. However, in most cases these benefits are lost in the implementation phase, mostly because the current implementation methodologies have a meager support for mechanisms of abstraction and composition. For this reason, our major objective is a mapping from the design artifacts to implementation components, and adding navigational capability to applications.

1.5 Historical Background

The earliest WIS applications delivered hypermedia content to users in the form of static web pages. Today, with the growing acceptance of the web, users can access up-to-date information presented by dynamic pages and interact with the web sites using the form functionality. The next generation of applications, called Semantic Web Information Systems, can obtain information from one or many sources and provide the user with an integrated version of that information. The design of WIS applications is a very difficult task due to the high volume of information, customization, large number of users, and the need for automatic creation of programs. Some of the methodologies are model-driven, and use models for characterizing various aspects of a WIS. Model-driven methods have some advantages, like better use of components, reusability, improved maintenance and evolution capability, and the possibility of program validation [2].

1.6 Motivation

The traditional design of WIS applications has created dissatisfaction among users due to long development and response time, lack of navigational capabilities, and dozens more problems. To overcome these shortages, we have no other way but to combine software engineering approaches (component-based architecture) with web engineering techniques.

1.7 Overview of the Article

This paper consists of 4 sections. In Section 2, we will review previous studies, including component-base architecture (J2EE) and object-oriented hypermedia design. In Section 3, we will focus on the mapping of the design constructs and we will add navigational capability to the components. Finally, in the last section, conclusion and future research directions will be discussed.

2 Research Background

2.1 Multi-tiered Design

When an application program is developed for a standalone computer, it is expected naturally to include all input form functions, data processing logic, or data access routines. Figure 1 shows such a program at run time.

![Figure 1: Architecture of an Integrated System.](image)

In Figure 1, if anything is changed in one part of the program, it will affect all other parts of the program, because all of them are integrated as a single unit. For this reason, if we separate each part as an independent layer, we can provide a possibility for reusability, decoupling of program parts form each other, more flexibility, and freedom in implementation of each part independent of other parts of the application. In addition, more flexibility in presentation design and the possibility of running the application on a server makes this design methodology an ideal prototype for web environments. Figure 2 shows a three-tiered structure for this integrated design.

![Figure 2: Architecture of a Three-tiered System.](image)

If we have an integrated system and convert it to a distributed (multi-tiered) system, so that many users can connect to it concurrently and it can be connected to multiple servers and databases at the same time, there are a few things that we may worry about. These include remote client access, session and transaction management, security, and resource management. Now if we decide to implement and maintain all of these functionalities, which are collectively called system service or middleware layer, we will have a difficult time doing the job. Therefore, today,
there are technologies that provide us with the necessary middleware services. One of these is the J2EE technology, which is one of the main solutions for construction of complex web-based applications [1]. J2EE is inherently a component-based architecture. Figure 3 shows two J2EE-based applications which are divided into several layers, including:

- **Client component layer**, which runs on the client machine.
- **Web component layer**, which runs on the J2EE server.
- **Business component layer**, which runs on the J2EE server.
- **Enterprise Information System (EIS) layer**, which runs on the EIS server.

![Figure 3: J2EE Architecture](image)

Each of these layers is composed of components that are executed inside a container. The container provides these components with middleware services. Additionally, the J2EE server runs in a multithreading mode, which is an important factor and enables us to respond to large numbers of users simultaneously, minimizing the response time [1].

### 2.2 OOHDM

Good application programs must be good hypermedia applications. Since the web is based on the hypertext design principle, we can use web-based design guidelines to support the hypermedia and navigation functionalities [4]. By separating the various parts of web-based development, we can reduce problem complexity, enhance reusability, and shorten the development period. According to the Object-oriented Hypermedia Design Method (OOHDM), the development of hypermedia applications is broken up into four steps, including conceptual design, navigational design, abstract interface design, and implementation. These activities are performed as a mixture of iterative and incremental methods.

#### 2.2.1 Conceptual Modeling

During this activity, we build a conceptual schema representing objects, their relationships and collaborations existing in the target domain. We model the target domain using the Unified Modeling Language (UML), an inherently object-oriented modeling tool, in the form of classes and associations between them. Figure 4 contains a simple schema for an online magazine. In this model, there are stories, which can be essays, translations or interviews; an interview is an aggregation of questions and answers. Every story has an author, and an interview is also related to the person who grants the interview.

Object-oriented modeling uses aggregation, generalization/specialization, and a packaging concept, called sub-systems, to enhance reusability and shorten development time [5].

![Figure 4: Conceptual Model for an Online Magazine](image)

#### 2.2.2 Navigation Design

Navigation is an important aspect in web-based application development, so that the user should be able to reach the desired part of the program with just three clicks. Unfortunately, most organizations do not pay enough attention to navigation and hypermedia functions when deploying their web applications and this confuses the users, so that they can not reach their desired goals. And in some applications, even though the developers have considered navigation, a consistent navigational structure is not applied.
to the whole application, so that the site becomes a huge set of disheveled links and nodes.

In OOHDm, navigation is built as a view over a conceptual model, thus allowing the construction of different models according to different users’ profiles. Navigation design is expressed in two schemas, the Navigational Class schema and the Navigational Context schema. In OOHDm, there is a set of pre-defined types of navigational classes: nodes, links, and access structures. Navigation nodes, which are built as views of conceptual objects, are called navigation objects. If some of the classes in the conceptual model are not related to navigation, they may be deleted in navigational objects or be converted to attributes in other classes. This case is shown in Figure 5. Access structures, such as indices and guided tours, represent possible ways of accessing nodes.

![Figure 6: Navigational Context Schema for the Online Magazine Application.](image)

2.2.3 Abstract Interface Design

Eventually, what the user sees is the interface. Even the navigational objects are not felt directly by the user, but they are presented to the user by the abstract interface. For a single navigation model, we can create different interfaces, so that the users can access the application from different devices, like mobile phones, or various types of browsers.

2.2.4 Implementation

Any application-specific model, like OOHDm design, must be eventually implemented using an implementation technology. As it was mentioned previously, we have chosen the J2EE technology for implementation, because it supports component-based, multi-tiered, multithreaded designs and has many other advantages, as well. In the remaining sections of this paper, we will describe the combination of OOHDm and J2EE, OOHDm-J2EE, which is an architecture paradigm that makes it possible to decouple design decisions concerning the application domain from decisions about navigation and interface design. Furthermore, this methodology exploits the benefits of both technologies and maximizes the reusability opportunities.

Why We Need Software Architecture?

Mapping design documents into implementation artifacts is a difficult task. For example, as it is shown in Figure 7, the designer may erroneously include business rules inside JSP pages, instead of placing them inside EJB components. For this reason, there is a visible trend in software architecture toward modular approaches. For example, we
can consider that the MVC architecture (which is consisted of three modules: model, view, and controller) is used for improving the development of J2EE-based applications, and has enhanced earlier proposals in which most architectural design decisions were left to the programmer’s savvy or ability.

In the MVC architecture, the “View” which is built by JSP components is responsible for implementing the user interface (page layout), and the “Model” which is made of EJB components is responsible for implementing the business logic of the application.[3]

While the MVC provides a set of structuring principia for building modular interactive applications, it does not completely fulfill the requirements of web applications for providing rich hypermedia structures. Concretely, in a naïve use of the MVC, nodes and their interfaces are handled by the same JSP components. In addition, navigational contexts are totally absent in the MVC architecture. If we want to support navigational contexts in JSP, we have to use the context as a parameter for the JSP page, and write conditional statements to insert context sensitive information as appropriate. The JSP becomes overloaded, difficult to manage and evolution becomes practically unmanageable. Now, if we use different JSP pages for different contexts, we will have even more problems. Summarizing, the main problem is that the navigational logic (node and context management) is dealt with in the JSP page, which mixes it with the interface layout, which is also a responsibility of the JSP page.

Figure 7: The MVC Architecture and Mapping of Design Documents.

Figure 8: The response to the Request Manager component, which translates it into a business event. The Request Manager uses auxiliary objects called Request Handler that know how to handle different http requests. The Request Handler contains the logic for generating a business event; an XML configuration file establishes the correspondence between http requests and the Request Handler. After translating the request into an event, the Request Manager sends this event to the Web Controller. This is a component that resides inside the scope of the user's session and is instantiated by the Request Manager in the first user request. The Web Controller is responsible for interfacing with the EJB Controller and providing access to conceptual objects.

Upon reception of the event, the Web Controller redirects it to the EJB Controller. This object controls the EJB layer, i.e., the business layer. The EJB Controller invokes the appropriate business objects, and notifies the Web Controller of the resulting changes to be reflected in the observer objects. The EJB Controller invokes the Event Handler, which has the specific logic for executing the events, i.e., knows which conceptual objects (EJBs) to invoke, when and how. For each business event there is a corresponding Event Handler and this correspondence is defined using an XML configuration file.

At this stage we need to send a response back to the client. The Front Servlet invokes a black-box component, the Interface Manager, which selects the response interface. When the mapping between the http request and the

3 New Findings

3.1 OOHDJ-J2EE

To solve the above-mentioned problem, we propose to build a navigation layer that includes all of the navigational logic. While JSP is responsible for the page layout, the navigational layer manages the node content and handles context-specific information.

The OOHDJ-J2EE architecture corrects the view part of the MVC architecture by adding a navigational layer. Now, the modified view sections comprises of the user interface (page layout) which is created by JSP and navigational nodes.

In Figure 8 we show the complete process for treating an http request. The following description is based in a 3-tier scenario (web tier, EJB tier, and persistence tier).

All http requests are re-directed by the application server container to the Front Servlet component, the single application entry point. When the Front Servlet is created it must initialize the application, instantiating all global application components.

Treatment of the request occurs in the scope of its main method that is executed within a thread. Front Servlet is a black-box component provided by the framework.

The Front Servlet sends the http request to the Request Manager component, which translates it into a business event. The Request Manager uses auxiliary objects called Event Handler that know how to handle different http requests. The Request Handler contains the logic for generating a business event; an XML configuration file establishes the correspondence between http requests and the Request Handler. After translating the request into an event, the Request Manager sends this event to the Web Controller. This is a component that resides inside the scope of the user's session and is instantiated by the Request Manager in the first user request. The Web Controller is responsible for interfacing with the EJB Controller and providing access to conceptual objects.

Figure 7: The MVC Architecture and Mapping of Design Documents.
response interface is not direct, the Interface Manager invokes the Interface Handler, containing the logic for selecting the interface based on the request and on the state of conceptual objects. The mapping between the http request and the response interface (and if necessary the Interface Handler) are also specified using an XML file.

In OOHDM-Java2 an interface template is a JSP page containing place holders, known as template parameters. There are two genres of template parameters; those whose value is a text to be inserted directly, and those whose value is another JSP page whose contents are inserted dynamically. Template pages are defined using the custom tag parameters containing an attribute defining the parameter name or placeholder holder of the template. A template plus its parameter's values define an interface. Application interfaces are also defined using XML files. When the response interface contains a JSP page that exhibits a navigational node or an index, it is necessary to instantiate them using components Navigational Node or Index. This JSP invokes the Navigational Manager component using the custom tag create_node or create_index. The Navigational Manager invokes the corresponding factory object (Node Creator or Index Creator) containing the specific logic for creating the node or index. These objects access appropriate conceptual objects through the Navigation View interface, which is responsible for implementing the view over the conceptual object. The Navigational View interface declares three methods: getCustomNodeIDs, getobjectList, and getobject. The first method returns the node identifiers (NodeIDs) of a navigational context. The second return the list of objects that contains the data used to create each index entry of an index. The last method returns a state object encapsulating the actual data used to create the navigational node in a context. Another important component (not shown in Figure 5 for the sake of simplicity) is the Navigational Context, which represents the set of nodes the user is currently navigating. This component interacts with the Navigation View component to access the conceptual objects corresponding to the nodes in the current context. The Navigational Manager interacts with the Navigational Context to provide adequate contextual information for a node creator or node index creator to instantiate the node or index. This information includes ID of the next and previous nodes, number of elements in the context, the URL(s) where the node in context is exhibited, etc. In other words, when the node is created, the linking information is retrieved and inserted into the node by The definition of contexts and the correspondence between contexts and the corresponding Navigational Context and Node Creator are specified using XML configuration files.

Finally, if the http request does not have an associated business event, i.e. it is only a navigational request, the process is fairly simple; the interface response must be defined, the corresponding node (in context) must be created and the JSP must generate the layout. In this case, the Front Servlet invokes the Interface Manager and processing follows from that point.

![Figure 5: Treating an http request in OOHDM-J2EE](image)

## 4 Results and Conclusion

### 4.1 Proof of the Theory

In this article, we presented a solution for the problem discussed in Section 1.2, namely:

1. In Section 2, we used OOHDM methodology to guide the users in the site and applied the hypermedia principles for the application.
2. Using a multi-tiered, component-based architecture, we realized decoupling of layers form each other and enhanced reusability capacity and shortened development time.
3. We exploited the advantages of the J2EE technology, like multithreading and rich libraries, to respond to large numbers of users.
4. In Section 3, with separation of the presentation layer form other parts and with definition of multiple presentation logics, we could serve users on different devices (PC, PDA, etc.).
4.2 The Goal
Using a combination of J2EE and OOHDM, we mapped the design artifacts created in OOHDM easily into components and maximized reusability and time to market. Subsequently, we added navigation as an important concept to the web-based applications.

4.3 Related Works, Discussion, and Comparison

J2EE technology, in comparison with other technologies like .NET, has advantages like scalability which is evident in some of use cases like electronic cities, university exam results portal, gas smart card, centralized account of Bank Mellli Iran, Ministry of Interior's national polling site, and many others.

Using a J2EE application server, we can exploit the many libraries and classes of this technology for various purposes. An application server is a software infrastructure that is responsible for application lifecycle management. In other words, it responds to the needs of a large-scale application, including database, web server usage, mail sending (via SMTP protocol support in J2EE), authentication (using Kerberos technology support in J2EE), smart card transactions (using the smart card API in J2EE), and many others, throughout the lifecycle of the application. Currently, J2EE is the only system that can be used for supporting smart cards in the application. Scalability is an important advantage of J2EE over other technologies like .NET framework, and for this reason, this technology is used for large projects where a mistake in choosing the suitable architecture and occurrence of incompatibilities could be unforgivable.

The difference between these two technologies is that learning .NET is easier and its complexity is lower than J2EE framework. The learning curve for Java is less steep (a steep curve would mean easier learning), while learning .NET is much more rapid. The run time in .NET is similar to Java. The JSP server has the capability of clustering and the servlets run in a multithreading environment. All of these advantages make J2EE preferable over .NET for serving large population of users.

4.4 Findings of the Article
The OOHDM-J2EE architecture provides these capabilities:

- Further separation between navigation logic and interface aspects.
- Since the components are run in a multithreading environment, it has short response time and it can respond to large groups of users simultaneously.
- It supports navigation contexts and sets of navigational nodes.
- Decoupling between JSP pages and business events.
- Centralized control of HTTP requests.
- Centralized control of business events execution.
- Centralized control on the selection of response interfaces.
- Single entry point for serializing requests of the same user.
- Centralized control of navigation logic.
- Changes in the navigational structure do not impact on the business logic, and vice versa.

4.5 Future Research Topics
We plan to make a tool for automatic creation of OOHDM-J2EE code. Additionally, we intend to design methodologies for the semantic web. These will be based on model-driven web methodologies. In other words, we will use all the advantages of model-driven methods to create a semantic web methodology, and we will add information integration concepts to this methodology.

5 REFERENCE


SESSION

SOFTWARE EVOLUTION

Chair(s)

TBA
Visualization of Software Evolution

Young Lee\(^1\) and Jeong Yang\(^2\)
\(^1\)Department of Electrical Engineering & Computer Science, Texas A&M University-Kingsville, Kingsville, TX, USA
\(^2\)7517 Vaquero Dr, Corpus Christi, TX, USA

Abstract - This paper presents a case study to analyze how a software system has evolved. The case study is to provide the global visualization of the evolution of an open source software system and to provide effective ways to analyze the evolution of the system. We present Evolution Track Table to observe the change along the software evolution. Evolution Track table is specially designed to give an overview of the software evolution history as a part of JamTool research studies [6, 7]. We examine twenty-two released versions of JFreeChart [1], which is an open source charting library, as a target system. The study apparently indicates that using Evolution Track Table, we could categorize the evolution of classes by characterizing the evolution pattern of a software system, identify unusual evolution pattern of the classes, and find out dramatically changed releases during the evolution.

Keywords: Software Evolution Analysis, Software Visualization, Open Source Software, JamTool, JFreeChart.

1 Introduction

Research on how a software system evolves over time is difficult and time consuming. The enormous amount of work required by analyzing software evolution makes it difficult without the dedicated tools such as JamTool [6, 7]. Automated environments could be key factors in conducting a successful study on software evolution.

Moreover, there are two major challenges that must be overcome in software evolution research. These challenges limit our ability to understand the history of software systems, thus prevent us from generalizing our observations into software evolution theory. The first challenge is how to organize the enormous amount of historical data in a way that allows us to access them quickly and easily. The second challenge is how to analyze the structural changes of software systems.

To overcome these challenges, we designed Evolution Track Table. Evolution Track Table uses visualization technique in a form of table to provide the overview of the evolution history. The data extracted from a software system is represented in a tabular form in Microsoft Excel. This is highly scalable because a tabular form does not dependent on the amount of data. We could manage huge amount of data, generate graphs from the data, and have ability to do statistical analysis.

Many studies on software evolution emphasize the statistical changes of the software system by analyzing its evolution metrics [2, 3, 8, 9]. Observing the software evolution has been hampered by the huge data size and lack of standard visualization technique of software metrics. Moreover, most of existing software visualization techniques are not intuitive, therefore it requires education on the users’ side to grasp important information of software source code data. On the other side, software visualization techniques could be applied the preprocessing of the software metrics and visualize the metrics results. Therefore, the visualizations themselves should be simple and intuitive to guide a programmer to apply software metrics.

In this paper we present how to use software visualization technique as a preprocessing tool for software metrics. Evolution Track Table provides condensed information of software evolution, such as capturing the difference two consecutive versions on the evolution of target system. It can be used as a useful means to understand complex software evolution data [11], because visual displays allow the programmer to locate the target release of the evolution to analyze software metrics.

We concentrate on visualizing the evolution history of multiple releases of a software system. Despite several attempts to visualization of software evolution metrics [5, 10], it is difficult to visualize the history of software evolution. The problem of the Kiviat diagram and Polymetric visualization technique [5, 10] is suitable for visualizing the information (structure and metrics) of one release but not for visualizing the information of \( n \) releases. Using our visualization technique, we observe the evolution history of 22 releases of open source software, JFreeChart, in this case study. The system is investigated to guide a programmer in the software reading phases and to demonstrate the effectiveness of our approach as an example to justify the use of various functionalities of JamTool. We also introduce several ways to track and analyze the software structural changes from past releases.

2 Evolution Track Table

In this section we present the global visualization of software evolution using an Evolution Track Table, which is created to visualize the evolution of a software system. The
The evolution of classes of a software system can be visualized in an *Evolution Track Table* as shown in Figure 1. This table visualizes each class’s lifecycle for a software system in Microsoft Excel to achieve various data analysis, and it provides effective ways to analyze the evolution of the system. Each column of the table represents a version of the software, while each row represents a class name in each version. The numbers in the last column show how long a class survives during the entire system evolution, and the numbers in the last row indicate how many classes are involved in each version. To create the table, we collect and list all class names which are the member of the system at least once, and display ‘1’ or ‘0’ depending on whether or not a class is a member of a version of the system. In this way, the class name which lasts the longest in the evolution appears first.

### 2.1 Characteristics of Evolution Track Table

From an *Evolution Track Table*, we are able to obtain the following information regarding the evolution of a system.

- **Size of the system**

  We can find out how many classes are involved in system evolution. Figure 1 briefly shows an example of software evolution in an evolution track table. The summation of ‘1’s in each column is the number of classes existed in a particular version of the system. For instance, 14 classes in versions 1 and 2 and a total of 25 classes are involved in this evolution.

- **Lifetime of a class**

  The lifetime of a certain class can be recognized by the number in the last column. The summation of ‘1’s in each row represents the lifetime of a particular class of the system. For instance, in Figure 1, the system has evolved 15 times from version 1 to version 15, and classes c1, c8 and c19 were involved in 15, 7 and 10 times of evolutions, respectively.

- **Removed and added classes**

  The classes which have been removed or added in a certain version can be easily detected. The difference between two subsequent versions tells us that if a class is removed or added. If the number is changed from ‘1’ to ‘0’ between two consecutive versions, a class is removed, and if the number is changed from ‘0’ to ‘1’, a new class is added. For example, in Figure 1, classes c4, c5, c6, and c7 are removed in version 12, classes c8, c9, and c10 are removed in version 7, and their absence will leave ‘0’ on the table from that version on. Classes, c15, c16, c17, c18, and c19 are newly added to version 6. Therefore, in this example, a total of 13 classes are removed and a total of 11 new classes are added. By detecting the removed and added classes, we see very easily when/how much the system is changed.

- **Persistent classes**

  Persistent classes have the same lifetime as the whole system [4]. They have stayed from the beginning to the end. Those classes should be examined since they may be important in performing key functions of the system as being a part of the original system design. In Figure 1, three classes, c1, c2, and c3 are persistent classes.

- **Added persistent classes**

  Some important added classes have stayed until the last version. They might be created to upgrade or improve system as being a part of redesign of the system with some problematic classes removed. In Figure 1, six classes, c17, c18, c19, c20, c21, and c22 are added persistent classes.

### 3 Case Study

Understanding the evolution of an object-oriented system based on various versions of source code requires analyzing a vast amount of data since an object-oriented system has a complex structure consisting of classes, methods, attributes and different kinds of relationships between them rather than simply a set of classes. Using an *Evolution Track Table*, we provide an approach to understand such an evolution by detecting and visualizing the evolution pattern that characterizes classes. *Evolution Track Table* helps us understand an overall evolution of a
system, discover problematic parts with unusual measurement values, and visually get a quick understanding of the analyzed history.

The objectives of this case study are two fold. First, we analyze how a real world software system has evolved based on the information explained in Section 2. We present the visualization of Evolution Track Table, and explain how this table can be read, thus how an object-oriented system has evolved into its current state based on the source code. Secondly, based on the analysis in the first objective, we find out the releases which make big changes in a target system, and classes which have unusual evolution pattern, thus how Evolution Track Table can be used in software evolution research. We used 22 versions of JFreeChart as a target system for this study since JFreeChart is a long-term open source charting library with many releases.

3.1 Size of the System

From the Evolution Track Table along with 22 versions of JFreeChart, we collect the number of classes, the removed, and the added classes in each version as shown in Table 1. Based on this information, we are able to find out how big the system is and how many classes are involved in the system evolution.

<table>
<thead>
<tr>
<th>Version of JFreeChart</th>
<th>No. of Removed classes</th>
<th>No. of Added classes</th>
<th>Total no. of classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9.0</td>
<td></td>
<td></td>
<td>139</td>
</tr>
<tr>
<td>0.9.1</td>
<td>1</td>
<td>0</td>
<td>138</td>
</tr>
<tr>
<td>0.9.2</td>
<td>0</td>
<td>6</td>
<td>144</td>
</tr>
<tr>
<td>0.9.3</td>
<td>0</td>
<td>113</td>
<td>257</td>
</tr>
<tr>
<td>0.9.4</td>
<td>3</td>
<td>21</td>
<td>275</td>
</tr>
<tr>
<td>0.9.5</td>
<td>22</td>
<td>74</td>
<td>327</td>
</tr>
<tr>
<td>0.9.6</td>
<td>0</td>
<td>2</td>
<td>329</td>
</tr>
<tr>
<td>0.9.7</td>
<td>1</td>
<td>25</td>
<td>353</td>
</tr>
<tr>
<td>0.9.8</td>
<td>0</td>
<td>3</td>
<td>356</td>
</tr>
<tr>
<td>0.9.9</td>
<td>43</td>
<td>48</td>
<td>361</td>
</tr>
<tr>
<td>0.9.10</td>
<td>11</td>
<td>2</td>
<td>352</td>
</tr>
<tr>
<td>0.9.11</td>
<td>0</td>
<td>13</td>
<td>365</td>
</tr>
<tr>
<td>0.9.12</td>
<td>5</td>
<td>17</td>
<td>377</td>
</tr>
<tr>
<td>0.9.13</td>
<td>0</td>
<td>6</td>
<td>383</td>
</tr>
<tr>
<td>0.9.14</td>
<td>3</td>
<td>15</td>
<td>395</td>
</tr>
<tr>
<td>0.9.15</td>
<td>0</td>
<td>9</td>
<td>404</td>
</tr>
<tr>
<td>0.9.16</td>
<td>2</td>
<td>10</td>
<td>412</td>
</tr>
<tr>
<td>0.9.17</td>
<td>19</td>
<td>30</td>
<td>423</td>
</tr>
<tr>
<td>0.9.18</td>
<td>1</td>
<td>10</td>
<td>432</td>
</tr>
<tr>
<td>0.9.19</td>
<td>9</td>
<td>24</td>
<td>447</td>
</tr>
<tr>
<td>0.9.20</td>
<td>0</td>
<td>1</td>
<td>448</td>
</tr>
<tr>
<td>0.9.21</td>
<td>3</td>
<td>15</td>
<td>460</td>
</tr>
<tr>
<td>Total</td>
<td>123</td>
<td>444</td>
<td></td>
</tr>
</tbody>
</table>

This system started with 139 classes at version 0.9.0 and ended with 460 classes at version 0.9.21, which means a 333% class growth. A total of 567 classes are involved in the evolution. The number of classes increases gradually and consistently as new versions evolve (Figure 2). During the evolution, 123 classes were removed while 444 classes were added, which is 3.6 times more than the removed. In most versions more classes were added than removed. Special attention can be given to versions 0.9.3, 0.9.5, 0.9.9, and 0.9.17 since 68% (84 out of 123) of the removed were removed and 60% (265 out of 444) of the added were added in those particular versions (Figure 3).

3.2 Removed Classes

From an Evolution Track Table, we can find what classes are removed from which version of the system. The removed classes can be detected by finding the differences between two subsequent versions from ‘1’ to ‘0’ as shown in Figure 4. In this way, we found that many classes are removed during the evolution (See Table 1). In particular, 22, 43, and 19 classes were removed in versions 0.9.5, 0.9.9, and 0.9.17, respectively. Over 60% (74 out of 123) of them were removed in those three versions. This might imply that in those versions the system was aggressively changed. Some classes, which were removed in previous version,
reappeared later, like classes CategoryToolTipGenerator and StandardCategoryToolTipGenerator. They were removed from the system in version 0.9.8, but came back in versions 0.9.18 and 0.9.19, respectively. Classes StandardXYZToolTipGenerator and XYZToolTipGenerator were removed in 0.9.16, came back in 0.9.19, and stayed until the last version of the system. These kinds of interesting changes can be detected by the Evolution Track Table.

![Figure 4. Removed Classes](image)

![Figure 5. Added Classes](image)

![Figure 6. Persistent Classes](image)

![Figure 7. Added Persistent Classes](image)

### 3.3 Added Classes

The added classes can be detected by finding the differences between two subsequent versions from ‘0’ to ‘1’ as shown in Figure 5. In this way, we find how many classes were newly added into which version of the system during the evolution (See Table 1). In the case of the target system, many classes were added at almost every version. In particular, there were 113, 74, 48, and 30 classes added to 0.9.3, 0.9.5, 0.9.9, and 0.9.17, respectively. It is 60% (265 out 444) of the added classes. Some added classes like Pie3DPlot and HorizontalMarkerAxisBand were removed after staying for several versions. From the results of the removed and added classes, we found that this system had made huge changes in versions 0.9.3, 0.9.5, 0.9.9, and 0.9.17.

These versions may need to be specifically investigated. In particular, it is clear that noticeable system change was made in version 3 having 113 (25%) new classes added into that version.
3.4 Persistent Classes

Persistent classes have survived through the entire life of a software system. They can be easily detected by looking at ‘1’ at all versions and the total number of versions in the last column. As shown in Figure 6 they have ‘1’s for all versions and ‘22’ in the last column, which is the number of versions of the target system from 0.9.0 to 0.9.21. We found out that 84 out of the 138 classes in the first version have survived through the entire life of the target system, which is about 61% of the original design classes. From this result, we see that 54 classes (39%) of the original were removed during the evolution.

3.5 Added Persistent Classes

Many classes added in the middle of the evolution have stayed until the last version of the system. We call them ‘added persistent classes’. Figure 7 shows the examples of added persistent classes, and they were added in different versions when the system was changed from one state to another. Table 2 displays the number of added persistent classes and their survival rate in each version.

Table 2: Number of added persistent classes

<table>
<thead>
<tr>
<th>Version of JFreeChart</th>
<th>No. of added classes</th>
<th>No. of added persistent classes</th>
<th>Survival rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9.2</td>
<td>6</td>
<td>3</td>
<td>50%</td>
</tr>
<tr>
<td>0.9.3</td>
<td>113</td>
<td>89</td>
<td>78.76%</td>
</tr>
<tr>
<td>0.9.4</td>
<td>21</td>
<td>19</td>
<td>90.48%</td>
</tr>
<tr>
<td>0.9.5</td>
<td>74</td>
<td>61</td>
<td>82.43%</td>
</tr>
<tr>
<td>0.9.6</td>
<td>2</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>0.9.7</td>
<td>25</td>
<td>23</td>
<td>92%</td>
</tr>
<tr>
<td>0.9.8</td>
<td>3</td>
<td>3</td>
<td>100%</td>
</tr>
<tr>
<td>0.9.9</td>
<td>48</td>
<td>35</td>
<td>72.92%</td>
</tr>
<tr>
<td>0.9.10</td>
<td>2</td>
<td>2</td>
<td>100%</td>
</tr>
<tr>
<td>0.9.11</td>
<td>13</td>
<td>10</td>
<td>76.92%</td>
</tr>
<tr>
<td>0.9.12</td>
<td>17</td>
<td>17</td>
<td>100%</td>
</tr>
<tr>
<td>0.9.13</td>
<td>6</td>
<td>6</td>
<td>100%</td>
</tr>
<tr>
<td>0.9.14</td>
<td>15</td>
<td>15</td>
<td>100%</td>
</tr>
<tr>
<td>0.9.15</td>
<td>9</td>
<td>9</td>
<td>100%</td>
</tr>
<tr>
<td>0.9.16</td>
<td>10</td>
<td>9</td>
<td>90%</td>
</tr>
<tr>
<td>0.9.17</td>
<td>30</td>
<td>24</td>
<td>80%</td>
</tr>
<tr>
<td>0.9.18</td>
<td>10</td>
<td>5</td>
<td>50%</td>
</tr>
<tr>
<td>0.9.19</td>
<td>24</td>
<td>19</td>
<td>79.17%</td>
</tr>
<tr>
<td>0.9.20</td>
<td>1</td>
<td>1</td>
<td>100%</td>
</tr>
<tr>
<td>0.9.21</td>
<td>15</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>429 (444-15)</td>
<td>349</td>
<td>81.35%</td>
</tr>
</tbody>
</table>

3.6 Lifetime of a class

The number in the last column of Evolution Track Table explains the lifetime of a class from a version to another version until the class is removed for whatever reason. The persistent classes, which have the longest lifetime (Figure 6), might be very important in performing key functions of the system. Some added persistent classes appeared in the beginning of the evolution (Figure 7) have similarities. Some classes like SortOrder and HorizontalCategoryAxid3d in Figure 5 exhibit short lifetime. The system may not need those classes any longer due to user’s new requirement or system refactoring/redesigning.

3.7 Summary

From the Evolution Track Table of JFreeChart, we summarize the following findings:

- Started with 139 classes in version 0.9.0
- Ended with 460 classes (333% growth) in version 0.9.21
- 569 classes were involved in whole system evolution
- 123 classes were removed during the evolution
- 444 classes were added during the evolution
- 84 (60%) out of the 139 original classes have stayed until the last version
- 349 (81%) out of the 429 (444 added classes – 15 classes in the last version 0.9.21) added classes have stayed until the last version
- Big changes occurred in versions 0.9.3, 0.9.5, 0.9.9, and 0.9.17 in terms of removed and added classes.
4 Conclusions

In this paper, we have presented a tabular visualization technique, Evolution Track Table, to give an overview of software evolution and to capture the critical change of the software evolution history. We have used our methodology by applying on the 22 releases of the JFreeChart and have been able to understand diverse evolution patterns of the target system. Based on the findings from the target software, in this case study, we conclude that the Evolution Track Table can be used to categorize the evolution of classes, to discover the releases changed dynamically, and to identify unusual evolution pattern of classes.

We found the groups of persistent, removed, added, and added persistent classes from the evolution track table of JFreeChart. This categorization methodology enables us to quickly capture, understand and characterize the evolution patterns of classes of the target system. We found that several releases of the target system had big changes between two consecutive versions. This finding is very important to help generalize an overview of the problems presented during the evolution, and analyze the system by applying software metrics from JamTool in our future research. It is also useful to guide programmers to apply software metrics for further analysis since the detection of several places where in-depth examinations are needed, as well as the capturing the stable and volatile areas in the system. We also discovered the unusual patterns of evolution of the classes. Some classes had stayed unusually for only one, two, or several versions. Such dynamic classes need to be analyzed to understand the architecture of the system.

In addition, our Evolution Track Table approach is especially useful in the software metrics process. We believe it can be preprocessing to locate the target releases where we need to apply metrics to analyze the evolution detail, because our approach can point out where these metrics can be used.

5 References


[10] Pinzger, Martin, Gall, Harald, Fischer, Michael and Lanza, Michele, “Visualizing multiple evolution metrics”, In ACM Symposium on Software Visualization, 2005

Formal Definition of Feature Models to Support Software Product Line Evolutions

Huilin Ye¹ and Wendy Zhang²

¹School of Electrical Engineering and Computer Science, University of Newcastle, Callaghan, NSW 2308, Australia
Huilin.Ye@newcastle.edu.au

²Computer Science & Industrial Technology Department, Southeastern Louisiana University, Hammond, LA 70402, USA
wzhang@selu.edu

Abstract - Feature models have been widely used in software product line based software engineering. The dependencies between the variants and variation points in a feature model have very strong implications on the product configurations. Usually these dependencies are represented informally and incomplete in existing feature modelling approaches. In this work we first further explore the complex dependencies existing in software product lines. And then we propose a formal specification using Z notation to specify the dependencies in a product line. The specification formally defines software product lines and specifies complex dependency constrains contained in product lines. A set of operation schemas that support product line evolutions have been developed. With these operation schemas the invariants defined in the formal specification of product lines can be ensured when new features and dependencies are added into or removed from the product line. As Z specifications provide proof mechanism to validate the formal model and natural transition from a specification to an implementation a reasoning mechanism and a feature modelling tool can be developed in future.

Keywords: product line evolution, feature modelling, feature dependency, formal specification, Z notation.

1 Introduction

Feature models are suggested as a useful abstraction to represent variability in software product lines [1]. Features are prominent and distinctive system requirements or characteristics that are visible to various stakeholders in a product family [2]. Feature oriented modelling approaches have been widely used in software product line engineering. Most of the approaches use diagrams to represent features and their relationships. Although the diagrams provide intuitive pictures of feature models there is not sufficient accurate definitions of the concepts used in feature modelling. This brings ambiguous semantics of feature models. Formal specifications are believed an effective means that provides a theoretical foundation for the principles of software engineering in general. Some research works that applied formalism to feature modelling have been reported [3-6]. But we think that the following issues of the formalisation have not been well addressed in the previous works.

• Feature dependency modelling: the lack of complete and accurate of formal specifications of feature dependencies has great implication on member product configurations in software product lines. Dependencies in feature models not only exist between features but also exist between features and variation points and between variation points. Understanding these different types of dependencies will help configure valid member products.

• Support product line evolutions: Software product lines will constantly evolve along the time. It must be guaranteed the invariants of a product line remain unchanged when new features and dependencies are added in or existing features are removed from the product line.

• Feature modelling tools are necessary for product line engineering. Connection from the formal specification to the implementation of the modelling tools should be established. Formal approaches distant to tool implementation are not practicable and may not be acceptable by the software industry.

This work addressed the above issues. Thus it contains several contributions. First, dependency relationships in feature models will be further explored in addition to the ones reported in the literature. Then a formal definition of software product line will be presented. A set of operation schemas that support product line evolutions have been developed. With these operation schemas the invariants defined in the formal specification of product lines can be ensured when features and dependencies are added into or removed from the product line. The remainder of the paper will be organised as follows. Section 2 reviews the relevant background of feature modelling and further explores feature dependency relationships in feature models. Section 3 presents a formal definition of product line and a set of operation schemas.
supporting product line evolution. Section 4 concludes the paper.

2 Background of Feature Modelling

2.1 Review of Feature Models

Features in a product line are classified as mandatory features and variable features to represent commonality and variability of the member products respectively. An important concept used in feature modelling is variation point. Currently existing feature modelling approaches usually use a tree structure to organize features. If a parent node of the tree has one or more variable child features, called variants, the parent feature together with variable child features is defined as a variation point [7]. The configuration of a member product in a product line will go through the feature tree to include all the mandatory features and select some of the variable features at each variation point. Variable features include alternative features, multiple alternative features, optional alternative features, optional multiple alternative features. Instead of using different symbols to represent these different types of variable features multiplicity is introduced to each variation point to differentiate the four kinds of variable features. Hierarchical feature relationships and different variability types are represented in Table 1 using extended UML notations. The semantics of the notations are also explained in the table.

Figure 1 shows a feature model for a Car product line [8]. This simplified model is used to demonstrate the abovementioned concepts rather than a real model for a car product line. In this feature model Car is composed by variation points Control, Ordinary Accessories, Luxury accessories, Secure device, Quality attributes and mandatory feature Engine. The otherwise features are variable features that can be identified by the attached stereotype <<variant>>. Manual and Automatic are the specialised features of Control. Quality attributes consists of Security and Reliability. Control is composed by Manual and Automatic. Its multiplicity is 1, which specifies that its two variants are alternative variants, i.e. either Manual or Automatic can be chosen for a product configuration. Ordinary accessories has two optional variants represented by the multiplicity 0..2 attached to it. A car can have no such accessories, or can have either fan or power steering, or can have both. Quality attributes has multiple alternative variants represented by the multiplicity 1..2 attached to it. One or two quality attributes may be chosen when configuring a member product.

<table>
<thead>
<tr>
<th>Relationships</th>
<th>Notations</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composition</td>
<td>Feature A</td>
<td>Feature A consists of Feature A1, A2, and A3</td>
</tr>
<tr>
<td></td>
<td>Feature A1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Feature A2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Feature A3</td>
<td></td>
</tr>
<tr>
<td>Generalisation/</td>
<td>Feature A</td>
<td>Feature A is a generalised feature of Feature A1, A2, and A3</td>
</tr>
<tr>
<td>Specialisation</td>
<td>Feature A1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Feature A2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Feature A3</td>
<td></td>
</tr>
<tr>
<td>Variation point</td>
<td>Feature A</td>
<td>One and only one feature can be chosen from {A1, A2, A3} at this variation</td>
</tr>
<tr>
<td>(Alternative)</td>
<td></td>
<td>point.</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;variant&gt;&gt;Feature A1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;variant&gt;&gt;Feature A2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;variant&gt;&gt;Feature A3</td>
<td></td>
</tr>
<tr>
<td>Variation point</td>
<td>Feature A</td>
<td>One or more features can be chosen from {A1, A2, A3} at this variation</td>
</tr>
<tr>
<td>(Multiple alternative)</td>
<td></td>
<td>point.</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;variant&gt;&gt;Feature A1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;variant&gt;&gt;Feature A2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;variant&gt;&gt;Feature A3</td>
<td></td>
</tr>
<tr>
<td>Variation point</td>
<td>Feature A</td>
<td>No feature or at most one feature can be chosen from {A1, A2, A3} at this</td>
</tr>
<tr>
<td>(Optional alternative)</td>
<td></td>
<td>variation point.</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;variant&gt;&gt;Feature A1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;variant&gt;&gt;Feature A2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;variant&gt;&gt;Feature A3</td>
<td></td>
</tr>
<tr>
<td>Variation point</td>
<td>Feature A</td>
<td>No feature or more features can be chosen from {A1, A2, A3} at this variation point.</td>
</tr>
<tr>
<td>(Optional multiple alternative)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2.2 Dependencies in Feature Models

Dependencies in a feature model specify the constraints on the selection of variable features when configuring member products. There are three kinds of dependencies: dependencies between two variable features, dependencies between a variable feature and a variation point, and dependencies between two variation points.

2.2.1. Dependencies between Variable Features

Three types of dependency relationships have been identified as follows:

1. Requires: If a feature requires, or uses, another feature to fulfill its task, there is a Requires relationship between the two features.

2. Excludes: If a feature conflicts with another feature, they cannot be chosen for the same product configuration, i.e. they mutually exclude each other.

3. Impacts: When a feature is selected for a certain configuration and the selection will have impact on another feature, it is called Impacts relationship between the features.

Feature dependency relationships are non-hierarchical. We use UML stereo types to represent different types of dependencies (see Table 2). The dependencies identified for a product line must be validated. For a complex product line involving a large number of features, some conflicting dependencies may be recorded without awareness of their existence [9]. The validation of dependencies is intended to discover these conflicting dependencies. The following rules are defined for the validation:

- Excludes relationship must be mutually exclusive.
- Requires and Excludes cannot be occur between the same pair of features.

There is a bi-directional Excludes relationship between the two features.

Table 2. Feature dependency relationships and notations.

<table>
<thead>
<tr>
<th>Dependency type</th>
<th>Notations</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requires</td>
<td><img src="image1.png" alt="Diagram" /></td>
<td>Feature A requires Feature B.</td>
</tr>
<tr>
<td>Excludes</td>
<td><img src="image2.png" alt="Diagram" /></td>
<td>Feature A excludes Feature B and Feature B excludes Feature A.</td>
</tr>
<tr>
<td>Impacts</td>
<td><img src="image3.png" alt="Diagram" /></td>
<td>Selection of Feature A will have impact on Feature B.</td>
</tr>
</tbody>
</table>
2.2.2. Dependencies between Variation Points

There are two types of variation points, mandatory and optional, that can be distinguished by the multiplicity associated with the variation points. A multiplicity of 0..* or 0..1 means optional variation point while the otherwise multiplicity types represent mandatory variation points. For a mandatory variation point at least one variant must be selected. For an optional variation point the variants associated with the variation point may or may not be selected. Selecting or not selecting variants from an optional variation point will generally be based on a certain product configuration requirement. However, dependencies between variation points may constrain this selection. Selection of some variants at one variation point may cause dependencies to the other variation points in a product line [10]. For example, selection of some variants from one variation point may require or exclude the selection of variants from another variation point irrespective of which variant is selected. If both variation points are mandatory variation points Excludes dependency is not permitted and Requires dependency has already supported. The only case needed to be considered is the dependencies between optional variation points. Assume that both variation point A and B are optional variation points. If A requires B when some variants are selected at A some variants at B must also be selected irrespective of which variant is selected from both variation points. If A excludes B when some variants are selected at A no variant at B can be selected or when some variants are selected at B no variant at A can be selected irrespective of which variant is selected from both variation points. In the car product line as depicted in Figure 1, both Ordinary accessories and Luxury accessories are optional variation points. There is an Excludes dependency between the two variation points. If Fan or/and Power steering is/are selected at Ordinary accessories no variants can be selected at Luxury accessories or if Air condition or/and Telescoping steering is/are selected no variants can be selected at Ordinary accessories.

2.2.3. Dependencies between a Variable Feature and a Variation Point

The selection of a variant from one variation point may require or exclude the selection of variants from another variation point without constraining on the selection of the variants. If the required or excluded variation point is a mandatory variation point it will not be a problem as Excludes dependency is not permitted and Requires dependency has already supported. However, if there is a Requires dependency between a variant and an optional variation point the selection of the variant will force the optional variation point to become a mandatory variation point, i.e. some variants must be selected from this variation point. If there is an Excludes dependency between a variant and an optional variation point the selection of the variant will prohibit any variants being selected from the optional variation point. In the car product line as depicted in Figure 1, there is a Requires dependency from variable feature security and the optional variation point secure device. If security is selected, air bag must be selected at the optional variation point secure device.

3 Formal Specification for Product Line Evolution

Figure 2 shows an updated version of a formal definition of product line reported in [11]. A product line consists of a set of features, variation points, and three kinds of dependencies. The relation features maps the features existing in a product line to the two different types (mandatory and variable). The variationpoints specifies a collection of variation points. There is an Excludes dependency between the two variation points. If Fan or/and Power steering is/are selected no variants can be selected at Luxury accessories or if Air condition or/and Telescoping steering is/are selected no variants can be selected at Ordinary accessories.

```
ProductLine
features: FeatureType -> FeatureInstance
variationpoints: VPInstance

dependencies1: DependentType -> (FeatureInstance X FeatureInstance)
dependencies2: DependentType -> (FeatureInstance X VPInstance)
dependencies3: DependentType -> (VPInstance X VPInstance)

∀f1, f2: FeatureType | f1, f2 ∈ dom(features) ∧ f1 ≠ f2 ·
features f1 ∧ features f2 = φ

∀vpins: VPInstance | vpins ∈ variationpoints ·
vpins.parent ∈ ran(features) ∧ vpins.variants ∈ ran(features)

∀d1, d2: DependentType | d1 = Requires ∧ d2 = Excludes ·
(dependencies1 d1 ∧ dependencies1 d2 = φ ∧
dependencies2 d1 ∧ dependencies2 d2 = φ ∧
dependencies3 d1 ∧ dependencies3 d2 = φ)
```

Figure 2. ProductLine schema
When a product line is defined it will be constantly evolved. New features and dependencies will be introduced to and some old features and dependencies may be removed from the product line. The AddFeature operation schema (Figure 3) is defined for adding a new feature into a product line. If a feature being added, NewFeature?, is a variable feature it may be attached to an existing variation point or a new variation point will be created by the addition. If it is added to an existing variation point the multiplicity may need to be updated by the addition. If this addition will create a new variation point, a multiplicity must be specified. In either case, an input variable Multiplicity? is needed to specify the multiplicity.

The predicate part of the AddFeature asserts the following:

- If NewFeature? is not an existing feature in the product line, add it in.
- If NewFeature? is a variable feature and its parent is the parent of an existing variation point, include it to the variants of this variation point.
- If NewFeature? is a variable feature and the parent of NewFeature? is not the parent of any existing variation points the addition of NewFeature? will create a new variation point. The id of the new created variation point will be the number of existing variation points plus one. The parent of the new created variation point is the parent of NewFeature?. Its multiplicity is Multiplicity?. Its variant is NewFeature?. The AddFeature uses a µ expression. The semantics of a µ expression is the same as that of a set comprehension except that a µ expression returns only one value for the set.

![Figure 3. AddFeature schema.](image)

![Figure 4. AddDependencies schema.](image)
The addition of new features may introduce new dependencies into product lines. New dependencies can also be identified among existing features and variation points. The \textit{AddDependencies} schema (Figure 4) is used to add dependencies to product lines. The relation \texttt{newDependencies1?} maps the new dependencies to be added between variable features to different dependency types. The relation \texttt{newDependencies2?} maps the new dependencies to be added between variable features and variation points to different dependency types. The relation \texttt{newDependencies3?} maps the new dependencies to be added between variation points to different dependency types. The relation \texttt{newImptMesgs?} maps a pair of features with \textit{Impacts} relationship to an impact message.

The predicate part of the \textit{AddDependencies} schema asserts that the new added set of dependencies should not conflict with the existing dependencies, i.e., a \textit{Requires} and \textit{Excludes} dependency should not exist between a pair of features, or between a feature and a variation points, or between a pair of variation points, at the same time in a product line.

A function named \textit{RemoveFeature} (Figure 5) is used to remove a feature from a product line. The feature to be removed is given as a parameter of the function. The predicate part of the \textit{RemoveFeature} asserts the following:

- If the deleted feature is a variant of a variation point, remove it from the set of variants of the variation point.
- Remove all the dependencies involving the deleted feature, including dependencies with other features and dependencies with optional variation points.
- \textit{RemoveFeature} is a recursive function. If a deleted feature is a parental feature all of its child features should be recursively deleted from the product line.

\textbf{4 Conclusions}

The formal specification presented in this paper provides a generalised approach to product line engineering and has several advantages. It supports product line evolution. The defined operation schemas will detect conflicting dependencies introduced by adding new features into or by removing old features from a product line. It will be difficult for the logic expression based approaches to support product evolution as there is no facility to check any inconsistent dependency after a product line is updated. Based on the formal specification a set of theorems can be developed and proved. The reasoning mechanism and proofs of the formal specification will be our future work. Based on the formal specification a modelling tool for product line engineering can be easily developed as \textit{Z} specifications provide natural transition from the specifications to an implementation. It will be much easier to implement a modelling tool based on the \textit{Z} specification than any of the formal models using logic expressions. A modelling tool for product line engineering and product configuration will be developed in the near future.
5 References


Aspect-oriented Approaches to Model Driven Engineering

Devon M. Simmonds
Computer Science Department
University of North Carolina at Wilmington
Wilmington, NC 28403
Phone: (910) 962-3819, FAX: (910) 962-7457
simmonsd@uncw.edu

Abstract

In aspect-oriented model driven engineering (AOMDE), a software design model consists of a primary model that describes the business logic of the application and a set of aspect models each of which describes a crosscutting software feature. The complete design is realized by composing the primary model with the aspect models. A variety of AOMDE approaches are possible depending on how the principal tasks of the AOMDE process are interleaved. This paper presents and compares two distinct AOMDE approaches.

Keywords: model-driven engineering, aspect-oriented software development, software engineering.

1. Introduction

Model driven engineering (MDE) [4, 11, 12, 13, 14, 16, 17] advocates the use of models as first class entities in software development. Models facilitate the understanding of complex systems by supporting the representation of systems at a high level of abstraction.

In order to realize the potential benefits of models, MDE approaches must facilitate the specification and reuse of models. However, reusing software models is difficult in the presence of crosscutting software features. Aspect-oriented technologies can be used to support MDE by facilitating the separation of platform-specific designs from business functionality designs. In aspect-oriented software development (AOSD) [1, 2, 7, 9, 18] crosscutting functionality are those that are spread across and tangled with other design elements. In AOSD crosscutting functionality are modularized in aspects and a complete software design consists of a platform independent primary model that describe the business logic of the application and a set of aspect models that describes crosscutting software features.

To support an aspect-oriented approach to model driven engineering (AOMDE), lifecycle process models delineating the principal tasks undertaken in AOMDE are needed, along with the identification of temporal and logistic dependencies between tasks. AOMDE approaches may be classified into two broad groups [6]. In one approach called the Generate Then Weave approach, code is generated separately from primary model and aspect design models and a complete application is created by weaving aspect code and primary model code. In the second approach, a complete design model of the target application is created by composing aspect models with a primary model. Code is then generated from this composed design model. The second approach is called the Weave Then Generate approach. While these approaches were identified in our previous work [6], the approaches were not explored. This paper identifies several high-level AOMDE tasks and utilizes the tasks in describing and comparing the two approaches.

The rest of the paper is organized as follows. Section 2 gives an overview of the high-level AOMDE tasks. Section 3 describes AOMDE approaches while a comparison of approaches is presented in Section 4. Section 5 presents a discussion of the approaches and identifies outstanding issues. Conclusions and future work are presented in Section 6.

2. Principal AOMDE Tasks

In the presented AOMDE approaches, classical software development lifecycle activities such as analysis, design and implementation are reinterpreted in the context of AOSD, and several AOSD-specific activities are added.

1. Analysis. This includes need identification and scoping, analysis of the scoped problem, the specification functionality requirements, and the identification of
potential crosscutting functionality (so called early aspects).

2. Primary Model Design. The Requirements Specification resulting from analysis is used to create a platform-independent primary model as well as to identify additional crosscutting software features that may be modeled as aspects.

3. Primary Model Enhancement. In an AOMDE environment, a complete application design is obtained by composing a primary model with aspect models. For a successful composition, the primary model must contain structural and behavioral properties required by the target platform. These required properties differ from one platform to another. The Jini middleware, for example, requires the creation of a proxy and a remote interface for each service. CORBA on the other hand, requires the creation of an IDL file for each service. The process of customizing a primary model to include required platform-specific features is called enhancement and the generated artifact is called an Enhanced Primary Model. In MDE, a primary model may be subject to many transformations, depending on the development goals of software engineers. For example, one may wish to refactor a primary model to make it amenable to a particular architectural style. In this paper the term enhancement is used to refer to that class of transformations that make an artifact platform-specific.

4. Aspect Model Design. An aspect model is a modularization of a crosscutting software feature. During aspect model design, models are created that represent the crosscutting software features identified during analysis and design of the primary model. While it may be possible to identify some crosscutting features during analysis, a complete list of these features is only available after design of the primary model.

5. Aspect Model Enhancement. Aspect model enhancement involves transforming an aspect to make it platform specific. For example, transforming a platform independent 2-phase transaction model to make it CORBA-specific. Aspect model enhancement is required when an aspect is platform independent which should be typical in AOMDE.

6. Aspect Model Instantiation. Aspect enhancement involves adding application-specific information to the aspect to enable the composition of the aspects with the primary model of the application. The process of adding the application-specific information to an aspect is called binding or instantiation, and the resulting aspect is called a mapped of context-specific aspect. Examples of application-specific information include primary model attribute names and operations.

7. Weaving (i.e. Composition). A complete application model is generated by composing application-specific aspects with an enhanced primary model. Composition may occur at either the design level or the code level.

8. Code Generation involves generating application code from design models.

9. Testing, Deployment and Evolution. During these phases executables are generated and the application is tested, deployed and evolved. For the scope of this paper, these phases have been omitted since the salient features of the two approaches may be represented without these tasks.

3. MDE Approaches

Templates have been used for describing design patterns [5] and testing patterns [3]. Similarly, in this paper, a template is used for describing MDE approaches. The template contains the items listed below.

1. Name. The name of the approach is a word or phrase that is used for identifying the approach and for suggesting its general intent.

2. Intent. The intent of an approach is a statement that highlights the objectives and distinctive features of the approach.

3. Structure. Each description of a MDE approach includes a graphical representation of the approach, (its structure), along with a textual description of the interrelationship of its subparts. The high-level AOMDE tasks defined in section 2 are represented in each graphical representation.


Each AOMDE approach is described using an activity diagram where action nodes are classified into six partitions (See Figure 1 and Figure 2). The partitions provide a high-level description of the AOMDE approaches with the high-level tasks being distributed among the partitions as shown in the figures.

3.1. The Weave-Then-Generate (WTG) Approach

Name: Weave-Then-Generate.

Intent: Create an integrated design model and use the model as the basis for generating application code.
**Figure 1. The Weave-Then-Generate Approach.**

**Structure:** The overall structure of the *Weave-Then-Generate* approach is depicted in Figure 1. The figure illustrates that code is generated from a composed design model created during weaving. In this figure, weaving precedes code generation.

**Consequences:** This approach may be considered ideal in AOMDE because a composed design model that facilitates design-time model analyses and model execution is created. For example, the composed model can be used for trade-off and error analyses as well as for scenario-based 'what-if' analyses. It is through these types of analyses that many of the expected benefits of MDE will be realized.

When a design aspect is transformed into a code aspect, it may not be possible to convert the design aspect into a single corresponding code aspect. The ability to effect this transformation is constrained by the sophistication of the target aspect-oriented programming language. However, this approach is not constrained by this limitation since code is not generated for individual aspect models.

One potential weakness of this approach is a potentially large semantic gap between individual design models and code because a design aspect is translated directly into a code aspect and a primary model is also transformed directly into code. Until MDE become a mature discipline, one would expect the need for manual code evolution in many cases, and as such, this semantic gap may be important in such cases.

3.2. Generate-Then-Weave Approach

**Name:** Generate-Then-Weave (GTW).

**Intent:** Create a complete application by integrating aspect code and primary model code generated separately from aspect models and a primary model respectively.

**Structure:** The Generate-Then-Weave Approach is depicted in Figure 2. The structure differs from the Weave-Then-Generate (WTG) approach described in section 3.1 in two respects. First, there are two code generation phases, one for the primary model and one for the aspect models. Second, in this approach, code generation precedes weaving.

**Consequences:** The advantages of this approach include:

1. A potential reduction in the semantic gap between individual design models and code because a design aspect is translated directly into a code aspect and a primary model is also transformed directly into code. Until MDE become a mature discipline, one would expect the need for manual code evolution in many cases, and as such, this semantic gap may be important in such cases.

2. Improved traceability between code errors and design artifacts (enhanced PM, mapped aspects). When an error is detected in code, it may be easier to relate the error to the aspect models or the primary model since the code resulted directly from these models. This differs from the *Weave Then Generate* approach where code is generated from a complex integrated design model and therefore before an error can be traced to the mapped aspect of...
the enhanced primary model it may be necessary to first trace the error to a component in the integrated design model.

3. Scattering and tangling is completely eliminated at the design level.

This approach has several weaknesses. First, the approach is constrained by the sophistication of available aspect-oriented programming language. Two important questions in this regard are: (1) whether there are cases where a design aspect cannot be directly translated into a code aspect in the target language, and (2) how these cases can be identified. For example, inner classes have been a problem for AspectJ [15] aspect developers. Secondly, the absence of an integrated design model constraints the capacity for model analysis and model execution.

4. Comparison of Approaches

In this section a comparison of the AOMDE approaches is presented. The goal of the comparison is to provide guidance to developers as to when and how the approaches may be used.

4.1. Tool Requirements

The Weave-Then-Generate approach requires the following tools:

1. Model development and editing tools for creating and modifying design models.
2. Model transformation tools to support both vertical and horizontal transformation of models, including the compilation of models into code.
3. Model composition tools to be used to compose aspect models with a primary model. These tools will need to support a variety of composition strategies, for example, signature based composition [10].
4. Aspect model instantiation tools.
5. Model debugging tools to facilitate traceability of code-level errors to design artifacts.
6. Model execution tools to facilitate identification of faults before code is generated from the models.
7. Model analysis tools to support the identification and visualization of apparent and emergent software properties, both static properties and behavioral properties.

The tool set required by all the Generate-Then-Weave approach differs from that listed above in two general ways. First, model composition tools are not required since a composed design model is not created. Instead, code composition tools are used. Secondly, tools that can generate aspect oriented code from design models are needed.

4.2. Design and Code Reuse Potential

Reuse potential of both approaches may be examined from both code-centric and design-centric viewpoints. From the design viewpoint, the Weave-Then-Generate approach has more reuse potential because the composed application design generated during weaving may be used multiple times for model execution and model analysis purposes. Conversely, the Generate-Then-Weave approach is superior with respect to code reuse potential because (1) the code generated for the primary model may be reused with many relevant aspects for the target platform, and (2) the aspect code may be reused with many different applications.
4.3. Error Correction and Modifiability

The traceability of application errors to code and design artifacts will depend in part on the semantic gap between the composed application code and the more abstract code and design artifacts. Since the Generate-then-Weave approach generates a code artifact for each individual aspect model as well as for the primary model, it may be easier to trace code errors to one of the individual design artifacts. This contrasts with the Weave-Then-Generate approach where a composed design model is intermediate between the application code and the individual design models. On the other hand a composed design model provides an opportunity for the design-time observation and analysis of the semantic coupling [8] inherent in an aspect-oriented design. The approach selected for a specific project will depend on availability of tools and the purpose and intent of the design models. For example, the availability of tools that can execute designs and a need to perform scenario analyses may mitigate against using the second approach.

5. Discussion

The two approaches presented represent two distinct AOMDE lifecycle models. The models presented support the incremental development of applications with feedback loops. Incremental development of the primary model is facilitated through the \textit{morePM} guard. The \textit{morePM} condition is true whenever either primary model features for one or more requirements or one or more sub-systems remain to be designed. Aspects may also be developed iteratively. This is facilitated through the \textit{moreAspects} guard. This condition is true whenever one or more aspect models remain to be designed. Therefore weaving can proceed as long as there are at least one available primary model feature and one associated aspect model.

In an aspect-oriented environment, errors in design and code may originate from the primary model, from the aspect model, or as an emergent property of the application. In any of these cases modification will need to be made to the primary model, one or more aspect models or to both aspects and primary model. This is supported in the approaches through the use of special error conditions, for example \textit{createError} and \textit{enhanceError} guards.

MDE is sometimes conceived as a unified process where code is generated from models and evolution of applications is model-centric based on a forward-engineering-only process. However, the two approaches presented in this article suggest that a variety of options are available. For example, in the Generate-Then-Weave approach code is generated separately for each aspect model and for the primary model. As a result, an error in an aspect model would require modification of the aspect and regeneration of only the code for that aspect. In contrast, an error in an aspect in the Weave-Then-Generate approach may actually require modification to the aspect, recomposition with the primary model, and code regeneration using the composed model. The two approaches presented in this article suggest that the code artifacts present in MDE environments will differ depending on the MDE approach selected. This difference in the code is likely to impact the decision to develop application by a forward-engineering-only approach versus a combination of forward and reverse engineering.

In summary, an optimal program evolution strategy in a MDE environment may actually involve a discretionary mix of forward and reverse engineering. Forward engineering is the obvious choice for generating the initial application code. However, a number of potential factors may affect the choice of forward versus reverse engineering when the software is to be modified. These factors include: (1) the AOMDE approach used (Weave-Then-Generate, Generate-Then-Weave or an hybrid approach), the scope of the modifications, and (3) whether the modifications are being made to aspect models or primary model, and (4) the nature of the modifications needed - adaptive, corrective or perfective. Further research is needed to determine how these factors are likely to impact an AOMDE environment.

6. Conclusions and Future Work

A model-centric approach to software development is needed to leverage the benefits of models in software engineering. Aspect-oriented approaches to model driven engineering are needed to help manage the complexity of cross-cutting concerns in MDE. In this paper two aspect-oriented approaches to model driven engineering have been presented. The differences in the approaches suggest that MDE in practice may not need to be the cohesive unary process that is sometimes suggested. Instead, a variety of potential options are available. Further research is needed to explore the details of the two presented approaches. For example, the Generate-Then-Weave approach seems to be a family of approaches rather than a single approach. This can be observed by noting that it is possible to generate primary model code and aspect code before customization.

In the generate-Then-Weave approach, when a design aspect is converted to code, part of the aspect may have to be added to the primary model of the application depending on the sophistication of the target aspect-oriented programming language. Further research is needed to determine the constraints and limitations of mapping design aspects into code.
References


Validating Requirements and Design Parameters for Aspects

Deepak Dahiya                      Usha Batra                        Sudha Dahiya
Department of CSE & IT        Department of CSE & IT                NIC
ITM Gurgaon, India                ITM Gurgaon, India               Delhi, India
deeap.d@acm.org                   btrausha@rediffmail.com           sudhamuscat@yahoo.com

ABSTRACT

There is a need to study various approaches in the use of object-oriented design patterns and aspect oriented design approach in enterprise systems for architecture and its implementation. The development of aspect oriented requirements gathering approach, design notation and environment for development of enterprise systems needs to be further refined in the context of software applications and industry. This paper discusses the parameters involved in requirements and design stages for aspect oriented software development and derives the specific requirements for the AOSDDL (Aspect Oriented Software Development Design Language) design language architecture that is proposed within this work.

Keywords — aspects, AOSD, AOSDDL, Separation of Concerns

1 INTRODUCTION

Recently there has been growing interest in propagating the aspect paradigm to the activities in the earlier phases of the software development lifecycle. A number of approaches to aspect-oriented design have been proposed e.g. [1, 2]. One approach proposes an extension to UML to support aspects. Composition Patterns is another approach to handle crosscutting concerns at the design level [1, 2].

The Unified Modeling Language (UML) [3, 4] is an object-oriented design notation that provides basic building blocks to model software-intensive systems, such as abstractions that represent structure and behavior of a system, relationships that state how the abstractions relate to each other, and diagrams that show interesting excerpts of a set of abstractions and relationships. The most important characteristic of UML in respect to the issue tackled in this work are its extension mechanisms [5]. UML’s extension mechanisms provide standardized means to extend existing UML building blocks with new properties, called tagged values, or with new semantic, called constraints. Besides the alteration of existing building blocks, the UML may be extended with completely new building blocks that are derived from existing ones. The new building blocks, called stereotypes, have the same structure (attributes, associations, operations) as the base building block they are derived from. One such approach is referred to as the “aspect-oriented design model”, or AODM for short [5] that extends the Unified Modeling Language with the aspect-oriented design concepts as specified in AspectJ [6, 7, 8].

Design Patterns became popular after the “Gang of Four” book of the same name (Gamma et al, 1995) [9, 10, 11, 12, 13] showed how design patterns could be used in object-oriented software development. Design patterns are a method of encapsulating the knowledge of experienced software designers in a human readable and understandable form. They provide an effective means for describing key aspects of a successful solution to a design problem and the benefits and tradeoffs related to using that solution. Using design patterns help produce good design, which helps produce good software. The ability to work with design patterns in conjunction with Unified Modelling Language (UML) is a major benefit. UML is now a standard for OO modeling and is an industry standard now. Compatibility with UML makes design patterns more palatable for many programmers and designers as they are already familiar with UML. For the implementation of design patterns, the design policy to consider the patterns as concerns is important. At the same time, it is to be desired that we have effective languages and tools supporting the advanced separation of concerns [14, 15]. The new implementation technologies that support the
advanced separation of concerns such as Hyper/J [16, 17] and AspectJ [6, 7, 8] help with coding this kind of design.

Separation of concerns is a basic engineering principle that is also at the core of object-oriented analysis and design methods in the context of UML. Separation of concerns can provide many benefits: additive, rather than invasive, change; improved comprehension and reduction of complexity; adaptability, customizability, and reuse. With its nine views that can be thought of as projections of a whole multi-dimensional system onto separate plans, UML [4] provides an interesting separation of concerns called the 4+1 view model (Design view, Component view, Process view, Deployment view, plus Use Case view). In turn, each of these views has two dimensions, one static and one dynamic. Furthermore the designer can add non-functional information (e.g. persistency requirements) to a model by “stamping” model elements, for instance with design pattern occurrences [5], stereotypes or tag values. It is appealing to think of many concerns as being independent or “orthogonal”, but this is rarely a case in practice. It is essential to be able to support interacting concerns, while still achieving useful separation. An aspect-oriented approach to design can help to express these concerns explicitly. Frameworks that provide methodological support for building and manipulating UML models with aspects have been proposed. One such framework is the UMLAUT [18] (UML All pUrpose Transformer) framework which allows the engineer to program the “weaving” of the aspects at the level of the UML meta-model.

For aspect-oriented software development (AOSD) to live up to being a software engineering paradigm, there must be support for the separation of crosscutting concerns across the development lifecycle [1] including traceability from one lifecycle phase to another. Concerns that have a crosscutting impact on software (such as distribution, persistence, etc.) present well-documented difficulties for software development [19]. Since these difficulties are present throughout the development lifecycle, they must be addressed across its entirety. One such work done is the investigation of traceability between one particular AOSD design-level language, Theme/UML [1, 20] and one particular AOSD implementation-level language, AspectJ. This provides for a means to assess these languages and their incompatibilities, with a view towards eventually developing a standard design language for a broad range of AOSD approaches.

This paper discusses the requirements for aspect oriented design language in general and derives the specific requirements for the AOSDDL (Aspect Oriented Software Development Design Language) design language architecture [1] that is proposed within the broader area of this work.

It has become apparent that design language research deals largely with trade-offs. For example, many of the aspect oriented design systems [2] trade-off implementation dependency for wide tool support or limited support with general purpose flexibility.

Research into aspect oriented design languages so far has shown that no single solution will meet all possible requirements of aspect oriented software development, and thus, multiple systems for domains with different demands must be able to co-exist and interoperate. The challenge in designing aspect oriented solutions therefore is to draw the optimal line between trade-offs depending on the requirements [3] at hand. For this, it is crucial to understand fully the requirements of a given domain [4].

2 REQUIREMENTS

As with systems in any programming paradigm, aspect-oriented systems need to be designed with good software engineering practices in mind.

The analysis and design of a system are at least as important as the implementation itself, with many considering these phases to be more significant in their contribution to the success of a project as a whole.

In any development effort, it is helpful for a developer to be able to consider the structure of the final implementation at all stages of the software lifecycle, rather than having to make a mental leap to get from a particular way of
encoding design, to another way of coding the software. In other words, developers need to be able to easily map their designs to the code in order for the design to continue to make sense during the development lifecycle.

In addition to seamless traceability between the design and code, another consideration is the benefits of separating aspects in the design for the design’s own sake. Aspect-oriented design [21] has similar benefits for design artifacts as aspect-oriented code has for code artifacts. In the infancy of aspect-orientation, developers simply used object-oriented methods and languages (such as UML) for designing their aspects[22]. This proved difficult, as UML was not designed to provide constructs to describe aspects trying to design aspects using object-oriented modeling techniques proved as problematic as trying to implement aspects using objects.

Without the design constructs to separate crosscutting functionality [23], similar difficulties in modularizing the designs occur, with similar maintenance and evolution headaches. What is required is special support for designing aspects, as we will then be able to improve the design process, and provide better traceability to aspect-oriented code.

A similar set of problems arises when analyzing requirements documentation [24] to try to arrive at how to design a system. Approaches for decomposing requirements from an object-oriented perspective simply don’t go far enough when trying to plan for aspect-orientation. Heuristics and tools to support such an examination will be helpful to the developer.

The work aims to support for how to both identify aspects in a set of requirements, and how to model them in UML style designs. The methodology we use is an aspect based approach to analysis and design.

3 Separation of Concerns

Separation of concerns (SOC) [25] is a long-established principle in software engineering. It has received widespread attention in modern programming languages, with constructs such as modules, packages, classes, and interfaces, which support properties such as abstraction, encapsulation, and information hiding. SOC has also received attention in software architecture and design, with techniques such as composition filters [26] and design patterns [27].

While advances in all of these areas have had significant benefits, problems related to inadequate separation of concerns remain. This has led to recent work on “advanced separation of concerns” (ASOC), including subject-oriented programming and design, aspect-oriented programming, and multidimensional separation of concerns [28]. These bring a number of innovative ideas to programming in particular and to software development in general, which are now beginning to mature and coalesce under the heading of aspect-oriented software development (AOSD).

Although ASOC has been emphasized in recent work, concerns themselves have remained something of second-class citizens. Current ASOC tools provide only limited support for explicit concern modeling, representations of concerns tend to be tied to particular tools or artifacts, and concern modeling usually occurs just in the context of a particular type of development activity such as coding or design. A global perspective on concerns, that spans the life cycle and is independent of particular development tools or artifacts, has been lacking.

During software development, concerns arise at all stages of the life cycle, from requirements specification through design, coding and testing to maintenance and evolution. Concerns also span multiple phases of the life cycle, relate to multiple instances and types of artifacts, and crosscut phases and artifacts in different ways. Finally, concerns are dynamic and relative, that is, that the concerns relevant to a particular software unit will change over time and that they also depend on the perspective or purpose of the user or stakeholder who considers the software.

4 Architectural Design Enforcement

Two major approaches have been suggested to bridge the gap between high-level design models and the system itself: user invoked i.e. the use of codified design principles must be supplemented by checks to ensure that the actual implementation adheres to its design constraints and guidelines versus the environment invoked
i.e. the gap between the architectural model and the implemented system can be bridged effectively if the model is not just stated, but is enforced\[29\]. These principles cannot be localized in a single module, they must be observed everywhere in the system, which means that they crosscut the system's architecture.

Aspect Oriented Programming (AOP) is a programming technique for modularizing concerns that crosscut the basic functionality of systems. Aspects provide a means to clearly capture design decisions.

In other words, we are addressing the possibility of using AOP in general, and AspectJ\[8\] in particular, in order to solve the problem of design enforcement.

5 Avoiding Design Incompatibility

The architecture design is an important step within the software development lifecycle. OO design has proved its strength when it comes to modeling common behavior. However, OO design does not adequately address design artifacts that crosscut an architecture\[30\]. They cannot be encapsulated by single components or packages and are typically spread across several of them and therefore also make design hard to understand and maintain. Crosscutting concerns are present during all phases of a software development lifecycle, leading to code tangling or code scattering during the implementation phase and graphical tangling during the design phase. AOSD is still lacking standardized concepts at the design phase that would foster the specification of crosscutting concerns at the high level architecture and low level design. Development of large software systems follows processes that all include activities like requirements engineering, analysis, design and implementations. Following a design methodology like OOD, and focusing on AOP at coding level causes a shift of paradigms between OO design and AO code. This leads to inconsistencies between design and implementation as the AO paradigm\[31\], \[32\] is not seamlessly supported during the early stages of the development lifecycle. To avoid the divergence of design models and code, crosscutting concerns must be identified at the requirements and architecture level and carried forward in the implementation phase. Concepts are needed for a seamless integration of AO design and implementation and will be a first step towards an integrated AO development process. To make AOSD more widely accepted, the different phases of an AOSD lifecycle have to be integrated more smoothly by supporting the AO paradigm in every phase. Our subsequent work includes a design notation for validation of AO models. Supporting design models and their transition to concrete implementations makes AOSD more usable, more efficient and more accepted among software engineers.

When analyzing OO design, one can see that OO modeling tries to adopt many of the OO programming features for design and analysis. Classes, their structures, and their relationships are identified and generalization and aggregation hierarchies are built. OO design techniques are not sufficient when focusing on the AO paradigm as crosscutting concerns also make design tangled and therefore hard to understand and maintain. When developing an AO modeling approach, the following requirements are obvious:

- A sufficient notation
- Design modeling
- Design notations should support modeling
- Models should be easy to read and offer a clear separation
- A direct mapping between the notation and supported implementation
- The notation should be applicable in real-world development projects

This work can be seen as a step towards a standardized way to capture aspects at the design phase of an AO development process. Existing approaches and prototypes are well aware of the fact that aspect-oriented modeling is a critical part of AOSD. Obviously, to obtain an AO development lifecycle, the gap between AO requirements engineering\[33\] and AOP has to be filled. This work makes a contribution to the problem of bridging this gap.
6 Business Model

A key factor as to whether or not aspect oriented software development will become a real success depends on a sound business model for the new technology. Without a promising business perspective, software tools vendor and middleware providers will have no urge to move towards an aspect oriented software environment in the first place. However, some vendors (for example Sun or Oracle) have started supporting aspect oriented tools by providing them as optional plug-in tools to their existing available tools and technologies available in the market.

From this, we can conclude that a wider deployment of aspect tools and technologies relies much on a promising business model for current stakeholders and additional business opportunities for new players in the area.

7 AOSDDDL Requirements

The primary goal of the work presented here is to develop a generic aspect oriented design language that can be used to design and build aspect oriented applications. However, since the prototyping of such a system is impeded by financial resources and time constraints in the context of the research project, the objective is rather to use open source tools as a base platform and to focus on the development of a minimal, but extensible design language that may serve as a flexible platform for future aspect oriented research.

8 Design and Implementation

Subsequent work encompasses presenting the design and implementation of the aspect oriented design language. This work is highlighted elsewhere [34, 35, 36] and enhancements in the work will be presented in a future paper. Both show how Aspect Oriented Software Development fulfills the requirements defined in this paper by design and implementation. Finally, we also evaluate to what extent AOSDDL has succeeded in meeting these requirements.

9 Contributions

Here we summarize the main contributions and achievements of the research carried out as part of this work.

The overall goal of this work, namely to design a aspect oriented design language that enables flexible extensibility of requirements and design functionality, has been successfully fulfilled in the form of AOSDDL structure. The validation of the architectural design with respect to its feasibility and practicality has been accomplished through prototype implementations of the AOSDDL architecture.

- Natural Extension to UML
- CASE Tool Support
- Extension of Architectural framework for design constructs
- Enforcing Architectural Regularities
- Commercial Viability
- Implementation Support
- Software Development

10 Future Scope of Work

Besides the ongoing development efforts to complete the AOSDDL prototype implementation, further work in this area focuses on using and extending the AOSDDL notation architecture and prototype platform in order to build and experiment with design language specifications.

The code generators, tool integration and notation deployment and are few examples of ongoing research that take advantage of the AOSDDL architecture and platform.

11 Conclusion

This paper has examined the requirements for aspect oriented software development and design language in particular.

It discussed the general requirements for aspect oriented systems. These requirements have been derived from related work and acknowledged publications in the field. They summarize the general requirements of today’s aspect oriented systems.

These requirements have been derived from
related work and acknowledged publications in the field. General factors, for example the commercial aspects such as the deployment of aspect oriented technologies are also taken into consideration.

There are still many issues to be solved until efficient AO development support comparable to current OO support is established. There is still a lot of challenging research to be done in the future until the paradigm is widely accepted and developers are aware of the benefits AOSD offers.

REFERENCES:


[27] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns-- Elements of Reusable Object-Oriented Software. Reading, Mass.: Addison-Wesley, 1995.


Knowledge Based Reverse Engineering Process Framework

Nasir ali
Computer science dept. The University of Lahore
1-KM Raiwind Road Lahore, Pakistan

Abstract - The magnitude and popularity of software reengineering increase as more and more successful computing systems become legacy systems. To perform reengineering on the legacy systems first we need to perform the reverse engineering task on the legacy system. Several frameworks have been proposed for reverse engineering, yet no conclusions have been reached about which are more appropriate. This is partly because each framework may work well for different types of scrutiny, but it hasn’t been shown if any are sufficient for all types of scrutiny. This paper presents a knowledge based framework for reverse engineering to define analysis techniques to address this problem. All reverse engineering process uses three layers (extraction, abstraction and presentation), while this framework introduces a new layer that is knowledge layer which works before extraction layer.

Keywords: Reverse Engineering, Artifacts Recovery, Reengineering, and Software maintenance.

1 Introduction

The large systems which have been developed and maintained by different people are called legacy systems. And these systems have been developed on structured analysis and design. Computer world is the world where you can’t stand tranquil you have to move with the technology. In computer world the technology doesn’t change daily or hourly, it change it self every moment, new technologies come and exit so quickly. The legacy systems have been built on PSM (Platform specific model) architecture and as technology changes these legacy systems also need to be updated to meet with the new requirements. Some time organizations enhance their setup and also need to update their software systems to meet new requirements.

It’s not possible to retain the same software team (who developed that legacy system) to maintain the system because employee turnover is so high in the computer industry, that’s why companies have to hire new programming teams to add or update components in their legacy system. These maintenance tasks change the original program structure and specification, as a result old documentation doesn’t remains compatible with the software structure. Some time many legacy systems don’t have the documentation and if it exists the consistency of the document with the software code is very squat and complicated to verify.

Reverse engineering of these kind of systems are very difficult task. Many computer aided software engineering tools provide reverse engineering facilities but still it need human expertise and domain knowledge to present the recovered artifacts at different levels of abstraction. Chikofsky and Cross defined the reverse engineering as the process of analyzing the subject system to identify the system’s components and their interrelationship, and to create representation of the system in another form or at a higher level of abstraction [1].

1.1 Related approaches

As a consequence, many interactive CARE (Computer Aided Reverse Engineering) environments have been developed in industry and academy, which used different approaches for reverse engineering. We will see some of more prominent approaches here according to type of information they provide:

- Knowledge based approaches [2]
- Design pattern based approaches [3]
- Program slicing based approaches [4]
- Formal method based approaches [5]
- Domain based approaches [6]
- Program comprehension approach [7]

All above mentioned approaches are best at their own but all of them lack human knowledge and experience to perform reverse engineering. We used the involvement of finest approaches for reverse engineering and introduced a new layer (Knowledge layer) in reverse engineering. Knowledge layer will be responsible to execute user knowledge and experience to perform reverse engineering.

2 Framework for REP

Proposed framework for reverse engineering integrates the existing reverse engineering layers and tools to support
recovery software system artifacts and present them at the higher level of abstraction.

Most of reverse engineering process works and believes all the information of a legacy system is hidden in the source code and documentation, and all these REP (Reverse engineering process) inputs source code, documentation and other related information. But all of these REP overlooks the most important thing, which is user's knowledge and experience. That's why the extracted end result remains not as perfect as it should be. For example if source code, current document and diagrams etc. are available then its understood that the REP should be perfect and do extract the artifacts which are required for maintenance. Suppose if the person who is doing reverse engineering don’t have domain knowledge of the legacy system or don’t have experience (Reverse engineering experience) then the whole REP would be ineffective to extract required software system artifacts. Same problem exists in the reverse engineering tools because RET (Reverse Engineering Tools) Dali [12], PBS [13], Imagix4D [14] and Bauhaus [15] don’t have knowledge layer to integrate human and artificial intelligence.

The paper presents a new layer (Knowledge layer) which uses human’s (who is doing reverse engineering) knowledge and experience also as input with objects (source code, documentation and other diagrams). Fig. 1 sketches an overview of the proposed framework for Improved REP. The input of the knowledge layer is the source code, design documentation, artifacts recovered from reverse engineering process. Knowledge layer also exist somewhere in the human mind while performing reverse engineering task, this framework integrates the human intelligence in the knowledge layer.

![Figure 1 Framework for reverse engineering process](image-url)
2.1 Recovery Process

Process (Latin, processus - movement) is a naturally occurring or designed sequence of changes of properties/attributes, of object/system. The reverse engineering process requires to extracts the artifacts at different levels of abstraction for reverse engineering activities. The extraction heavily depends on the nature of available source code and existing documentations and artifacts require for the maintenance tasks in hand. The required artifacts specifications are mapped to the source code and the existing documents to extract the artifacts for maintenance tasks. The extraction of artifacts also depends on the required artifact specification and mapping process. Reverse engineering is an imperfect process driven by imperfect knowledge [2]. We have tried to improve reverse engineering process with the perfect knowledge layer. Following are the steps we have defined in our knowledge based REP framework.

2.1.1 System Artifacts

In start RET pulls together all the artifacts which are available (source code, documentation or other available UML diagrams etc). Some time there is only source code that RET has to take as a first step in the reverse engineering process. The maintenance activities require the artifacts; the engineers have specific goals for maintenance tasks in hand. What type of artifacts are required and at what level of abstraction ? The artifacts exist at the domain, functional, structural and implementation levels[8][10]. The artifacts at implementation are the files, libraries or directories which contain the particular source codes and describe the relationships between them at this abstraction level. The structural artifacts are the design documents, architectures, processes and at the functional abstraction level the artifacts describe the functionality of the system i.e. Use Cases, Scenarios and other documents. The domain abstraction further comprehends the functionality and logic of the application and the external knowledge about the particular domain.

2.2 Knowledge layer

This is the main important layer that has been presented in this paper which deals and acts like a human mind. AI (Artificial intelligence) is really powerful thing that can be used in the reverse engineering tools to extract, abstract and present the artifacts at the higher level of abstraction. Knowledge layer works on the AI theme to extract the artifacts and present them at the different level of abstraction. Following are the some components which work on the knowledge layer to provide the more accurate and exact data to extraction layer.

2.2.1 Goals

It’s important to define the goals in the REP so the reverse engineering process keep on track and don’t diverse from its line of action and cause overhead. Reverse engineering tools would be responsible to allow user to define some goals for the reverse engineering whether he want to abstract class level diagram or architecture level.

2.2.2 Examine Information

With an ability to represent or recognize (like features or relationship) using knowledge, sensing and understanding objects in the light of goals, knowledge layer will examine all the information which has been provided by the user that whether it’s relevant or irrelevant with the legacy system. That can be done comparing documentation and diagrams with the source code and the goals which were defined.

2.2.3 Objects

Object is some thing that has properties and relations. It might be possible, some objects which were provided for reverse engineering, don’t help in the REP but can diverse the goals of abstraction and increase the cost of reverse engineering. The existing objects (system artifacts) of the legacy software system are,

- Source code
- Documentation
- Diagrams
- All other related information about the software system.

2.2.4 Define Rules

Rules are defined on the basis of source code syntax and semantics using the RE (regular expressions). Regular expressions also have some limitation so the knowledge layer will also check it up whether the regular expression meets with goals. Rules should be as much as strict as they can be so the REP don’t diverse its line of action. For example while extracting the class level diagram it shouldn’t extract or deal with the architecture artifacts.

All of these components make the knowledge layer a powerful base layer for the reverse engineering and make simple the work for extraction and abstraction layers to present the recovered artifacts at the higher level of abstraction. When extraction layer will get all the sanitized artifacts as input it will be more capable and effective.
2.3 Extraction layer

Extraction layer is responsible for extracting the artifacts at the different level of abstraction for reverse engineering. The extraction heavily depends on the nature of available source code, existing documentations and artifacts require for the maintenance tasks in hand. The required artifacts specifications are mapped to the source code and the existing documents to extract the artifacts for maintenance tasks. Extraction produces accurate results if it gets more clean and structured input. Before mostly reverse engineering tools just provide facilities to take input raw source code and documentation which produce not as accurate results as it should. While our KREP provides a base layer for the REP to purify and structure the system artifacts (documentation, source code and other diagrams) to give good input to extraction layer for more accurate results. As KREP has taken place for some initial decision, it also helped extraction layer produce accurate results in lesser time.

2.4 Abstraction layer

The extracted artifacts are also required to abstract at the certain levels. The extracted artifacts are abstracted at the domain, functional, structural and implementation levels [8][9][10][11]. For example the functional description is extracted from the only available source code and Use Cases, scenarios are abstracted from this functional description. The artifacts abstraction also depends on the extracted information and at the levels of abstractions. If the extracted artifacts are well structured it would be easy to abstract the recovered artifacts at the different levels of abstraction.

2.5 Presentation layer

The reverse engineering process also requires presenting the artifacts in a specific format or diagrams (i.e. UML diagrams) at different levels to perform the maintenance tasks in hand. In forward engineering many development processes exist are in practice, and engineers are trained and the preference is to use these and present the system artifacts in these formats and diagrams for different maintenance activities.

3 Conclusions

Reverse engineering is quite obvious in concept, but the layers on layers of old maintained code that resides in our legacy systems, in a variety of languages, make it highly complex technically. So it is very necessary to understand these complex systems before re-engineer or reuse these systems. Reverse engineering does not produce a new program, it simply analyze and displays the parts of the old system. KREP is the more effective framework to perform reverse engineering as it separates the knowledge layers from the extraction layer to provide more accurate and purify input. This paper shows that how human mind think and do abstraction of things and how it can be integrated into the reverse engineering tools to make it effective.

4 References


On Systematic Design of Service-Oriented Architectures

Benjamin Kanagwa  
Radboud University Nijmegen  
The Netherlands  
Email: B.Kanagwa@cs.ru.nl

Ezra K Mugisa  
University of The West Indies  
Kingston, Jamaica  
Email: ezra.mugisa@uwimona.edu.jm

Th.P. van der Weide  
Radboud University Nijmegen  
The Netherlands  
Email: tvdw@cs.ru.nl

Abstract—Because services are developed independently, with no prior knowledge about each other, there is only a limited possibility that such services use similar message templates, initiate calls to any other service, or generate messages to support desired architecture configurations. As a result, construction of service-oriented systems rely on intermediary services to pre-process, transform and route messages to appropriate locations. The paper suggests a systematic design for the intermediary services and construction of services-oriented systems using a reuse drive approach. The approach is founded on category theory - a formal foundation for capturing and preserving structures. The approach supports parallel and incremental inter-connection of services in a planned and reusable manner.

I. INTRODUCTION

Service-Oriented Systems are constructed from independent services that are composed and coordinated to satisfy a set of functional and non functional requirements. The consumer may be an end-user or another service. In service-to-consumer interactions, the behavior of the provider is predefined and any adjustments are possibly on the side of the consumer. The consumer can interact in any way that does not violate the rules set by the provider. The key question we aim at answering is how to transform independent services into a service-oriented system in a systematic manner. By systematic we mean a way that is reusable based on a set of principles and guidelines that provide a sound step-by-step methodology for putting service-oriented systems together.

In this paper, we propose a design approach that blends the structure of services, the composition, coordination, the design and placement of intermediary services into a single formal framework that imposes a sound incremental methodology. Our idea is to rely on category theory [1], [2] which inherently preserves structure and provides tools for composition and incremental construction. Most previous approaches based on Finite State Machines [3], Petrinets [4], Process Algebra [5] emphasize description of service compositions rather than design and construction.

Construction of service-oriented systems is a function of composition and coordination of services. Because services combine on the basis of what they offer, the process of construction of service-oriented systems is the process of consuming services [3]. The importance of composition of services is reflected by the current interest in the subject with most recent efforts targeting automatic composition [6], [7], [8], [9]. Business Process and Execution language for Web Services (BPEL4WS) [10], now a standard for description of composition of services allows one to describe a business process in terms of the behavior of other services. It provides for coordination of services based on the actions. It assumes a single output message for each action. BPEL4WS does not offer a construction approach but a description framework. However, because services are independently designed, owned and deployed, it is a rare occurrence that the consumption of services is free of challenges. Challenges stem from lack of compatibility both at the syntactic and semantic level. Even when they are compatible, there is usually need for intermediary services to ensure smooth coordination of services and address any additional concerns that may be necessary to fulfill the goal of the service-oriented system.

Focusing on service intermediaries is reasonable because they have important responsibilities that include adaptation, coordination and general routine keeping such as logging. Our strategy is to make the coordination of services together with design and placement of service intermediaries more systematic and principled. Our intention is to provide systematic support for direct messaging, and construction of intermediary services that optionally involve the consumer.

A. Web Service SOAP Solution

Web services [11] rely on Simple Object Access Protocol (SOAP) [12] as the messaging framework. Within the SOAP framework, direct messaging is supported through ayciclic message processing as the SOAP envelop moves from node to node. Below we highlight the solution suggested by the web services community.

The SOAP [12] distributed processing model assumes that a SOAP message originates at an initial SOAP sender and is sent to an ultimate SOAP receiver via zero or more SOAP intermediaries. However, SOAP itself does not define any routing or forwarding semantics. It is anticipated that such functionality can be described as one or more features and expressed as SOAP extensions or as part of the underlying protocol binding. A SOAP intermediary is a node which forwards a message to another SOAP node on behalf of the initiator of the inbound SOAP message. SOAP defines two
different types of intermediaries: forwarding intermediaries and active intermediaries.

Forwarding intermediaries process the message according to only the rules specified in the SOAP message, while the active intermediaries may undertake additional processing beyond what is in the rules in the SOAP message. According to the SOAP specification, it is expected that features provided by active intermediaries are described in a manner that allows such modifications to be detected by the affected SOAP nodes. For example, an active intermediary may describe the processing performed by inserting header blocks into the outbound SOAP message that inform downstream SOAP nodes acting in roles whose correct operation depends on receiving such notification. The semantics of such inserted headers require that either the same or other headers to be (re)inserted at subsequent intermediaries as necessary to ensure that the message can be safely processed by nodes yet further downstream. For example, if a message with headers removed for encryption passes through a second intermediary (without the original headers being decrypted and reconstructed), then indication that the encryption has occurred must be retained in the second relayed message.

According to this specification, it is risky for a service integrator to rely on active intermediary nodes that are not under his/her control. The likely situation is that active intermediary nodes may turn out to be coordination points designed and controlled by the service integrator. Most of the work of a service-oriented engineer is to design and manage the coordination points.

During composition, the way messages synchronize is non trivial because some messages become internal to the composite service [13] while new messages may have to be introduced to complete the interaction. The way messages become internal and those that become exposed by the composite service is fundamental to the composition of services. Service composition is a recursive problem because composite services can be exposed as new ‘atomic’ services whose coordinated behavior is concealed from the consumer. With this in mind, construction of service-oriented systems can be carried out in an incremental manner.

II. Example and Practical Scenario

We use the example from Web services Interoperability Organization [14], [15] with sequence diagrams in Fig.1. The application being modeled is that of a Retailer offering Consumer electronic goods to Consumers; a typical B2C model. To fulfill orders the Retailer has to manage stock levels in warehouses. When an item in stock falls below a certain threshold, the Retailer must restock the item from the relevant Manufacturer’s inventory (a typical B2B model). In order to fulfill a Retailer’s request a Manufacturer may have to execute a production run to build the finished goods. The above scenario involves three simple services, the Shop, Warehouse and Manufacturer. Since, we can have many instances of each service; the example is rich enough to demonstrate our formal systematic design. Starting with these three services, we want to explain how to compose them directly without implying changes in the original services. While composing services in a service-oriented environment, the following restrictions apply to the service-oriented approach (i) services are consumed ‘as is’ and therefore we can not change the templates of the messages they send or receive. (ii) services operate in a request-response/client-server manner. So a reply is always sent to the requestor of the service.

III. The Approach

Our work extends earlier work, including some parts that are most closely related to work in [16], [17], by allowing explicit separation of guarantees from outputs and creation of classes over action outputs. We also explicitly model replication of outputs which allows a larger set of architecture configurations. Our approach is also constructive; one can construct a larger system by following a step-by-step methodology. The approach is based on the notions of coordinated activity, guarantees (semantic implication) and public view of services. The idea of guarantee is informally introduced in [3], [16]. It can also be related to triggers in [18]. Consider the shop service in Fig. 1, which specifies actions getCatalog and submitOrder. Ideally these actions combine to perform the activity sellItem which could result in an item being SOLD or FAIL for some reason. SellItem is not considered as an action but rather a coordinated activity because (i) it is not transparently presented to the user (ii) it is entirely in terms of other actions. By transparency we mean that a user does not invoke sellItem without being directly involved in other intermediary actions. The actions getCatalog and submitOrder make the public view of the shop service, because they are directly invoked by the consumer.

Normally, the expected results of a coordinated activity do not match (both in number and type) the outputs of the actions involved and this creates a source of behavior incompatibility [19], [20]. To mitigate this challenge, we create ‘classes’ of coordination over the outputs of actions. The outputs of an action are partitioned into sets that associate meaning in terms of the coordinated activity.

The rest of the approach is cast in a categorical setting that preserves structure, presents a composition and coordination framework. The specific contributions of this paper are (i) structure for representing services and service-oriented systems. The structure is formal which implies there is a precise and concise meaning associated to it. (ii) We present a systematic and incremental means of constructing larger systems based on a categorical approach making the process more planned and reuse driven (iii) We provide a criteria for design and placement of service intermediaries to complete the process of service-oriented design. (iv) Finally we present loose semantics for the core building blocks of service intermediaries.

One advantage with our approach is that it preserves the structure of services, which makes the process of constructing services incrementally reused and systematic. This approach provides for intermediate services (wrappers) that perform the
role of transforming, coupling messages and adaptation of services.

IV. RELATED WORK

In [18], [21], a category theoretical approach to service-oriented modeling is presented. Their categorical formalization is centered on the notion “coordination contract” which they relate to their earlier work on software architectures, Parallel Program Design and Reconfigurable distributed systems all coached in different categorical settings. As the name suggests, coordination interfaces are intended to coordinate systems. Algebraic approaches proposed by [22] aim to capture coordination in the co-span object. Our work is directed towards design with emphasis on intermediaries and structure of services.

In [16], a set of interface theories for web services are presented. The work emphasizes the interface theory with models for substitutability, compatibility and well-formedness of interfaces. Well-formedness is also discussed in [3]. We include these concerns through our concept of classes over action outputs.

Other related work deals with issues of automatic composition [8], [3], [9], [23] where the target is to find a set services and combine them to fulfill a request. A request may be just a result/output or a service that exhibits a given behavior. Although the major aim there is formal description of automatic composition approaches, the models used form a major part of construction and computational structures in our categorical setting especially tree-based structures.

In [6], [24] they deal with adaptation, replaceability and compatibility. In relation to these concepts, our approach is not expressive enough especially in terms of behavioral adaptability at the level of service behavior.

V. THE CATEGORICAL APPROACH

The approach proposed here is guided by three principles. First, the process of constructing a service-oriented system is the process of consuming services [3]. Second, in agreement with [25], we want to keep the concept of service at the center of the construction (design) process. So, we would like to keep a clear view of the consumer and provider entities in the entire process. Third, we would like to support the autonomy and reactive nature of services as much as we can. Our key guiding factor here is that services combine on the basis of what they provide independently but not on the basis of what they offer to each other. For instance, a medical service provider can combine with an insurance service provider not because the two providers (directly) offer anything to each other, but because they produce an added valued of a medical insurance service when combined.

To support these concepts into the design of service-oriented systems in a systematic manner, we use category theory [1]. A category $E$ consists of two classes, the members of the first of which denoted by the letters $X, Y, \ldots$ – are called objects (structures) and the members of the second of which denoted by the letters $f, g, \ldots$ – are called arrows (morphisms). Each arrow $f$ is assigned an object $X$ as domain and an object $Y$ as codomain, indicated by writing $f : X \rightarrow Y$. If $g$ is any arrow $g : Y \rightarrow Z$ with domain $Y$, the codomain of $f$, there is an arrow $fg : X \rightarrow Z$ called the composition of $f$ and $g$. For each object $Y$ there is an arrow $idy : Y \rightarrow Y$ called the identity arrow of $Y$. These notions are assumed to satisfy the following identity and associativity axioms:

$$f.idy = f, idy.g = g, f(g)h = (fh)g$$

Given two categories $D$ and $E$, a functor $F$ from $D$ to $E$ consists of a pair of functions (both denoted by $F$), one from the class of objects of $D$ to that of $E$, and the other from the class of arrows of $D$ to that of $E$, such that

$$F(f) : F(X) \rightarrow F(Y) \in E; F(id_X) = id_{F(X)}$$

for all objects $X$ of $E$ and

$$F(gh) = (F(g))F(h)$$

Morphisms preserve the structure of the objects. In software engineering morphisms $p : X_1 \rightarrow X_2$ in a suitable category are used to define (i) refinement - specifies a way in which the target object is a refinement of the source object, (ii)
inclusion- capturing legal transformations on objects to form more complex objects. Inclusion morphisms specify part-of relation (iii) interpretations - capture relationships between theories and can be used to model relationships between abstract and concrete implementations (iv) simulations - target execution behavior of systems such as behavior of the stack.

With the categorical approach, suitable categories are used to represent the structure of different artifacts that make up the target entity such as a service or service-oriented system. The process of putting small services together into bigger services/systems is then based on standard categorical techniques such as colimits [26].

Our use of category theory is restricted to defining and preserving the structure of services and providing a systematic means of constructing larger services from smaller services in an incremental and principled manner through the pushout construction [2]. The morphisms that we target are therefore inclusion morphisms, that aim at expressing how smaller services become part of larger services.

VI. NOTATION FOR REPRESENTING SERVICE ARTIFACTS

This section presents the general notation for representing service artifacts such as messages and actions and their relationships. The primitive artifacts are actions and activities. Rules imposed on the relationship of these artifacts yield new artifacts such as service interfaces. From now on, $M^I$ and $M^O$ are the sets of input and output messages respectively and $A$, a set of actions.

A. The Actions

A service action is associated with a single input message and zero or more output messages. Each output has a guarantee associated with it. A guarantee is used to provide (i) evidence of occurrence of an action (ii) semantic interpretation to create ‘classes’ of coordination based on the activity in which an action participates. Let $O(a)$ represent the set of possible outputs of action $a$. Consider the example of an action submitOrder, with input submitOrderReq and corresponding output and guarantee pairs; submitOrder,submitOrderReq =\{(\text{SubmitOrderResp, OK}), \text{BadOrder, Fail}, \text{InvalidCode, Fail}, \text{ConfigFault, Fault}\}. The model assumes that only a single output is possible at any given moment in time. However, the design of intermediary services allows for replication of outputs to coordinate more services using single output. Below is a formal definition of an action alphabets.

**DEFINITION 1** The output alphabet of an action $a$ is a pair $(M, F)$, where $F$ is a total map from $M$ into $O(a)$, where $M$ is the set of actions that process message $M$. An action morphism for an action $a$ in $M$, denoted by $f_M : A_1 \rightarrow A_2$ is a total map such that $F_1(a) = F_2(f_M(a))$ for all actions $a \in M$.

The action alphabet and action morphism form a category of actions $\text{ACT}$. The intention of the above definition is similar to idea in [27] where slot and slot morphisms are defined. The underlying intention is to synchronize services on the type of output for each action. Therefore each $O(a)$ defines the codomain of action $a$ and $F(a)$ is the actual message/value that is output by the action $a$. Consider an action SubmitOrderReq. The types of messages that it can output are seen as ‘variables’ that can be changed by the action. Viewed in this perspective the slots of different actions are disjoint. Put in another way, no two actions can change the same slots. In other words, the output types of each action are unique.

B. Service Activity

A service activity is coordinated over a set of visible actions. An action is visible if it can be invoked directly by a consumer. A visible action may invoke a series of other actions which return one of the expected outputs. Consider the activity sellItem with possible outputs SOLD or FAILED. An activity has no explicit inputs; they depend on the sequence of actions involved.

**EXAMPLE 1** Below is an example of an activity

```
SellItem[SOLD, FAILED]
getCatalog:\{(\text{GetCatalogResp, submitOrder}), \text{GetCatalogFailed, SellItem.FAILED}\}
submitOrder:\{(\text{SubmitOrderResp, SellItem.SOLD}), \text{BadOrder, SellItem.FAILED}, \text{InvalidCode, SellItem.FAILED}, \text{ConfigFault, SellItem.FAILED}\}
```

In this activity, the GetCatalog action has two possible outputs. The first output, GetCatalogResp is coordinated with the action submitOrder. The second output GetCatalogFailed has been ‘interpreted’ to imply FAILED in terms of the coordinated activity. The idea behind the guarantees as used in this article is to partition all action outputs into classes of the guarantees of the coordinated activity. This technique facilitates adaptation because the coordinated actions are able to account for all the behavior (outputs) expected by the coordinated activity. Note that although this example is structurally similar to that used in [16], our approach is significantly different. First the coordination is based on both actions and the different outputs associated with them. Secondly, explicit separation of outputs and guarantee/semantic implications allows one to form classes over the outputs of actions - a technique that enforces adaptation of services to support coordinated activities. In this example, the outputs of each action are partitioned into two classes determined by the output of the coordinated activity. Because the activity SellItem is coordinated over actions, the GetCatalogResp output is followed by submitOrder which also determines its class/guarantee. The outputs of action submitOrder have the following classes: SOLD:\{(SubmitOrderResp\}, FAILED:\{BadOrder, InvalidCode, ConfigFault\}. Although we have used outputs for clarity (because they have descriptive names), classes are formed over the corresponding guarantee/semantic implication.
The classes/guarantee of outputs connected to other actions are determined by the final actions that they invoke. Since outputs of actions may be connected to other actions in different ways, there is need for a mechanism of evaluating guarantees. The connection can be sequential or in parallel. Parallel connections have two variations, one as alternative options and the other as complementary options. We use the symbols \( \cap \) and \( \lor \) to represent sequential, parallel alternatives and parallel complements respectively. They are evaluated as follows:

- \( G_1 \cap G_2 = G_2 \) the guarantee of the last action
- \( G_1(G_2 \lor G_3) = G_1 \) if G2 or G3 imply G1 otherwise take G2 or G3
- \( G_1(G_2 \lor G_3) = G_1 \) if both G2 and G3 imply G1 otherwise take G2 or G3.

### VII. SERVICE STRUCTURE

The service structure consists of a collection of actions, input messages and output messages.

**Definition 2** A service signature \( \Phi \) is defined as follows

| \( M_I \) | set of input messages |
| \( M_O \) | set of output messages |
| \( A \) | a set of actions |

**Example 2** The signature of the shop service can be represented as follows:

\[
M_I = \{ \text{SubmitOrderReq, GetCatalogReq} \} \\
M_O = \{ \text{SubmitOrderResp, BadOrder, InvalidCode, ConfigFault, GetCatalogResp} \} \\
A = \{ \text{getCatalog, substringOrder} \}
\]

**Definition 3** A service signature morphism \( h = (M_I^1, M_O^1, A_1) \rightarrow (M_I^2, M_O^2, A_2) \) is defined as tuple \( < h_I, h_O, h_A > \) defined as follows

\[
\begin{align*}
h_I : & M_I^1 \rightarrow M_I^2 \\
h_O : & M_O^1 \rightarrow M_O^2 \\
h_A : & A_1 \rightarrow A_2
\end{align*}
\]

**A. Service-Oriented System Structure**

The notion of design used is based on the concept of coordinated activities. We perceive a service-oriented system as a connection of various services coordinated to perform a specific activity. A service-oriented system may be coordinated to perform several activities.

A coordinated activity has the following conditions:

(i) It has set of possible outcomes. For example for the shop service we shall define an activity \( \text{sellItem} \) as having two possible classes of outcomes \( \{\text{SOLD, FAIL}\} \).

(ii) An activity has a public view part and a detailed coordinated part. The public view of the activity is defined only in terms of actions that are directly accessible and invoked by the consumer.

(iii) For each action that takes part in any activity, its possible output is partitioned over the set of outcomes of an activity. The idea is for each output of an action, we need to know its implication in terms of the activity outcomes.

(iv) During incremental design, two actions from the same public view must not violate any order imposed by the service for that public view.

Condition (i), ensures that we separate between coordinated activities and actions defined by the service. The inputs of the activity are determined by the different inputs of the actions involved. Condition (ii), ensures that coordinated activities are transparent to the user. This is further enhanced by restricting that all activities map their outputs to those of the coordinated activity. Condition (iii), ensures that adaptability of service is built within the framework, by accounting for the behavior of actions in terms of the activities it coordinates.

The idea behind condition (iii) of partitioning guarantees of synchronized outputs over activity outputs is to provide a simple technique for establishing a ‘subtype’ relationship between an activity and actions that synchronize at different points. In a ‘subtype’ relation, to avoid unexpected outcomes, all behaviors (outputs and methods) of the subtype must be accounted for [28]. Therefore, the guarantee mapping between outputs of actions and outputs of coordinated activity ensures that any extra outputs by synchronized activities do not introduce any extra behavior that was not anticipated by the coordinated activity.

Condition (iv) is to ensure that if a service participates in the same activity more than once, it must ensure that new actions are inline with the relative ordering(specified by the instance) between the action and actions already involved. This minimizes potential deadlocks.

**Definition 4** A service-oriented system signature \( \Psi \) is defined as a pair \( (\Phi, \Delta) \) with morphisms \( k = (\Phi_1, \Delta_1) \rightarrow \)
(Φ₂, Δ₂) such that hₖ : Φ₁ → Φ₂, k₆ : Δ₁ → Δ₂ such that Φ is the service signature and Δ is a set of activities.

The morphisms in definition 4 are inclusion morphisms and they establish the classes of the coordinated activity as well the outputs on which services synchronize. The above structural representation is similar to that used in BPEL4WS [29] also used in [30]. Examples of service representation are given in the next section.

VIII. COMPLETE EXAMPLE CONSTRUCTION

Table II shows the actions and outputs of the warehouse WH and Manufacturer, MF whose specifications are given below

\[
WH ≡ M^I = \{\text{ShipGoodsReq}\}, M^O = \{\text{ShipGoodsResp, ConfigFault}\}, A = \{\text{shipGoods}\}, \Delta = ∅
\]

\[
MF ≡ M^I = \{\text{POSubmit}\}, M^O = \{\text{ackPO, POFault, ConfigFault}\}, A = \{\text{submitPO}\}, \Delta = ∅.
\]

To combine the shop and warehouse in the coordinated activity sellItem, we use a shared object (CH) which identifies the actions, output and guarantees. Use of shared objects (cospan) is standard technique[31] to construct new combined objects through pushout construction.

\[
CH_1 ≡ M^I = \{\text{u}\}, M^O = \{\text{Sold, Failed}\}, A = \{\text{a}\}, \Delta = ∅.
\]

The morphisms f and g

\[f : u \mapsto \text{SubmitOrderReq, Sold} \mapsto \{\text{SubmitOrderResp}\}, \text{Failed} \mapsto \{\text{BadOrder, InvalidCode, ConfigFault}\}, a \mapsto \{\text{submitOrder}\},\]

\[g : u \mapsto \text{ShipGoodsReq, Sold} \mapsto \{\text{ShipGoodsResp}\}, \text{Failed} \mapsto \{\text{ConfigFault}\}, a \mapsto \{\text{shipGoods}\} .\]

The corresponding categorical diagram is given below. The pushout for S' is given in Fig 2 (because of space, only the coordination part is presented) S' is the pushout construction of the shop SH and the warehouse WH, followed by an incremental composition of the MF which leads to S'', the the coordinated structure of shop, warehouse and manufacturer. The combined structure of S' is given in Fig 2. The structure of S'' is similarly constructed using appropriate actions.

\[
\text{SellItem[SOLD, FAILED]}\]

getCatalog:\(\langle\text{getCatalogResp, submitOrder}\rangle,\)
\(\langle\text{getCatalogFailed, SellItem.FAILED}\rangle)\)
submitOrder:\(\langle\text{SubmitOrderResp, shipGoods}\rangle,\)
\(\langle\text{BadOrder, SellItem.FAILED}\rangle,\)
\(\langle\text{InvalidCode, SellItem.FAILED}\rangle,\)
\(\langle\text{ConfigFault, SellItem.FAILED}\rangle)\)
shipGoods:\(\langle\text{ShipGoodsResp, SellItem.SOLD}\rangle,\)
\(\langle\text{ConfigFault, SellItem.FAILED}\rangle)\)

Fig. 2. Shop, Warehouse and Manufacturer Configuration

The structure is Fig.2 can be equivalently represented as a tree structure with actions as nodes and outputs as branches. A node with many branches is translated into an intermediary service.

IX. INTERMEDIARY SERVICES AND SYSTEMATIC DESIGN

We have already expressed the importance of intermediary services in the construction of service-oriented systems. Service intermediaries serve a role similar to adaptors and message brokers [32]. Adapters may intermediate between other applications rather than services. We limit our scope of service intermediaries to mediating between services.

We propose basic building blocks for the intermediaries which form most of the implementation for service-oriented systems that rely on third party services. We therefore think that once the construction of intermediaries is handled in a principled manner, then we have a systematic means of designing service-oriented systems. The categorical constructions above give a ‘specification’ of service intermediaries, which may be implemented as a single centrally coordinated service or split into multiple intermediaries.

A. Primitive Intermediary Services

Intermediary services serve both as forwarding and processing nodes in sense that they ensure that messages forwarded are in the format expected by the next service. Specifically, intermediary services pre-process messages and then route them to their destinations. In this design, an intermediary node may ask to receive the response or be sent to another intermediary service. The combined collaboration of the intermediary services together with other nodes (predefined services) define a service-oriented system architecture. We generalize three primitive intermediary services; Forwarder, Replicator and Joiner that can be used to build any architecture of a service-oriented system.

- **Forwarder**: receives a message from one node, pre-process it and forwards it to another node.
- **Replicator**: receives a message from a single node, pre-process it into multiple messages that are sent to different nodes.
### Table II

**WARE HOUSE, CALLBACK AND MANUFACTURER SERVICES**

<table>
<thead>
<tr>
<th>action</th>
<th>input</th>
<th>output</th>
<th>guarantee</th>
</tr>
</thead>
<tbody>
<tr>
<td>shipGoods</td>
<td>ShipGoodsReq</td>
<td>ShipGoodsResp</td>
<td>Ok</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ConfigFault</td>
</tr>
<tr>
<td>WARE HOUSE SERVICE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>submitSN</td>
<td>SNSubmit</td>
<td>ackSN</td>
<td>orderOk</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CallbackFault</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>errorPO</td>
<td>ProcessPOFault</td>
<td>ackPO</td>
<td>submitPOFault</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>CallbackFault</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MANUFACTURER SERVICE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>submitPO</td>
<td>POSubmit</td>
<td>ackPO</td>
<td>Ok</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>POFault</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Joiner

- **Joiner**: receives a message from multiple sources, preprocess and then sends to a single node

We capture the above primitives in a systematic manner using the categorical framework. They can therefore be extended to any number of messages.

In the following subsections, we provide loose semantics for the **forwarder**, **replicator** and **joiner** intermediary services.

1) **Forwarder Semantics**: Below we give the general structure of a forwarding intermediary service.

\[
\begin{align*}
\text{FWD} \equiv & \quad \text{Ports} = \{ \text{in}, \text{out} \} \\
M^I &= \{ y \} \\
M^O &= \{ u \} \\
\Delta &= [ m := \text{in}(y) \\
& \quad u := \text{transform}(m) \\
& \quad \text{out}(u) ]
\end{align*}
\]

An action can lead to different outcomes; we shall assign an output location (port) for each possible outcome associated with an input message. The transformation function represents the pre-processing needed by the service integrator such that there is syntactic compatibility with the providers to be interconnected. The transformation may involve renaming the message, preprocessing of message components or removal of message components to ensure syntactical compatibility with the next service provider.

2) **Replicator Semantics**: Below is the general structure of a replicating intermediary service.

\[
\begin{align*}
\text{RPL} \equiv & \quad \text{Ports} = \{ \text{in}, \text{out}_1, ..., \text{out}_n \} \\
M^I &= \{ y \} \\
M^O &= \{ u_1, ..., u_n \} \\
\Delta &= [ \begin{array}{l}
\text{for}(i = 0 \text{ to } n) \\
\text{do}
\quad m_i := \text{in}_i(y_i) \\
\quad u := \text{transform}(m_1, ..., m_n) \\
\quad \text{out}(u) 
\end{array} ]
\end{align*}
\]

For a single input message, it transforms it into multiple messages to be sent to different service providers. The transformation may involve simple renaming of messages in which exact copies are reproduced or it may involve preprocessing of message components or removal of message components to ensure syntactical compatibility with the target service providers.

3) **Joiner Semantics**: This is a general structure of a joining intermediary service.

\[
\begin{align*}
\text{JNR} \equiv & \quad \text{Ports} = \{ \text{in}_1, ..., \text{in}_n, \text{out} \} \\
M^I &= \{ y_1, ..., y_n \} \\
M^O &= \{ u \} \\
\Delta &= [ \text{for}(i = 0 \text{ to } n) \begin{array}{l}
\text{do}
\quad m_i := \text{in}_i(y_i) \\
\quad u := \text{transform}(m_1, ..., m_n) \\
\quad \text{out}(u) 
\end{array} ]
\end{align*}
\]

As expected, a joiner intermediary service does the opposite of the replicator. It receives information from multiple sources and then combines it into a single output.

### B. Placement of Intermediary Services

The next logical question is to identify ‘positions’ where to place intermediary services. As a general rule, intermediary services are needed wherever there is a link between actions of different services. In practice, **forwarding**, **replication** and **join** services could be offered by the same node, managed by the service integrator.

An output slot may be associated with more than one action at the same time or at different times. In this case, the slot value has to be replicated and synchronized with different actions. In the coordinated structure such points will appear as \( \langle \langle \text{SubmitOrderResp}, \text{(shipGoods1 } \cup \text{ shipGoods2)} \rangle \rangle \) where the output is coordinated with several other actions. In this example, two warehouses are involved. Because we do not assume message templates to match, synchronization points are modeled using service intermediaries. One-to-One synchronization point is modeled by a **forwarder**, **replicator**, and **joiner**.
One-to-Many is modeled by a replicator. The typing of message templates/envelops is not fixed in that the intermediary services can transform between different message templates to satisfy service interfaces.

X. Conclusions And Future Work

Smooth coordination of services is faced with three interrelated concerns of compatibility, adaptability and replaceability. The key questions are if two services are compatible such that one can replace the other and if not what is required to adapt the services such that they work together. Our design approach considers a service-oriented system as a collection of activities where coordination of different services is aimed at performing a given activity with a set of possible outcomes.

To deal with adaptability, we explicitly separate service outputs from semantic implication or guarantees. Semantic implications provide means of coordinating activities over actions with several outputs. It answers the question, what does an action output imply in terms of the coordinated activity? In other words, outputs of service actions classified in terms of the expected result of the coordinated activity. The outputs of an activity provide classes over the outputs of actions involved in a particular activity.

We have provided a layout for a systematic methodology for constructing service-oriented systems. The approaches address some of the compatibility issues through its construction based on the notion of activities and guarantees that are used to form classes over action outputs. We plan to provide a more firm compatibility model that address compatibility at the level of services within the construction approach. We intent to automate the above process such that the task of service integrator is reduced to classification of action outputs in terms of coordinated activities, and minor adjustments on transformation of messages where inputs to do not exactly match with outputs.

REFERENCES

Abstract—Modern enterprises need to respond effectively and quickly to opportunities in today's ever more competitive and global markets. Service Oriented Architectures (SOA) is an emerging approach that addresses the requirements of loosely coupled, standard-based, and protocol independent distributed computing. One of the main challenges for SOA initiatives is the lack of unique and standard approach and it leads to urgent need for an adaptable software methodology systematic approach.

This paper presents reasons to this urgent need and offers an approach for overcoming SOA developing challenges. The approach combines Unified Process (especially RUP) with SOA and represents the pattern for service definition with Unified Modeling Language (UML) and helps to make a solution for developing SOA projects.

Keywords: Service Oriented Architecture. Rational Unified Process. Unified Modeling Language.

Introduction

SOA is designed to allow developers to overcome many distributed enterprise computing challenges including application integration, transaction management, security policies, while allowing multiple platforms and protocols and leveraging numerous access devices and legacy systems.

With SOA infrastructure, we can represent software functionality as discoverable web services on the network. A web service is a software component that is accessible by means of messages sent using standard web protocols, notations and naming conventions, including the XML protocol.

The major driving factors of a typical software development project are the application development process, project management, and the technologies used. These are all driven by the specific consultants assigned to the project. A generic software development project has four variables: time, budget, scope, and quality. Compromising on any one variable has an impact on the overall project. Because of changing business needs, scope and quality are the two most difficult factors to manage. Technological complexities can lead to problems with time and budget management, as well.

SOA projects are significantly more complex than typical software projects, because they require a larger cross-functional team along with correspondingly more complex inter-team communication and logistics. Typically, an SOA is not well defined, and the vision for the final result is thus often not clear at the project's inception. An SOA can have a very positive impact on an organization, but it comes with a potentially high price tag. Moreover, an SOA project has a higher risk of failure than other projects. The key factors that can help an SOA rollout succeed are a clear definition of the development process, enhanced lines of communication across project teams that know the business, and clear support and governance policies.

Why Unified Process

Researchers and practitioners have realized that processes can not be defined and "frozen" once for all. Processes need to undergo changes and refinements continuously in order to increase their ability to deal with the requirements and expectations of all stakeholders, i.e. processes need to be continuously assessed and improved.

The Rational Unified Process (RUP) [6] is a software engineering process that provides a disciplined approach to the assignment of tasks and responsibilities within a development organization [5]. Its goal is to ensure the production of high-quality software that meets the needs of its end users within a predictable schedule and budget.

RUP uses the Unified Modeling Language (UML) [4] to model the software and describes how to apply best
practices of software engineering via guidelines, templates, and tool mentors for all critical software lifecycle activities. Within the growing number of textbooks and articles on RUP, one finds descriptions such as “RUP is a software engineering process” [5], “RUP is a process framework” [5], “RUP is a process and a process framework” [5], and “RUP serves as the organization-wide process” [6]. Both “process” as well as “process-framework” are seen at the same level in the meta level hierarchy, the difference being that framework expresses the complete set of RUP activities from which a project-specific process or organization-wide process can be configured (essentially a tailored subset of RUP) [7],[5].

The Unified process has 4 major advantages:

1. Risk driven
2. Iterative
3. Architecture centric
4. Use case driven

These four specifications make Unified Process suitable and unique for combining and using in software engineering developments

Overview of The Service Oriented Architecture

An SOA provides a flexible architecture that unifies business processes by modularizing large applications into services. A client from any device, using any operating system, in any programming language, can access an SOA service to create a new business process. An SOA allows the creation of a collection of services that can communicate with each other using service interfaces to pass messages from one service to another, or coordinating an activity between one or more services [1].

The SOA’s loose-coupling principle (especially the clean separation of service interfaces from internal implementations) to guide planning, development, integration and management of their network application platforms make them indispensable for enterprise wide and cross-enterprise applications [11].

There are basically three functions that must be supported in a service-oriented architecture:

1. Describe and publish service
2. Discover a service
3. Consume/interact with a service

In this paper one of the main challenges of SOA projects which is service definition is reviewed.

The scope of service definition

The following is a non-exhaustive list of the numerous categories in which a service could require specification:

- Performance,
- Capacity,
- Business organization,
- Risks and issues,
- Ownership,
- Reliability,
- Security,
- Business impact,
- Tolerance,
- Service contract (pre-conditions, post-conditions, and invariants), and
- Dependencies.

What is therefore required is a set of standards whose deterministic elements may be defined, managed, and monitored. Some elements will remain beyond computing scope for some time to come because they’re “soft” elements that require human judgment to be applied [9].

Service-Oriented Architectures in UML

The method of this approach for the roles and artifacts of the service-oriented style is UML. Services are components with stereotype “_service_”. Provider and requester are interpreted as roles of components implementing and using an interface, respectively (the business perspective of e.g. the provider as organization owning the service is not considered in this paper).

A service registry is a service implementing a special interface (marked by “?” as shorthand for a stereotype “_registry_”), where service descriptions can be published by providers and queried by requesters. A service description is represented as a UML package marked by a stereotype “_desc_”, (see the upper right of Fig. 1). The models contained in this package specify the service and its interface(s) as indicated by the “_specifies_” dependencies. Requesters store their requirements for a service in packages marked by a stereotype “_req_”. We assume a conceptual relation “_satisfies_” to represent the fact that all required properties are guaranteed by a description. Further, we have to represent the fact that a component (requester or registry).
The SOA style provides three distinguished operations to publish and discover service descriptions, and to bind to a service. In the bottom of Fig. 1 a collaboration diagram is provided to specify the semantics of the find operation by which a requester obtains all service description known to a registry that satisfy the requester’s requirements [10].

Combining SOA and Unified Process

The major driving factors of a typical software development project are the application development process, project management, and the technologies used. These are all driven by the specific consultants assigned to the project.

In order to make a standard for SOA development, best process is Unified Process. In this paper focus is on RUP. Each phase of this approach maps onto RUP. Although RUP is a generic process framework that can be specialized for a large class of software systems for different application areas, different types of organizations, different competence levels, and different project sizes [6], there may be some challenges.

The tools gap between various structure information sets. The gap occurs because the tools do not communicate information with each other but require people to interpret the unstructured information and towards an acceptable definition of services methodologies such as Extreme Programming (XP) abhor it, the Rational Unified Process requires it, and it remains the way that developers document most systems and approve changes—via verbose, word-processor-created documents.

While XP maintains that the truth is in the code, the reality is that modern computing languages do not have concrete mechanisms for the definition of service beyond a simple interface [9].

In this paper, an approach for identify services are proposed to close this tools gap and provides support for the automatic flow of information among architects, designers, developers, maintainers and, most importantly, the clients. The iterative work-flow of RUP provides the possibility to reach a standard definition.

This approach predefined a methodology that targeted specifically at SOA projects that are underway in any organization. The proposed approach offers 10 iterations, for which, in each iteration one of the RUP phases is developed.

Iterations require business modelling, requirements, analysis and design, implementation, test, deployment, configuration and change management, project management, and environment. In each iteration activities, roles and artefacts are modified based on SOA. Both RUP and SOA meet the Capability Maturity Model (CMM). The combination of RUP and SOA maps to CMM [12].

Figure 1. Syntax of UML for SOA and semantics of the find operation.
Conclusions

SOA is attracting more and more attention recently, hence the need of methodology and standards appear serious through the software projects.
In this paper one of the approaches for SOA methodologies discussed by the advantages of unified process. The Unified process has 4 major advantages:
1. Risk driven
2. Iterative
3. Architecture centric
4. Use case driven

These four specifications make Unified Process suitable and unique for combining and using in software engineering developments.

The Rational Unified Process (RUP) presents a well-defined approach on software project management and software engineering processes. The assessments show that RUP meets most of the requirements for combining with Service Oriented Architecture (SOA) in order to making a software methodology. This approach predefined a methodology that targeted specifically at SOA projects that are underway in any organization. The proposed approach offers 10 iterations, for which, in each iteration one of the RUP phases is developed. By combining RUP and SOA and using their capabilities, the requirement of Capability Maturity Model is also met.

References

SESSION

TOOLS AND LEGACY SYSTEMS

Chair(s)

TBA
“Unit Metrics” – A Tool to support Refactoring in Agile Software Development

Martin Kunz, Niko Zenker, Steffen Mencke, Reiner R. Dumke,

University of Magdeburg, Germany
{makunz, mencke, dumke}@ivs.cs.uni-magdeburg.de
nzenker@iti.cs.uni-magdeburg.de
P.O. Box 4120
D-39016 Magdeburg
Fax +49391 67 12810
http://kunz.smlab.de/

Abstract

Agile Software Development Methods are nowadays wide spread and accepted. From the Software Measurement point-of-view not all metrics and methods from conventional lifecycle models can be used without adaptation. Distinct techniques in agile software development like refactoring needs new approaches and quality models in the area of software measurement. Therefore this paper describes a quality model, distinct metrics and their implementation into a measurement tool for quality management in agile software development.

1. Introduction

Many technological ambitious products were designed with new complex functionality. Due to the fast alteration and the high cost of change in the late life cycle phases the agile software development method becomes more important in this field of application. The demand for refactoring software products developed with agile software development services from the fact that the content of the software is often produced very quick and standards of software engineering are not used throughout the whole project. Agile software development methods like eXtreme programming try to decrease the cost of change and therewith reduce the overall development costs.

The different cost of change for agile software development in comparison with traditional software development according to the project progress is shown in Figure 1.

![Figure 1: Cost of Change compared to development method [1]](image)

The major element of the XP life cycle is the "Iteration". The iteration is recurring event in which an actual version is edited for example by adding additional functionality, correcting errors or removing unnecessary functionality. In this iteration the programmer is concerned about the functionality, to provide the desired result to the customer.

To realize this totally different project characteristic, the agile software development has established a different life cycles in comparison with traditional life cycle models like waterfall-model or V-Model.

The general characteristics of the XP life cycle are shown in Figure 2.
Besides the different life cycle, agile software development is distinguished by one major element: the refactoring.

Refactoring means the change of source-code and software-design without altering the functionality or external behavior of software [1]. In XP, a refactoring cycle is proposed after each iteration to adjust the software product. The main goal is to ensure product quality attributes [3].

Therefore the refactoring cycle has essential importance for agile software development, so it is a fundamental toehold to support agile software development.

Since traditional standards for software product quality like ISO/IEC 91216 [4] are not designed for agile software development, we have to identify a new set of metrics, thresholds and measurement artifacts which are suited for agile software development and especially for the refactoring.

After describing the quality model consisting of selected metrics and thresholds in the following chapter we present an implementation of this model into an Eclipse Plug-In [12] to support the refactoring cycle in agile software development.

2. Product Metrics for Agile Software Development

To support agile software development and especially refactoring, mainly source-code based product metrics are beneficial to increase quality and productivity [5].

Primarily internal quality attributes have to be ensured to control the source-code quality and to evaluate refactoring steps [6].

The goal of our approach is to combine refactoring with software measurement to give advice about the following aspects:

- Appropriate point in time for necessary refactoring steps
- Significance of a refactoring step and the apparent quality effect
- Side effects of refactoring steps

With these three aspects one can ensure quality along refactoring steps.

The metrics should deliver indices for distinct refactoring steps and they should be easily interpretable. The measurement results should be a trigger or activator for useful refactoring steps and they should avoid quality loss through refactoring steps.

Defining a set of metrics for distinct information need or specific conclusions is solved with various possibilities. Traditional top-down approaches like the Goal Question Metric-Method [7] can be used. Additionally, metrics and calculation methods out of agile environment [6] [8] [3] needs to be considered.

One major base was the metrics suite from Chidamber & Kemerer [9]. Secondly, we chose source code metrics among others: Number of Name-Parts of a method (NNP), Number of Characters (NC), Number of Comment-Lines (CL), Number of Local Variables (NLV), Number of Created Objects (NCO), and Number of Referring Objects (NRO). [8][10]
For the interpretation and analysis we normalize the value according to different thresholds.

The mapping between the normalized values and the linguistic quality terms are shown in the following table:

<table>
<thead>
<tr>
<th>Linguistic term</th>
<th>Normalized value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best/Excellent</td>
<td>1.00</td>
</tr>
<tr>
<td>Very Good</td>
<td>0.82</td>
</tr>
<tr>
<td>Good</td>
<td>0.66</td>
</tr>
<tr>
<td>Average</td>
<td>0.50</td>
</tr>
<tr>
<td>Poor</td>
<td>0.32</td>
</tr>
<tr>
<td>Very Poor</td>
<td>0.16</td>
</tr>
<tr>
<td>Worst</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Table 1: Mapping between values and linguistic terms

The thresholds for the normalization are defined on the basis of empirical data but they can easily be modified according to distinct projects specifications. This mapping defines the general quality model. We used the same mapping for all metrics, but this just the standard setting and editable.

Beside the described adoption of well known metrics we used a metric for the analysis of additional features in every executable product after each iteration: the Running Tested Features Metric (RTF). The RTF Metric was introduced in [11] and it is defined as the number of features which passed their acceptance test. In this way RTF delivers a possibility to quantitatively evaluate different iteration according to their implemented functionality.

Subsuming we can classify the metrics in three categories:

- Method-level (e.g. Number of Parameters, Lines of Code)
- Class-level (e.g. Coupling between Objects, Number of Children, Depth of Inheritance Tree)
- Package-level (e.g. Cycle Count of Dependency Graph)

In agile software development and especially in the refactoring the early observation of quality is essential to keep the software stable throughout the evolutionary development process. Due to the specific characteristic of each distinct software development project it is not useful to define strict thresholds. Instead it is important to keep the limits and the interpretation of the measurement values as simple and flexible as possible.

It is very important to accommodate this requirement in the realization of metrics for agile software development.

3. “UnitMetrics” – Measurement Tool

To support agile software development at the origin we decided to implement the tool as a Plug-In for an Integrated Development Environment (IDE). We choose Eclipse because of the preconditions and its open source character [12]. Especially because of the many iterations’ of a product until it reaches final status the integration in a development environment instead of a stand-alone realization is recommendable.

The project was realized as an open source project and published under sourceforge.net [13]. Since August 2007 the plug-in was downloaded over 100 times but empirical information about the usage of the tool is not yet available.

The general goal was to create a measurement tool which expands the Eclipse-Development-Environment to provide source-code analysis by the use of appropriate metrics.

Four major features should be supported:
- Continuous analysis
- Fundamental interpretation
- Interactive visualization
- Extensibility

Besides the described features and characteristics the UnitMetrics Plug-In provides four major functionalities: presentation of metric values, normalized metric values and mapping according to threshold of distinct quality models, visualization of package and type dependencies, and package distance diagram.

In Figure 3 one can see the measured values for a chosen snapshot. The metrics are calculated at method level and without normalization. The measurement value table presents the core measurement results without any interpretation or normalization.

One has to recognize that without normalization the interpretation is very difficult due to less clarity.

To give the user a more graphical representation of the normalized values we established a “star” concept where three stars means excellent and zero stars means worst (in half star steps according to the mapping in table 1).

Figure 4 shows the same results as shown in Figure 3 in a normalized way by using the “star” concept. This normalization and mapping using the thresholds
of a distinct quality model for the specific project provides more clarity and a better presentation of measurement results.

Another important set of views is represented by dependence graphs [8] [6]. A dependence graph displays the coupling between different elements. The UnitMetrics tool provides the visualization on type-(class) - and package-level.

Dependence graphs allow to display the netting of coupling and to identify strong components. A strong
component contains all elements which are connected into a loop. Because of the fact that a goal of agile software development is to avoid loops or circles, identification is very useful.

Figure 5 shows an example of a dependence graph on package level. As one can see nearly every package is connected to every other package which manifests a loop-problem.

Loop-problem means that exist a connection from an origin to every other node by using just two paths. In this case the chosen architecture seems to contain substantial problems.

Figure 5: Dependence graph on Package-Level

Figure 6 shows the dependences on type-level and allows the further analysis of the discovered loops. At this point it stands out that the loops are not limited to interfaces (nodes marked with an “I”) but also obtains other components.

In both views strong components are illustrated by black bordered colored planes. For a more clear view the strong components were highlighted.

Both views possess manifold possibilities to analyze the netting of dependencies (e.g. fade in and fade out of distinct node, selection of nodes according to the distance of a specific node).

Another important visualization is the distance to the main sequence of specific packages in the measurement values [15]. This Package Distance Diagram is shown in Figure 7. To increase the interpretability standard deviation and the double standard deviation is marked with dashed lines.

The distance of a package represents the balance between instability and abstractness. Packages approaching one are either abstract and stable (Abstractness=1; Instability=0) or concrete and instable (Abstractness=0; Instability=1). While the first case indicates a lack of design integrity, the second case represents useless design [15]. The goal is to create a composition which is as stable as abstract.

Figure 6: Dependence graph on Type-Level

Figure 7: Package Distance Diagram
4 Usage of the Package Distance Diagram to identify refactoring candidates

The following section describes the usage of the package distance diagram to identify refactoring candidates within an agile software development process.

According to the measured value one can determine four classes of packages distributed over four areas in our Package Distance Diagram.

Area I subsumizes packages with nearly no external coupling but also very abstract in comparison to the main sequence.

That means that such packages are often only stubs which need a lot additional work to enhance package design and to add functionality by creating more concrete classes.

In Area II packages are more instable and still very abstract. That the abstractness is still high means that the lack of design is still inherent.

Packages in Area III bears mostly concrete classes but the high instability indicates that the design needs a lot of improvement.

Area IV contains packages with low abstractness and stability which gives reason to believes that the development of these classes is close to the finish.

In order to improve the program a continuous process has to take place. Therefore all packages necessary for the project have to be identified and unnecessary packages have to be deleted out from the project, then the refactoring of all other packages achieves a state of low instability and low abstractness for the whole project. This is of course the goal of every good developer. All packages will achieve Area IV after a certain amount of time. In order to use the four areas again the distribution of all areas will be renewed. The original square is now replaced by a rectangle with the bounds derive from the package with the highest abstractness and the highest instability.

Each individual area is now simply one of the four quadrants labelled clockwise. The meanings of the area will then slightly change. In area I all packages are still the most abstract packages, but due to the new upper bound of abstractness it is possible that these packages are still very important for the project. The same applies for area II, combined with the instability. In order to improve the overall project all packages in area III are considered for refactoring. Changes in the package will raise the value for instability, but it the proposed process is still applicable and after some iteration the result will satisfy the needs of the project.

To get more appropriate results single packages can be manually deleted. This applies for packages where no source code is available or it is a off the shelf commercial component. Then the boundaries of the rectangle can shift towards a narrow look on interesting packages.

The approach will lead in the end to a project where all packages have the value 0 for abstractness and instability. In larger projects this is of course unlikely. Therefore a programming manager needs to determine the best achievable value for abstractness and instability. A value close to zero means of course more work for the project. For the final presentation to a customer a smaller value for abstractness and instability can then present a stable and concrete program, with an object measurement to prove this conclusion.

5 Conclusions and Future Work

This paper presents an approach to support refactoring in agile software development by the use of software measurement.

The selection of the metric is also important to the outcome of the refactoring process. Future implementations may also include different types of metrics, e.g. a resource utilization metric, to improve the stability and speed of the program.

An empirical analysis about the usage and impacts of the tool is useful to identify the influence of
measurement values for the agile software development process.

The existing implementation enhanced the Eclipse IDE to support agile software development. Besides Eclipse, there are some other widespread IDE’s. On the basis of our reference implementation one can simply adapt the tool to other IDE’s.

At the moment the tool supports only Java as a programming language. The enhancement to other programming languages can increase practical benefit.

References


A Decision Making Tool for the Selection of Service Oriented-Based Legacy Systems Modernization Strategies

Rafik A. Salama, Sherif G. Aly
The American University in Cairo
raamir@aucegypt.edu, sgamal@aucegypt.edu

Abstract
In this article, we present a process and tool that facilitates the decision making process of selecting an appropriate migration strategy for legacy software systems using the Service Oriented Architecture. Issues related to legacy system migration were initially discussed, followed by two famous techniques namely the Service-Oriented Migration and Reuse Technique of the Department of Defense, as well as that of Erradi. A hybrid process was then introduced, and a tool following the process was illustrated. The tool takes into consideration choosing key migrating evaluation factors, rating the relative importance of such factors, inputting the organizational significance of each factor, as well as operating on individual system components.

Keywords: Legacy Software, Modernization, Service Oriented Architecture, Decision Making, Tool

1. Introduction

Legacy Information Systems are termed as such since they mainly use old technologies and architectures that became obsolete relative to the current state of art in technology. They are mainly mission critical systems, and their failure can have a serious impact on business.

In fact, a Legacy Information System can be defined as “Any information system that significantly resists modification and evolution” [1]. They can cause host organizations several problems:

- Legacy Information Systems usually run on obsolete hardware that is slow and expensive to maintain.
- Software maintenance can also be expensive, due to the lack of documentation, understanding of system details, and time consumption for tracing faults.
- The lack of proper and current interfaces makes the task of integrating Legacy Information Systems with other systems a difficult task. Legacy Information Systems are also difficult, if not impossible, to extend.
- A legacy information system represents a massive, long-term business investment. Unfortunately, such systems are mostly brittle in the sense of being resistant to change, are slow, and non extensible when faced with new technologies [2].

Although the above problems with the current legacy systems exist, such systems represent organizational assets that are virtually important for business continuation, so they need to be evolved to either match the continuously evolving business or to cope with continuous evolution of the other systems around them.

Much research has been done in the domain of legacy system analysis and modernization. Reverse engineering research of legacy systems has been discussed in [10-14]. Design discovery of legacy systems was also tackled in [15-17]. The work in [18] also tackled the issue of design metrics in legacy systems.

The continuous emergence of new technologies that are proven to be more robust, scalable, and maintainable, such as those using the Service Oriented Architectures (SOA), is currently attracting numerous legacy systems modernization stakeholders. The selection process of the SOA modernization strategy that best fits a given legacy system has been proposed previously in multiple researches. The most comprehensive process is the process of the Department of Defense (DoD) termed the Service-Oriented Migration and Reuse Technique (SMART) [3]. Another work is that of Erradi [4]. Erradi’s contribution mainly presents an assessment process followed by a decision framework for strategy selection.

The Erradi and SMART processes agree in the main steps, except for the fact that SMART ends by designing the strategy that best fits the application being modernized while the Erradi process ends by decision making for selecting the best strategy out of the existing migration strategies.
Modernizing a legacy system to the Service Oriented Architecture, as for example to Web Services, is somehow a straightforward process since the SOA offers a promise to enable and expose the existing legacy functionality in the form of services presumably without making any changes in the legacy system using the wrapper design pattern [5,6].

Although migration to the Service Oriented Architecture is mostly a straightforward process, it is sometimes not particularly as such. In some extreme cases, the SOA modernization of legacy systems involve total reengineering of the system [8] or even migration to new platforms [9]. The Service Oriented Architecture however still shapes the hope for the modernization of legacy systems in the form of web services with minimum change involved as in the case of componentization [7]. In all cases, the SOA has attracted the focus of major organizations to modernize their legacy systems using a variety of modernization techniques that fit their current architecture and the desired architecture.

In this article, we focus on providing a tool that assists the selection of a suitable hybrid modernization strategy from the current catalog of modernization techniques while taking into consideration the modernization business drivers, the quo state, the post modernization state of the system, and finally the boundaries of the organizations in terms of cost and duration (effort and resources).

2. The SMART and Erradi Strategies

One of the most comprehensive processes of legacy systems modernization is the Service-Oriented Migration and Reuse Technique (SMART) [3]. The word migration is used here to express the idea of modernization in the Department of Defense (DOD). This technique, in summary, involves:

1. Understanding the target SOA requirements.
2. Analyzing the current system, and performing a gap analysis between the target system and the current one.
3. Developing a migration strategy.

The Erradi technique on the other hand explores the organizational factors that influence the decision of strategy selection, but they assess the system using a simple paper and pencil matrix as follows:

1. **Legacy modernization business context:** Identify the key business goals and the key pains in the current system and requirement from the future system in terms of performance and reliability.
2. **Legacy portfolio assessment:** Assess the current system portfolio in terms of size, interfaces with other systems and technologies used. Manual and automated techniques should be used to handle such step.
3. **Choosing of legacy modernization options using scoring and pair-wise comparison models:** Using the below matrix as in Figure 1, the legacy system strategy is chosen according the best matching between the factors given through the assessment phase and the context of the modernization.

![Figure 1: Matrix of SOA Modernization Strategy Evaluations][4].

The Erradi model however lacks considering the organizational factor weights in the decision, as well as lacking considering the legacy system as a set of components rather than one big system that should be modernized using the one strategy fits all approach.

3. A Hybrid Decision Framework

We enhanced the decision making process of the selection of appropriate modernization strategies starting from the work of Erradi and guided by the work of SMART. Hence, we started from the decision framework proposed by Erradi and
designed a new selection process that uses Erradi’s work and complemented it using the SMART process. We called this process the Modernization Strategy Selection Framework (MSSF).

The final evaluation of the strategies is typically achieved given the following parameters (which flows directly from the assessment process):

- Factors contributing to the decision and their rates versus each strategy.
- User input to give the key important characteristics of the decision and their relative importance.
- User input to rate the importance of each characteristic contributing to the decision, which basically flows from the gap analysis.
- Legacy system/component source code which is used to measure code quality. These characteristics affect all the relevant characteristics of the decision as defined by the organization.
- The estimated cost of each strategy to give a rough cost benefit analysis.

Using the above parameters, we designed a process that automatically uses this information to give a general guidance for the strategies that should be followed to modernize the legacy system.

The decision making process mainly relies on decision theory specifically the Multi Criteria Decision Making (MCDM) tools to calculate the value of each strategy. It mainly aims on using all the available information to automatically order the strategies using decision theory. The aim of these ordered strategies is to give the organization a guidance of the best strategies to follow in their modernization process.

4. The Decision Making Tool

The decision making tool is mainly based on the enhanced decision framework based on the work of Erradi and SMART. The tool helps the organization through the whole process either using the artifacts created or by helping them input the values of some artifacts on the tool itself. The real contributions of the tool are:

- Guides the SOA modernization strategy selection framework through the mapping between the needed artifacts by the tool and output artifacts from the process.
- Automates the source code analysis for code quality metrics while forming a questionnaire for the rest of legacy system characteristics. The current version allows only for C++ code, although the code quality characteristics could be fed into the tool manually. The source code analysis is currently been done using the CCCC API [12].
- Automates the decision making process of the framework over a component-service scope through an easy to follow wizard.

The tool is a wizard application walking the organization through our combined migration selection strategy using the artifacts and executing the decision making process at the end. The wizard mainly ends by a report explaining the output of each step in the decision making process. The wizard screens is a one to one mapping to the decision making process. The screen flow is shown in Figure 2.

![Figure 2: Screen Flow of the Tool](image-url)
The decision making process only involves the user in only two input steps, while the rest of the evaluation is done automatically by the tool.

5. Screen Descriptions

The following subsections explain in more detail the various steps in the strategy selection process inherent in the tool.

5.1 Welcome Screen

This the first screen of the application as shown in Figure 3 mainly aims at introducing the wizard to the user to expect what is next and explain the requirements of each screen. This screen also is the point where the application loads the artifact files and validates them.

![Figure 3: The Informational Welcome Screen](image)

5.2. Choosing the Key Evaluation Factors

This is the screen corresponding to the decision making process mainly concerned with selecting key evaluation characteristics, which will affect two steps of the decision making process, namely:

- **Filtration**: The filtration step is an important step where the user gets the opportunity to filter the modernization strategies based on important key characteristics. For example, the organization can choose a certain modernization driver to be used as a filtering criteria where all the strategies that fall in this criterion are ignored.

- **Significance**: By default all the key characteristics gets a double significance to favor the strategy that has higher rate for these characteristics. The screen is shown in Figure 4.

![Figure 4: Choosing Key Evaluation Characteristics Screen](image)

5.3 Rating Relative Importance

This screen as shown in Figure 5 is intended to mainly rate the selected key characteristics relative to each other. The idea is to simply say which one of the key evaluation characteristics is more important than the other. The reason behind this is to give the organization a control over ordering the priority of these keys or to even give them equal priority. This will allow the focus to be more on the higher priority characteristics rather than the lower ones during the filtration step.
5.4 Input Significance of Each Factor

This is the step added to the process of Erradi which gives the organization more control over the organizational variables using the result of the gap analysis. The gap analysis will indicate how far the current system is from achieving such characteristics. The significance of the variable doesn’t necessarily indicate how large the gap is, but the organization may choose to neglect its effect when modernizing this component to the service. We also added the value highly significant that is basically used to double the effect of a certain characteristic, which will give it more emphasis over the other characteristics. The screen is shown in Figure 6.

5.5 Choosing the Source Code

This is the step where the tool allows for the selection of the source code of the component in the legacy system that is currently being evaluated. The source code selected is measured using the CCCC API. This is a recurrent screen in the sense that this screen is being used each time a component for evaluation is considered. The organization may want to consider a hybrid modernization approach, so it should evaluate each component vs. service in a separate thread to focus on its criteria’s values alone.

5.6 Inputting Legacy System Characteristics

This is the screen where the organization gets to input the result of their legacy system assessment. These are the values that are not involved in the code quality metrics or that are involved but the tool does not provide them. The screen is shown in Figure 7.

5.7 Viewing the Evaluation Report

This is the final screen of the tool, as shown in Figure 8, where the organization gets to see a report of the process steps output as follows:

- First, the organization sees the output of the filtration process indicating the rates of the strategies before the filtration and the resulting filtered strategies.
- Second, the organization sees the output of the assessment process in the form of an ordered list of the strategies giving the organization more benefits based on the following:
  - Rates of the characteristics
  - Code Quality Metrics
  - Significance of the characteristics

- Finally, the cost benefit analysis shows the result of the automatic cost benefit analysis using the benefits computed and the cost provided. The output is also an ordered list of the strategies based on the cost benefit analysis.

![Your Evaluation Report](image)

**Figure 8**: Final Report and Decision made by the Tool

6. Conclusion

In this article, we presented a process and tool for assisting the decision making process of selecting appropriate migration strategies of legacy software systems using the Service Oriented Architecture. We discussed two key approaches for legacy software systems migration, presented a hybrid approach, and subsequently illustrated the tool that facilitates this decision making process. The tool takes into consideration key organizational factors deemed important to the migration process, as well as each factor’s significance.

7. References


A Library-Based Approach to Translating OCL Constraints to JML Assertions for Runtime Checking

Carmen Avila, Guillermo Flores, Jr., and Yoonsik Cheon
Department of Computer Science
The University of Texas at El Paso
El Paso, TX, USA
{ceavila3, gflores3}@miners.utep.edu, ycheon@utep.edu

Abstract – OCL is a formal notation to specify constraints on UML models that cannot otherwise be expressed by diagrammatic notations such as class diagrams. Using OCL one can document detailed design decisions and choices along with the behavior, e.g., class invariants and method pre and postconditions. However, OCL constraints cannot be directly executed and checked at runtime by an implementation, thus constraint violations may not be detected or noticed, causing many potential development and maintenance problems. In this paper we propose an approach to checking OCL constraints at runtime by translating them to executable JML assertions. The key components of our approach are a set of JML library classes, use of model variables, and a separation of JML assertions from source code. The library classes implement OCL collection types and facilitate a direct mapping from OCL constraints to JML assertions by using model variables. The translated JML assertions are stored in specification files, separate from source code files, to ease change management of OCL constraints and Java source code. Our approach also facilitates a seamless transition from OCL-based designs to Java implementations.

Keywords: class invariant, pre and postconditions, assertion, runtime assertion checking, object-oriented class library, OCL, JML

1 Introduction

A UML diagram such as a class diagram cannot express a rich semantics of an application being modeled [1]. There is a need for describing additional constraints on the objects and entities present in the model. The Object Constraint Language (OCL) is a textual, formal specification language for specifying the semantics of UML models [13]; OCL specifications are commonly referred to as constraints. Using OCL, for example, one can specify the behavior of a class by writing, among other things, class invariants and method pre and postconditions.

As a design notation, however, OCL is not executable, and OCL constraints are not reified to implementation artifacts. This may lead to many problems in development and maintenance, such as inconsistency. For example, as design constraints are not explicitly expressed in source code, a change to source code that causes a drift or deviation from the initial design may not be detected or noticed by the developer.

In this paper we advocate runtime assertion checking as a partial solution to the problem of design drift. We propose to reify OCL constraints to source code in a form that can be executed and checked at run-time. Specifically we translate OCL constraints to executable assertions written in JML. JML is a formal behavioral interface specification language for Java [9], and a significant subset of it can be checked at runtime [1] [4] (see Section 2.2). Assertions translated from OCL constraints can detect violations of design constraints, thus design drifts at run-time. They also provide excellent API documents that are precise and kept synchronized with the implementation. In addition, as evidenced by a recent introduction of the assert statement to the Java language, assertions are recognized as a practical programming tool and are said to be most effective when they are generated from formal specifications such as OCL constraints.

For the translation we use a set of immutable library classes that implement the collection types defined in the OCL standard library [13]. The use of library classes makes the translation intuitive and traceable, as most OCL constraints are directly mapped to the corresponding JML assertions. We also expect the use of library classes facilitate automation of the translation. Another feature of our approach is the way we organize translated assertions. Instead of embedding them directly to source code, we store them in separate specification files; the JML compiler does an appropriate weaving by combining the JML specification files with Java source code files [4]. This organization facilitates change management of both OCL constraints and Java source code; e.g., changes to OCL constraints can be automatically propagated to JML assertions by retranslating or regenerating the JML specification files, and thus having a minimal impact on the implementation, i.e., Java source code. Our approach is facilitated by several language and tool features of JML, in particular, specification-only variables called model variables [5] and specification refinement (see Section 4).
There is a few previous work done on runtime assurance of OCL constraints by translating them to programming languages [1] [14]. As in our approach, the common theme here was to define a set of OCL library classes for the translation. However, OCL constraints are typically translated into source code, i.e., a sequence of program statements, not into assertions or annotations; therefore, they cannot be used as source code-level documents, e.g., precise API specifications.

An assertion is a predicate placed in a program to indicate the truth of the assertion at that place [8] [12]. It is used to specify and reason about the correctness of a program both statically, as in Hoare-style pre and post-conditions [8], and dynamically, as in Design by Contract [11] and assert macros or statements [12]. Surprisingly, however, there is not much work done for translating OCL to executable (JML) assertions. One exception is the work of Hamie, which inspired our own work. Hamie suggested translating OCL constraints to JML assertions by defining a mapping for OCL operators [7]. The operators of OCL collections types are mapped directly or indirectly to the methods of JML’s collection model types that implement various kinds of container abstractions, such as sets, bags, and sequences. However, the organization, structures, and vocabulary of JML collection types are somewhat different from those of OCL, and it is unclear how this mapping is to be refined into implementation artifacts, e.g., JML assertions directly referring to program variables. The use of both model variables and immutable collection classes proposed in our approach will greatly simplify the implementation of the mapping and also result in a clear and intuitive mapping. For the development of an automated translation tool, we plan to adapt and refine his mapping.

The remainder of this paper is organized as follows. In Section 2 we briefly review OCL and JML through a small example that we will use throughout this paper. In Section 3 we explain the problem of translating OCL constraints to assertions by referring to program states or variables. In Section 4 we describe our approach by applying it to our running example. We also discuss how our approach solves the problems described in Section 3. In Section 5 we mention our on-going evaluation effort, which is followed by a concluding remark in Section 6.

## 2 Background

### 2.1 OCL

The Object Constraint Language (OCL) [13] is a text-based, formal specification language extension to UML for specifying constraints or behaviors of UML models that cannot otherwise be expressed by diagrammatic notation. It supplements UML by providing concise and precise expressions that have neither the ambiguities of natural language nor the inherent difficulty of using complex mathematics. As an example, consider the class diagram shown in Figure 1 that depicts different types of banking transactions along with associated accounts.

![Figure 1. Sample UML class diagram](image)

Different types of transactions and accounts are organized into two class hierarchies, rooted by an abstract class and an interface, Transaction and Account, respectively. The diagram also shows two concrete classes. The class CheckPostingTransaction models a transaction that posts a check to an account. Suppose that a check can be posted to only one checking account and that the account should have enough balance to cover the check to be posted. Then, this constraint can be precisely documented in OCL by writing following statements.

```plaintext
context CheckPostingTransaction inv:
  self.accounts->size() = 1 and
  self.accounts->forAll(c: Account | c.isKindOf(CheckingAccount))

context CheckPostingTransaction inv:
  self.accounts->forAll(c: Account | c.getBalance() >= self.amount)
```

As shown, each OCL constraint is preceded by a context specification that identifies the UML model being constrained, in this case CheckPostingTransaction. As indicated by the keyword inv, both statements specify class invariants; the first constraint statement states that a CheckPostingTransaction object should be associated with only one CheckingAccount object, and the second statement states that the associated account should have enough balance to cover the check being posted. Note that due to its multiplicity, the account aggregation is viewed as a collection, and thus we can use collection operations such as size and forAll (see below).

OCL comes with several primitives types such as Integer, Real, Boolean, and String and collection types such as Collection, Set, OrderedSet, Bag, and Sequence [13]. In the example constraints above, we used collection operations such as size and forAll; the size operation returns the number of elements contained in a collection, and the forAll operation tests whether an expression is true for all objects of a given collection.

### 2.2 JML

The Java Modeling Language (JML) is an interface specification language for Java to formally document the
behavior of Java program modules such as classes and interfaces [9]. JML specifications or assertions can be added directly to source code as a special kind of comments called *annotation comments*, or they can reside in separate specification files. In JML, the behavior of a Java class is specified by writing, among others, class invariants and pre and postconditions for the methods exported by the class. The assertions in class invariants and method pre and postconditions are usually written in a form that can be compiled, so that their violations can be detected at runtime [1] [4].

```java
// File: CheckingAccount.jml
public class CheckingAccount {
    spec_public private int bal;
    public invariant bal >= 0;
    requires amt > 0 && amt <= bal;
    assignable bal;
    ensures bal == old(bal) + amt;
    public void postCheck(int amt);
    // the rest of definition
}
```

**Figure 2. Sample JML specification**

Figure 2 shows a sample JML specification written in a separate specification (.jml) file. It describes the behavior of class **SavingAccount**. The JML keyword **spec_public** states that the private field **bal** is treated as public for specification purpose; e.g., it can be used in the specifications of public methods such as **postCheck**. As shown in the example, a method specification precedes the declaration of the method. The **requires** clause specifies the precondition, the **assignable** clause specifies the frame condition, and the **ensures** clause specifies the postcondition. The JML keyword **old** in the postcondition denotes the pre-state value of its expression; it is most commonly used in the specification of a mutation method such as **postCheck** that changes the state of an object.

JML supports several features that make it an ideal language to explore our idea of checking OCL constraint at runtime by translating them to executable assertions. As a Design by Contract (DBC) [11] language for Java, it supports class invariants and method pre and postconditions as built-in language features. It combines the practicality of DBC language with the expressiveness and formality of model-oriented specification languages; its powerful assertion language such as various forms of quantifiers will allow us to translate any OCL constraint into a JML assertion. In addition, the vocabulary for writing assertions can be tuned and enriched by add specification-purpose library classes; this is supported by the **model import** clause (refer to Section 4.1 for an example).

JML allows one to write assertions in terms of abstract values provided by model variables [5] (see Section 4.1). There are at least two advantages to writing specifications with abstract values instead of directly using Java variables and data structures. The first is that by using abstract values the specification does not have to be changed when the particular data structure used in the program is changed. Second, it allows the specification to be written even when there are no implementation data structures available.

As shown in the example above, JML assertions can be written in a separate specification file. This not only facilitates the propagation of changes from OCL constraints to automatically-generated JML assertions but also allows one to check OCL constraints even if no Java source code files are available. This also has a practical value because one can ship the object code for a class library to customers, sending the JML specifications but not the source code. Customers would then have documentation that is precise, unambiguous, but overly specific. Customers would not have the code, protecting proprietary rights. In addition, customers would not rely on details of the implementation of the library that they might otherwise glean from the code, easing the process of improving the code in future releases.

JML supports the notion of specification refinement for associating multiple specification files to the same source code file or bytecode file (see Section 4). This will allow us to easily add and maintain automatically-generated assertions (from OCL constraints) and manually-written assertions for the same class.

## 3 The Problem

We translate OCL constraints to executable JML assertions to recognize inconsistencies between a UML design model and its implementation during the development phase and also to detect design drifts during the maintenance phase. The big question then is to translate an OCL constraint to a corresponding JML assertion. As assertions are generally written in terms of program states, we first need to find an appropriate mapping from OCL modeling elements, e.g., the **accounts** aggregation in the **Transaction** class, to their representations, e.g., program states or variables, in the implementation classes. As an example, let us consider the **CheckPostingTransaction** class and its OCL constraints from Section 2.1, and translate them to a Java implementation annotated with JML assertions. Figure 3 shows one such an implementation where JML assertions are directly embedded into the source code as annotation comments (i.e., // and /* @…@ */). The **accounts** aggregation of its superclass, **Transaction**, is reified into a JDK set (java.util.Set) with its multiplicity expressed as a class invariant. As easily guessed, our example OCL constraints are also translated into JML invariants. Note that except for a small notational difference and the use of a universal quantifier (**forall**) in place of OCL’s **forAll** operation, the JML assertions are direct translations of the OCL constraints reflecting their structures.
files (see Figure 4). The introduction of OCL-like library classes to JML enables us to map OCL constraints to JML assertions in a one-to-one fashion by preserving the original structures and using almost the same vocabulary. The specific technique is to write JML assertions not in terms of Java program states, i.e., program variables, but in terms of their abstractions using the library classes. In JML, this abstraction is called a *model variable* [5]. A model variable is different from a Java program variable in two aspects. First, it is a specification-only variable meaning that it can be referred to only in assertions, but not in program code. Second, its value is not directly manipulated using assignment statements but is given implicitly as a mapping from a program state, called an *abstraction function* (see Section 4.1 below for an example). In summary, for a UML modeling element such as an aggregation, we introduce a JML model variable of an appropriate type and translate OCL constraints written in terms of the UML element into JML assertions written in terms of the corresponding model variable. In the following subsection, we will illustrate our approach in detail by using our running example.

![Figure 3. Java implementation with JML annotations](image)

What is wrong with the above translation and resulting assertions? Although at first it looks fine for this particular example, there are several potential problems with such a translation using program variables (i.e., the `accounts` field) in assertions and adding annotations directly to source code. The main problem is that it may not be always possible to find an appropriate, direct mapping between OCL models and Java representations; e.g., what if the `accounts` aggregation is implemented as an array? There may be no corresponding Java vocabulary for the OCL terms used in the constraints. In general, OCL constraints have to be recast into the vocabulary defined by a particular choice of representations, e.g., sets, lists, or arrays. The translation is not only hard but also results in assertions with structures different from those of the OCL constraints. Such assertions tend to be lengthy, hard to read and understand, and difficult to be traced back to the original OCL constraints. The translation itself is less amenable to automation.

Worst of all, the translation doesn’t accommodate evolution or maintenance of both OCL constraints and Java programs. For example, what happens if the representation becomes changed, e.g., from sets to arrays? The whole JML assertions might have to be rewritten in terms of the vocabulary given by the new representation, i.e., arrays. Similarly, it is also difficult to propagate changes of OCL constraints to the corresponding JML assertions embedded in the source code. Embedding JML assertions directly into the source code also aggravates the problem because it hinders automated tool support for change propagations in both directions.

## 4 Our Approach

The key idea of our approach is to introduce a new JML library that implements the standard OCL library such as collection types and to store the translated assertions in JML specification files, separately from Java source code files (see Figure 4). The introduction of OCL-like library classes to JML enables us to map OCL constraints to JML assertions in a one-to-one fashion by preserving the original structures and using almost the same vocabulary. The specific technique is to write JML assertions not in terms of Java program states, i.e., program variables, but in terms of their abstractions using the library classes. In JML, this abstraction is called a *model variable* [5]. A model variable is different from a Java program variable in two aspects. First, it is a specification-only variable meaning that it can be referred to only in assertions, but not in program code. Second, its value is not directly manipulated using assignment statements but is given implicitly as a mapping from a program state, called an *abstraction function* (see Section 4.1 below for an example). In summary, for a UML modeling element such as an aggregation, we introduce a JML model variable of an appropriate type and translate OCL constraints written in terms of the UML element into JML assertions written in terms of the corresponding model variable. In the following subsection, we will illustrate our approach in detail by using our running example.

![Figure 4. Approach to translating OCL into JML](image)

### 4.1 Illustration

Let us apply our approach to the transaction classes that we have been playing with. Remember that the abstract class `Transaction` has an aggregation named `accounts`, representing the set of accounts involved in a transaction (see Figure 1), and both of the OCL constraints are written in terms of this aggregation. As shown in Figure 5 below, we introduce a JML model variable for this aggregation. The model variable has the same name as that of the aggregation and is of type `OclSet`. The `OclSet` class is from our new JML library and implements OCL’s set. The rest of the specifications are identical to the previous one except for renaming of the method to follow the OCL’s naming convention.

---

1 We expect that a significant portion of the translation can be automated, and we have a plan for developing such an automated translation tool.
How is the value of a model variable such as accounts defined? In other words, how can the assertions written in terms of a model variable can be checked at runtime? For a model variable to be executable, it should be provided with a so-called abstraction function that specifies its value as a mapping from a program state, i.e., program variables [5]. For example, the abstraction function for the model variables accounts can be specified in the source code of class Transaction as follows.

```java
// File: Transaction.java
//@ refine "Transaction.jml";
public abstract class Transaction {
    // File: Transaction.jml
    model import ocljml.OclSet;
    public abstract class Transaction {
        spec_public protected int amount;
        public model OclSet<Account> accounts;
        public invariant accounts.size() > 0;
    }

    // File: CheckPostingTransaction.jml
    public class CheckPostingTransaction extends Transaction {
        public invariant accounts.size() == 1
            (forall a: Account; accounts.includes(a);
            a instanceof CheckingAccount);
        public invariant (forall a: Account; accounts.includes(a);
            a.getBalance() == amount);
    }

    Figure 5. JML specifications from OCL constraints
```

The @refine directive states that this file refines the given JML specification file, thus inheriting all its assertions such as class invariants and method specifications. The abstract function is specified using the @refines clause. It maps the array representation (accountsRep) to a set abstraction (accounts). The static method convertFrom creates an OclSet object from an array. The in clause specifies a so-called data group [10] and states that any method that can modify the model variable accounts can also modify the program variable accountsRep. In addition to the abstraction function, additional implementation invariants (e.g., no duplicates) can also be specified in terms of the representation variables.

How does our approach solve the problems associated with translating OCL constraints into JML by referring to program variables? Note that even if the accounts aggregation is represented as an array, in JML assertions it is still viewed and manipulated as a set as in OCL constraints. Thus, our approach clearly alleviates the problems of readability, understandability, traceability, and translation automation, as OCL constraints are one-to-one mapped to JML assertions preserving the structures and also using almost the same vocabulary. Let’s next consider the problem of evolution and maintenance. Let’s first consider a change to the implementation, say the representation from an array to a tree. This change is localized, as all we need to do is to rewrite the @refines clause to define a new abstraction function for the tree. The rest of the specification, in particular, assertions translated from OCL constraints remain the same, as they were written in terms of the model variables. How about changes to OCL constraints? They also have a minimal impact and are localized in that we only need to rewrite the corresponding assertions in the specification files or, with automated translation, regenerate the whole specification files; i.e., there is no or little need to change the source code files.

### 4.2 JML Library for OCL Collection Types

We implemented in Java all collection types defined in the OCL standard library. Our classes are organized into a class hierarchy with an abstract class OclCollection at the root; other collection classes include OclSet, OclOrderedSet, OclBag, and OclSequence. Since our intention is to use them as JML model classes, all of them immutable; i.e., there is no method that can change the values of these classes. For each collection class, we implemented all the operations defined by OCL except for operations such as forAll (see below). In addition, we defined a set of conversion methods such as convertFrom\(1\) to convert Java arrays and collections to our implementation of OCL collection types.

In OCL, there are a number of collection operations called iterators that take OCL expressions as parameters and work on all elements of a collection. Operations such as select, reject, collect, forAll, and exists fall in this category. Because Java doesn’t support this kind of (higher-order) methods, no such methods are defined in our implementation. Instead, they are translated indirectly into JML expressions; e.g., operations such as forAll and exists are translated into JML quantifiers as done in our example.

### 5 Evaluation

Our implementation of OCL library classes as described in Section 4.2 has several limitations and notable features. First, as the current version of JML doesn’t support generics introduced in Java 1.5 [2] (refer to the JML website at http://www.jmlspecs.org), all the collection classes are implemented as so-called raw types. This works well for all the classes and methods except for the sum method of the Collection type. The sum method returns the sum of all the elements contained in the collection. The OCL standard states that each element of the collection must be of a type supporting the binary addition (+) operation and the return type must be the element type given as a type parameter [13]. This causes a trouble in our raw type implementation, OclCollection, as no type parameter is available denoting the element type. We can’t specify the exact return type and we can’t make any
assumptions about the elements. Our solution is to specify the most general type, i.e., Object, as the return type and check each element’s runtime type for the addition compatibility. Depending on the types of elements, the sum is returned as either a Long or Double object; if at least one element is not addition-compatible, then an IllegalStateException is thrown.

Second, as mentioned in Section 4.2, OCL defines a set of iterator operations such as select, reject, collect, forAll, and exists that take an OCL expression as a parameter. Because Java doesn’t yet support this kind of (higher-order) methods, no such methods are defined in our implementation. We believe that this problem can be solved when Java 1.7 supports a form of closure called a code block [6]. We also proposed to the JML developers to introduce a limited form of OCL-like iterators such as select, collect, and reject which, if adopted, will make the translations of these iterators more direct and natural.

Third, some of OCL collection types such as Set and Sequence define an equals method, and the method is overloaded in that it takes an argument of the same type. In our implementation, however, we followed the Java convention and overloaded the equals method; i.e., its argument type is the class Object, thus overriding the one inherited from the Object class.

Last, in addition to the methods defined in OCL, our implementation adds several new methods such as convertFrom to enable conversion from Java arrays and collections to OCL collections (see Section 4.1).

We noticed several deficiencies in OCL specifications of some of collection operations. For example, operations such as first and last of types Sequence and OrderedSet are partial in that they are defined only when the sequence or ordered set is not empty. However, a precondition asserting this fact, e.g., self->nonEmpty(), is missing from the standard [13]. The append, prepend, insertAt, and subOrderedSet of type OrderedSet also have missing preconditions, and the at method of types Sequence and OrderedSet have missing postconditions.

We are currently evaluating our approach through case studies. Our plan is to perform both quantitative and qualitative measurements to evaluate the effectiveness and efficiency of our approach. In particular, we are interested in knowing the percentage of OCL constraints that we can translate with our approach and the quality of the translated JML assertions. We will also measure the runtime efficiency of the translated assertions that use our new JML library classes. The secondary goal of our evaluation is to gain more insights on our approach, especially its support for and limitations on automation, prior to a full-blown development of an automated OCL-to-JML translation tool.

6 Conclusion

We proposed an approach to translating OCL constraints to JML assertions so that violations of design constraints can be detected at runtime. The key components of our approach are (1) new JML library classes implementing OCL collection types, (2) specification-only variables, called model variables, and (3) separation of specifications from source code. Although we are still evaluating our approach, we believe that our library-based approach will enhance the quality of the translated assertions, accommodate constraint and implementation changes rather than avoiding them, and support translation. Our approach will assist in coping with the plaguing problem of design-implementation inconsistencies, through runtime assurance.

Acknowledgement

Avila and Cheon’s work was supported in part by NSF under grants CNS-0509299 and CNS-0707874.

References

Automatic MDA (Model Driven Architecture) Transformations for Heterogeneous Embedded Systems

Woo Yeol Kim\(^1\), Hyun Seung Son\(^1\), Y. B. Park\(^2\), B. H. Park\(^3\), C. R. Carlson\(^4\), R. Young Chul Kim\(^1\)
\(^1\)Dept. of Computer & Information Comm., Hongik University, Korea
\(^2\)DanKook University, \(^3\)The Armed Forces Medical Command, Korea
\(^4\)Dept. of ITM, Illinois Institute of Technology, Chicago, USA
{john, son, bob}@hongik.ac.kr

Abstract - We adopt the MDA mechanism for embedded s/w development, which reduces the lifecycle of s/w development. In this paper, we propose the automatic MDA (Model Driven Architecture) transformations to develop the heterogeneous embedded software. We first model 'Target Independent Meta Model' (TIM) through Requirement analysis. With the automatic MDA transformations, then automatically produce some 'Target Specific Model's (TSMs) selecting the different OS APIs and/or different processors, and then possible generate 'Target Dependent Code' (TDC) such as Java, C++, or C per each specific TSM. As a result, it is possible to port a specific TDC into the target system. We show one example which illustrates the proposed approach.

Keywords: Model Driven Architecture, Embedded System, Small Unmanned Ground Vehicle, Unified Modeling Language

1 Introduction

Today’s demands for numerous diverse embedded systems are on the very rapidly and greatly increase in recent years. Some industrial software developers are now in progress about the huge complexity of embedded system. Our researches focus on automatic development tool for embedded software (such as design, model, code, and test) to develop the heterogeneous embedded systems. But it may be hard automatically to develop embedded software because the embedded software mechanism is dependent on the particular hardware system and also is just code oriented development [1]. To solve these problems, we propose the automatic MDA transformations mechanism [2] using a general meta-model for embedded s/w development. We suggest detail activities of lifecycle for embedded s/w development and refined multiple V-model on MDA [6]. This will be possible automatically to generate target dependent code per each of target specific models via target independent software. With this transformation, we can help automatically to reuse software by product (such as design, model, code, and test), and to reduce the lifecycle of heterogeneous embedded software development.

This paper is organized as follows: in section 2, we describe the embedded model driven architecture (MDA) and refined multiple V-model. In section 3, we show our automatic MDA transformations for develop embedded s/w system. In section 4, we show the modeling example of heterogeneous embedded systems used in this paper.

2 Embedded Model Driven Architecture

The original MDA[2] is the OMG proposed approach for system development. It primarily focuses on software development platform. The MDA is based on one meta-model describing the systems to be built. A system description is made of numerous models, each model representing a different level of abstraction. The modeled system can be deployed on one or more platforms via model to model transformations [7].

We mention to adopt MDA mechanism into embedded software development as follows.

2.1 Refined Multiple V-model on MDA

The original multiple V-model[8] is focused on developing (or modeling) a system based on a particular target domain. This is difficult to apply for other target domains. So, we attempt to refine multiple V-model on MDA, which solves problems of the original V-model.
In figure 1, it describes to adapt multiple V-model with MDA. The refined multiple V-model is also a development model process (Target Independent Model, Target Specific Model(s), Target Dependent Code(s)) which is developed the different versions of the heterogeneous systems. The first V-model is focused on the target independent model. The middle V-model is focused on the target specific model(s). The last V-model is focused on the target dependent code(s).

The refined one may usefully develop heterogeneous embedded systems with reusability on different target domains. Moreover it can also work parallel with both of s/w development process and test process. Due to these double processes, it may be possible to develop more safe and reliable software components.

So we develop the automatic tools for automatic MDA transformations, and code generations such as Java, C++, or C.

In figure 2.a,b,c, it is just more detail described to map diagrams at the first V-model. The use case diagram is used during requirement to represent the functionality of the system from the user’s point of view. During analysis, the class diagram describes the structure of the system. The concurrent message diagram and concurrent state diagram describe the internal concurrent behavior of the system during design.

It is necessary work to make the automatic transformation from TIM(Target Independent Model) phase to TSM(Target Specific Model) or from TSM(Target Specific Model) to TDC(Target Dependent Code). Therefore, it should make a reliable model of TIM at the first step. We don’t mention about next remaining phases in this paper.
Test Case Generation
for Architecture

2.2 MDA based Modeling Approach

We apply MDA to embedded software modeling approach. Our modeling approach consists of static modeling and dynamic modeling. Static model uses class diagram to represent the static aspect of a system. In dynamic model, concurrent message diagram and concurrent state diagram [11] represent the dynamic concurrent behavior of the system in figure 3.

2 MDA Transformation Process

Our defined MDA Transformation Process [5,10] consists of TIM (Target independent model) stage, TSM (Target specific model) stage, and TDC (Target dependent code) stage. TIM defines a general model independent of the particular target domains. After requirement analysis, we may design TIM with extended xUML[4,5,11] and UML profile[3] at this TIM stage.

TSM defines a specific model dependent of the OS, and hardware within the particular target domains. At this TSM stage, we can model more complete with additional functions on TSM automatically transformed with TIM. TDC defines the source code per the specific target system. At TDC stage we can automatically generate codes whatever we need such as Java, C++, and C. But we manually need to write a little part of the particular function codes that do automatically impossible. Finally we can port complete executable code(s) into the target system(s).

Our MDA Transformation Process consists of two transformation steps T1, T2(or T2’, T2’’) in figure 4.

The first transformation T1 is automatically transformed TIM (which merges with the predefined Processor profile and OS profile) into TSM. The second transformation T2 is automatically generated the right executable TDC per each different target system. Its possible generable code languages are C, C++, and Java. The code generating method is automatically generated using conversing text from model [6].

3.1 Transformation T1: Transforming from a TIM to TSM(s)

To transform TIM into TSM(s), we mention about four layers such as application layer, service layer, operating system layer, and processor layer. Each Layer is described as follows: Application Layer is the top layer for modeling a system. The user can do modeling with services on service layer. But we can not change the service name for T1 transformation[9].

Figure 5. The structure of four layers for embedded system development
In figure 5, there is mapping the behaviors of application layer into services of service layer. Then associate with operating system layer and processor layer. The user will possible develop a system having only knowledge of services without detail implementation. Operating System Layer consists of APIs of OS. Just work with context switch, scheduling, memory management, timer service of OS on the processor layer. The last processor layer is the bottom layer dependent of hardware.

3.2 Transformation T2: Transforming from TSM(s) to a TDC(s)

To transform the actually executable code from TSM through T1 transformations, we should do execute T2 transformations.

T2 is generated a language with meta-template model on the basic of the class diagram, concurrent message diagram, and concurrent state diagram of TSM. Figure 6 shows the meta template model for these diagrams.

<table>
<thead>
<tr>
<th>Class Name: string</th>
</tr>
</thead>
<tbody>
<tr>
<td>Package List: List</td>
</tr>
<tr>
<td>Parent List: List</td>
</tr>
<tr>
<td>Interface List: List</td>
</tr>
<tr>
<td>Association List: List</td>
</tr>
<tr>
<td>Attribute List: List</td>
</tr>
<tr>
<td>Function List: List</td>
</tr>
</tbody>
</table>

**Figure 6. Meta template model**

Meta template model consists of all elements for the actual code generation in figure 6. “Class Name” stores the strings of class name. “Package List” stores a type of linked list to include package or library. “Parent List” is the above one of current class, which stores a type of linked list. “Interface List” stores interface name. “Association List” also stores a type of linked list about association relationship between other classes. “Attribute List” also stores a type of linked list about the class attributes. “Function List” stores a type of linked list about the class methods.

After storing all information of diagrams through meta template model, it is transformed like figure 7. It shows that each of meta template names in figure 6 matches each code template in figure 7. As a result, we can execute T2 transformation automatically to generate each code with collecting information on Meta template model of TSM in figure 6.

4 The automatic MDA transformation tool

We would like to use one example of small unmanned ground vehicle system (SUGV) in figure 8.

**Figure 8. Small Unmanned Ground Vehicle System**

Table 1 describes the information comparison of two heterogeneous SUGV systems.
Table 1. Hardware information for SUGV Systems

<table>
<thead>
<tr>
<th></th>
<th>SUGV1</th>
<th>SUGV2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microcontroller</td>
<td>Ubicom SX48AC 20MHz</td>
<td>Hitachi H8/3292 16MHz</td>
</tr>
<tr>
<td>RAM</td>
<td>32 KByte</td>
<td>512KByte</td>
</tr>
<tr>
<td>EEPROM</td>
<td>32 KByte</td>
<td>16KByte</td>
</tr>
<tr>
<td>Sensor</td>
<td>Two ultrasonic sensors</td>
<td>Two light sensors</td>
</tr>
<tr>
<td>Display</td>
<td>Text LCD</td>
<td>Text LCD</td>
</tr>
<tr>
<td>Servo-motor</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>JVM</td>
<td>H/W</td>
<td>N/A</td>
</tr>
<tr>
<td>language</td>
<td>Java</td>
<td>C/C++</td>
</tr>
</tbody>
</table>

4.1 Transformation T1: Transforming from a TIM to TSM(s)

The first transformation T1 is automatically transformed TIM (which merges with the predefined Processor profile and OS profile) into TSM in Figure 9, 10. Figure 9 shows to choose Ubicom SX48AC and Javeline for SUGV1, then click the ‘Generate’ button to transform into one TSM. Figure 10 shows to choose Hitachi H8/3292 and brickOS for other SUGV2 then also click the ‘Generate’ button to transform into other TSM.

4.2 Transformation T2: Transforming from TSM(s) to TDC(s)

The second transformation T2 is automatically generated the right executable TDC per each different target system. Its possible generable code languages are C, C++, and Java.

Figure 11, 12 shows automatically to transform into code through code template after modeling a TIM. All transformations are automatically executed on the tool. In the case of SUG1, java code can be generated as clicking ‘Java’ button while C++ be generated in SUGV2.
On T2 transformation in Table 2, it shows Forward(), Backward(), Stop() functions of motor class. Motor class links to one other motor to work the servo motor. So, it rotates each opposite direction if the same code to move the left servo motor and the right servo motor. As a result, we can store the location status information with ‘m_chPosition’ variable. We show to compare the underline on three functions generated with the tool, and to transform into the right code for the heterogenous embedded systems.

Table 2. The comparison of codes on each SUGV

<table>
<thead>
<tr>
<th>SUGV1 (Javeline)</th>
<th>SUGV2 (MindStorm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>public void Forward()</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>if(&quot;left&quot; == m_chPosition)</td>
<td></td>
</tr>
<tr>
<td>CPU.pulseOut(STOP + m.speed, m_nPin);</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>CPU.pulseOut(STOP - m.speed, m_nPin);</td>
<td></td>
</tr>
<tr>
<td>m_status = &quot;forward&quot;;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>public void Stop()</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>public void Forward()</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>if(&quot;left&quot; == m_chPosition)</td>
<td></td>
</tr>
<tr>
<td>CPU.pulseOut(STOP + m.speed, m_nPin);</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>Motor.forward(m.speed);</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

5 Conclusion

It may be hard to reuse embedded software products (such as model, design, and code) for the hardware dependent systems, that is, heterogeneous embedded systems. We propose the automatic MDA (Model Driven Architecture) transformation to develop the heterogeneous systems. With this transformation method, we can possibly develop target independent model, then automatically generate target specific model(s), which also are generated target dependent code(s).

To solve a problem for embedded software development, we implement the automatic tools for model transformation and code generation.

As a result, we can lead to reduce the lifecycle of the heterogeneous embedded software development.

6 References

Story Cards Process Improvement Framework

Chetankumar Patel, Muthu Ramachandran
Innovation North, Faculty of Information and Technology
Leeds Metropolitan University, Leeds, UK, LS6 3QS.

Abstract – This paper describes an ongoing process to define a suitable process improvement model for story cards based requirement engineering process and practices at agile software development environments. Key features of the Story card Maturity Model are: solves the problems related to the story cards like requirements conflicts, missing requirements, ambiguous requirements, define standard structure of story cards, to address non-functional requirements from exploration phase, and the use of a simplified and tailored assessment method for story cards based requirements engineering practices based on the CMM, which is poorly addressed at CMM. CMM does not cover how the quality of the requirements engineering process should be secured or what activities should be present for the requirements engineering process to achieve a certain maturity level. It is difficult to know what is not addressed or what could be done to improve the process. We also presents how can be the identified areas of improvement from assessment can be mapped with best knowledge based story cards practices for agile software development environments.

Keywords: Software Process Improvement, Requirements Engineering, Story cards, Agile Requirements, User story.

1 Introduction

Requirements elicitation process is one of the challenging processes in the software development methods. In traditional software development methods end users or stakeholders predefined their requirements and sent to development team to analysis and negotiation to produce requirement specification. Traditional software development has a problem to deal with requirement change after careful analysis and negotiation. This problem is well tackled by the XP, which is one of the agile software development methodologies.

Extreme (XP) programming is a conceptual framework of practices and principles to develop software faster, incrementally and to produce satisfied customer. It is a set of twelve practices and four principles, which makes XP successful and well known among all the agile software development methods. The goal of XP is to produce the software faster, incrementally and to produce satisfied customer [1]. According to Bohem (1998) the cost of change grows exponentially as the project progresses through it lifecycle [31]. The relative repair cost is 200 times greater in the maintenance phase than if it is caught in the requirement phase [32]. XP maintain the cost of change through iterative software development methods and Refactoring.

While CMM and CMMI or software process improvement has gained a lot of attention during the last decade. Due to the increasing competition in the software market faster delivery, high quality products and customer satisfaction are the major concerns for software organisations. A quality process can have a positive impact on services, cost, on-time delivery, development technology, quality people and quality of products [16].

This paper presents an approach for simplified and tailored assessment method for story cards based requirements engineering process using CMM, and present how assessments can be performed on story cards based RE process.

1.1 Empirical Study of Story Cards Improvements Problems in ASD

Getting requirements on story cards right continues to be a universal problems same as the requirements problems in the traditional methodology. Story cards errors can be costly in terms of low time, lost revenue, loss of reputation and even survival [13].

A critical aspect of the requirements process is the selection of the an appropriate requirements set from the multitude of competing and conflicting expectation elicited from the various project stakeholders or from an onsite customers [33].

Looking at methods of CMM for process quality, measurement and improvement they tend to cover the area of requirements engineering poorly. It covers the area of requirements engineering inadequately. CMM does not cover how the quality of the requirements engineering process should be secured or what activities should be present for the requirements engineering process to achieve a certain maturity level.
Some time it is really difficult to assess the maturity of a requirements engineering process for a certain projects, and it is difficult to know what is not addressed or what could be done to improve the process.

As we discussed earlier agile software development methodology is the iterative software development methodology based on the story cards, for small to medium organisation and main objectives are lower cost, high productivity and customer. The CMM tends not to focus the software process on an organisation’s business objectives in their software process improvement programme [8].

The main thing is that CMM and ISO 9000 do not say much about requirements engineering and subsequently little about how the quality of the requirements engineering process should be maintained and ensured [14].

Herbsleb and Goldenson (1996) reported the use of the CMM in several software organisation. The study consistently showed significant organisational performance improvements that were directly associated with process maturity. The study also mentioned that the CMM improvement path is not always smooth, the efforts generally took longer and cost more than expected. While story card is an agile software developments practice. Agile software development methodology is targeted to lower cost. Some of the KPAs have been found difficult to apply in small projects [27]. This may be because CMM was originally structured for big enterprises [29]. CMM addresses practices such as document policies and procedure that large organisations need because of their size and management structure [27].

Normally story cards for agile software development do not support the heavy documentation at all and people communicate verbally on on-going basis. Unlike CMM, CMMI does not just focus on software process management; it also considers other department such as marketing, finance and purchasing [18]. So it could be seen unnecessarily complex, when it is applied to agile software development practices like Extreme programming, Scrum and lean development.

CMM can be described a common sense application of the process management and quality improvement concept to software development and maintenance but it focuses on software development and does not cover the requirements engineering process [2][3].

Without the standard for ensuring the quality of requirements engineering process, it is hard to ensure the result of the requirements engineering process. A consequence of this can be that requirements do not reflects the real needs of the customer of the system, requirements are inconsistent, incomplete requirements and requirements are not specified in a standardised and adequate manner [14].

When businesses adapt the CMMI they should be familiar with the CMM practices (Menezes, 2002). CMMI Based upon the software CMM and has most of the same process areas. It may also inherit some of the same problems as CMM, such as the problem in reaching higher capability levels [21]. This is not acceptable against the agile software developments principle and motivation.

There is a need for a requirements process improvement model to suit story cards based requirements engineering process. Therefore the purpose of this paper is to propose and evaluate a requirements process improvement model for story cards based requirements engineering process and enhance the adaptability of story cards based requirements engineering process.

We propose a model for assessing the story cards based requirements engineering process within software engineering projects. This model should cover the area of story cards based requirements engineering process and practices for agile software development. The model can be used to evaluate the story cards based requirements engineering process maturity for certain projects.

The Story card Maturity model, requirements improvement framework offers the user many advantages. The SMM includes an assessment method that guides the user to understand current story cards based requirements engineering process. The rationale for building the SMM is as:

- To define a generic process model for Story cards based requirements engineering process improvement that is suitable for RE at agile software development environments.
- To design and implement an automated tool that support to apply the proposed model in order to help facilitate process improvement.
- Identify and define story cards based requirements engineering practices
- Recognise story cards based requirement engineering practices problems
- Access and agree story cards based RE practices improvement priorities
- Relate story cards based RE practices problem to RE practices improvement goals
- Contains guidelines for many story cards related requirements engineering activities
- Is designed to be tailored to focus on specific process areas
- Goal focused
- Maturity structure to help with process prioritisation.

2 Process Improvement Framework for Story Cards

2.1 Story Cards Based Requirements
Engineering Maturity Model (SMM)

According to Christie (1999), defining processes is recognised as critical elements in the software process improvement [30] yet to be useful model must be clear simplification of the complex world it is modelling (David 2000). To keep the representation clear, understandable and usable the SMM links the Story Cards based requirements engineering practices to maturity levels, but it is not an exhaustive representation of agile software development practices. The SMM model is based on the agile requirements engineering values, practices and principles.

The SMM model is designed to improve and enhance the agile software development methodology and boost up the agile requirements principles and objectives like the lower cost, customer satisfaction, requirements quality, etc. Figure 4 introduces the SMM (Story card Maturity Model for agile software development requirements engineering). We divided our SMM model into four maturity level compared to the CMM five level maturity model This high level view of the model shows how story cards based requirements engineering practices mature from an initial or ad-hoc level to continuously improving level based on the agile principles and practices. In this model each level has a pre defined goal to help practitioner or organisation focus on their improvement activities. The ultimate goal of the SMM (Story cards maturity model) is as:

- Customer satisfaction
- Maintain story cards (requirements) change
- Solution to Vague requirements
- Obtain an understanding of the user story on the story card
- Obtain commitment to the user story (user Requirements)
- Maintain bidirectional traceability of requirements
- Identify inconsistency between project work and user story
- Identify and involve stakeholders instead of single on-site customer
- Manage Requirements Stories, on-site customer, daily meeting,
- Establish Estimates of story cards and define acceptance tests with story cards
- Develop a project plan based on the story cards

2.1.1 Level 1 : Initial Level (Not accommodating at all)

There is no process improvement goals defined at this unstructured level.

The story cards practices or story card based requirement engineering process is very slim at this level and not necessarily repeatable. Organisations typically do not provide a stable environment for story cards based requirements engineering practices. Level 1 company do not have defined story cards practices. Here at this level RE problems were found to be common. The main problems at this level relate to overtimes, schedule slips, communication, requirements quality and vague requirements. These companies operate in their own unique way and depend on particular people rather than whole team. Paulk et al (1995) describe for traditional software process, success at this level is depends on ‘the competence and heroics of the people in the organisation and cannot be repeated unless the same individuals are assigned to the next project’.

2.1.2 Level 2 : Explored

Level 2 denotes a more structured and complete software development practices than level 1. Organisation with level 2 capability experienced fewer problems with their software development process than their level 1 counterparts.

Problems with communication, complex requirements management and undefined RE process along with staff retention. Technically difficulty for level 2 companies centred on communication (mutual interaction), and to handle complex requirements.

An organisation at this level has focused on the cost, schedule and functionality and story cards elicitation process,
the story cards elicitation practice is used to elicit user goals, elicit the functional requirements. An organisation at this level has introduced policies that ensure that story cards (Requirements) are specified and used the standard structure of the story cards and story cards are written by the on-site customer. Level 2 in general denotes that an organisation has devoted resources to the identification of story cards (requirements engineering) practices as a whole.

In general companies at this level 2 process capability have established the scope of the story cards, story cards based requirements elicitation and identification of stakeholders to track schedules, requirements (Story cards), cost and functionality.

The SMM at level 2 maturity aims to help developers and customers to identify and improve problems related to requirements elicitation and identification of stakeholders by learning from previous project success and failures. This is achieved by an assessment of current process and to identify where weakness lie will help development team gain a general overview and allow them to address any planning or requirements issues associated with individual projects. The Appendix 1 summaries goals, key process areas and assessment questionnaires for SMM maturity level 2.

2.1.3 Level 3: Defined Level

Level 3 denotes a more focus on practices related to customer relation ship management, consideration of dependencies, interaction, conflicts between story cards, acceptance testing on early stage of story cards, prioritization of story cards based on the agile values for iteration planning and stakeholders are consulted to improve the quality of the story cards.

The customer relationship is maintained very well at this level. At this level companies ensure a deeper understanding of acceptance testing for the requirements testing, and subsequently stakeholders are consulted to elicit the requirement from the multiple viewpoints.

At this level companies stored the story cards on the database or the computer system for story cards reuse compared to use and throw concept of the traditional story cards. Story cards analysis can be done through the checklist based on the story cards and agile requirements value and principles.

Level 3 companies had increased their control over their technical practices like requirements testing practices and furthermore the practices related to dependencies of story cards and requirements conflicts are focused, but saw little improvement on the support of the RE process. They improved user understanding of story cards or requirements, internal and external communication but continued to report problem on analysis of domain where the system is going to be implemented and estimation of story cards. At this level no structured risk assessment is performed. Furthermore no consideration is taken towards non-functional requirements.

The SMM at level 3 maturity aims to help developers identify and improve problems related to customer relationship; story cards early testing (Acceptance Testing), dependencies and conflicts management of story cards. This is achieved by an assessment of current process and to identify where weakness lie..

2.1.4 Level 4: Improved (People orientation and project management Practices)

Companies at this maturity level are in a position to collect detailed measure of the story cards based requirements engineering process or practices and product quality, both the story cards based requirements engineering practices and products are quantitatively understood and controlled using detailed measurements (Paulk et al.,).

At this level the system’s environment is studied in grater detail, not only the technical aspect but also the demands coming from the application domain, as well as the business process where the system should support. The improved level of the SMM model is also focused on the estimation of story cards, risk assessment, release planning and testing for non functional requirements. At this level the development team also focus on the advance validation of the story cards and identifies the unit tests from the story cards for the development stage. This is an internal attribute of the team which is not directly visible to the customer. Level 4 denotes a more active and mandatory examination of risk.

The SMM at level 4 maturity aims to help developers or managers to identify the non-functional requirements to improve the quality and to improve the estimation and release planning. This is achieved by an assessment of current process and to identify where weakness lie. The Appendix 1 summaries goals, key process areas and assessment questionnaires for SMM maturity level 4.

2.2 Software Process Improvement Roadmap for SMM

The SMM model is summarised in figure 2.
The key features of the process are as:

- Adaptability and suitability assessment is carried out by the agile team members which are any like developers, coach, testers with collaboration of on-site customer. This is found to be a useful process during the SMM implementation. The purpose of involving this process is to ensure or to identify that the organisation follows the story cards based requirements engineering practices and process or not. If not then this adaptability and suitability recommends what they needs to do to follows the story cards based agile software development.

- Early in the SMM programme the business objectives or business goal are defined by the agile team. The goals drive much of the subsequent activity, especially the selection of KPAs or maturity level and prioritisation of the area for improvements.

- A tailored version of the SMM assessment (similar like CMMI model but the key process areas and goals are entirely different than CMMI) is carried out by the agile team, to identify area for improvement. This is also indicating the maturity level of the software process.

- The plan for the improvement is identified based on the inputs provided to the assessment questionnaires for each maturity level key process areas. In this plan, practices should be identified to support the implementation of the prioritised area for improvements.

- After the identification of the KPAs for each maturity level, a guide based approach was designed to capture the best practices in order to improve the prioritised area for improvement.

### 2.3 The Adaptability and suitability assessment

Adaptability framework is based on the questionnaires, like the determining the main problems in the existing requirements engineering process or requirements engineering practices used or intend to use during the next project, existing knowledge on traditional requirements engineering practices and story cards based requirements engineering practices and process, customer relationship with development team, customer availability during the project, developers attitude or characteristic towards the process and by assessing their knowledge on agile requirements process.

An adaptability questionnaire is actually divided in the following four sections.

- Requirements engineering process used or intends to use.
- Problem identification during the story cards based requirements engineering process and Solution adopted or trying to adopt to solve problems
- Customer availability and relationship
- Developers and Managers knowledge on Agile requirements engineering

Our adaptability assessment brings three result based on the answers supplied on the adaptability Model. Those results are as following

1. Recommended to adopt story cards based requirements engineering methodology on you pilot project.
2. Ready to adopt a story cards based requirements engineering methodology but needs an improvement or needs to pay attention or focus on the recommended area.
3. Pilot project is not suitable for story cards based requirement engineering methodology, but they can still apply agility after adopting agile software development knowledge

### 2.4 Story Cards Practices Assessment

The purpose of the assessment method is to assess the current story cards based requirements engineering practices. Process assessment consists of the knowledge on story cards based agile software development methodology and business case workshop, which focus on process improvement and provides a roadmap for process improvement. The SMM assessment model is based on a story cards based requirements practices and agile software development practices and principles; it is modified and customisable version of the SW-CMM assessment questionnaires. Emphasis placed on the agile practices, developers and on-site customers means people rather then processes. This process is expected to enhance the communication and understanding; in particular it is expected to clarify the actual issues of the people involved in the process improvement actions. SMM recommended
having a shared vision of the process improvement and any one can control process improvement activities at any stage. Self-assessment is the most common way of performing software process assessment (Dutta et al., 1999). The popularity for self-assessment lies in its low cost, good accessibility and ownership of the result (Dutta et al., 1999). We are going to follow the self-assessment for the software process assessment. Automated assessment also considered for this approach.

Figure 3  Automated Tool Support

3 References


A Frame Work for Software Engineers to Support Collaboration

A. Samina Jadoon¹, B. Kashif Hesham Khan², C. Ijaz Ahmad³,
Department of Computer Science
COMSATS Institute of Information Technology Abbottabad Pakistan

Abstract- Good collaboration can be established when each collaborator is fully involved in participation, they have the related skills and they know the fundamentals of collaborations such as communication skills, integration, self control, meta cognition, and most important is the capacity for self assertion etc. Like other disciplines such as Bioinformatics, Environmental Sciences, Social Sciences, Information Technology, Information Assurance etc, it is perceived that mostly computer science applications do not promote teamwork, although teamwork or collaborative task is the very important characteristics that the workplace demands, particularly when designing and developing the large-scale software projects. When multimedia technologies are included in collaborative systems, it became more challengeable because it is very difficult to combine different Medias in the way that are natural for people to use. Some issues of computational load and networking, seat arrangement, floor management, workspace awareness arises when multimedia technologies are used. In this paper we are going to propose a collaborative modal for software engineers that will overcome the problems related to seat arrangement, floor management, network load and workspace awareness.

Key words: CSCW, audio, video, floor control

I. INTRODUCTION:

In collaboration the main unit is collaborative entity which directly involves to develop an idea, design creation and to achieve shared goals. Here the participant continuously exchanges this collaborative entity, which is converse of transactional interaction [1]. CSCW is the Computer Supported Cooperative Work that enables the group workers to share their ideas by using different computer hardware, software and network technologies. [18] Paul M. Cashman and Irene Greif firstly thought about the CSCW in 1984 [2]. Their system not only provides the virtual face to face collaboration among people at distance but also improves the face to face collaboration through a more generalized mechanism and overcome the problems among collaborators [7]. In face-to-face communication same surface e.g. design paper, all pictorial and gestural messages can be share, and CSCW allows people to share information at the same or different time from distributed locations. The main goal of any CSCW application is to integrate audio, video and textual communication that can be local or remote into a coherent software environment like real world collaborative environment or face-to-face meeting. Particularly video is used to improve faraway collaboration, and it has been carried by e-mail or phone. Video actually increases the quality of communications between remote communicators. Ellen, Isaacs and John and Tang SunSoft have found that if remote group workers are connected with both audio and video links their performance can be enhance and they can achieve better results as compare to connect with audio only. [3]. Video is very significant for collaboration because it makes interactions more attractive. For collaboration we need special methods, tools and techniques such as audio video and text base chat and Shared White-Boards etc. Wide range of CSCW tools exists but sharing information is quite limited. When multimedia technologies are included in collaborative systems, it became more challengeable because it is very difficult to combine different medias a way that is natural for people to use. Some issues of computational load and networking, seat arrangement, floor management and workspace awareness arises when multimedia technologies are used. In this paper we are going to propose a collaborative modal for software engineers that will overcome the problems related to seat arrangement, floor management, network load and work-space awareness. Section II defines background and literature review i.e. mode of collaboration, audio video conferencing and their limitations. Section III will present our propose framework.
for software designer i.e. workspace awareness, seat arrangements, floor management and computational load and networking. Section IV will conclude the paper.

2 BACKGROUND AND LITERATURE REVIEWS:

2.1 Modes of Collaboration:
A Modes of Collaboration: Human-Computer Interaction (HCI), Social Computing, Management System, and Collaborative Intelligent System are different fields included in CSCW and it is focus on the technologies that effect groups, organizations, communities and societies [7] [8] [9] [10] [11] [12] [17]. Among CSCW available tools i.e. [11] Email and text based chat is commonly used people to talk each other on internet by text. Graphics based chat uses both text and graphics to express understanding. Audio/Video-based Chat uses video and audio facilities. Shared White-Boards includes different windows on it and each person can write and draw on his site. Similarly many other tools are available for CSCW but all have a common problem that they can not support real world face to face collaboration.

2.2 Audio/video conferencing:
Videos play an important role to improve the interference between the participants. Other than by using only the audio link the joint connection of audio and video are more likely give the better results for problem solving and also making the better decision [3]. In this video is used for managing the conversational mechanism for turn taking, scrutinizing, and more over reaction is judged on basis of noting and adjustment. If video is effective in process of interaction, then remote partners can interact more effectively and get the encouragement. Now video has become an important tool for collaboration because it has become the striking demand for the market. For this purpose we must implement such type of system that gives the full utilization of multimedia type of systems in an effective way. Desktop video over the face-to-face meeting get the two main features [3]. One is the establishing the eye contact of distanced participants in case of when they can’t do so in face-to-face meeting. Other is that both the off-line and on-line material regarding to their discussion can be shown to all other participants at the mean time.

To support collaboration in the field of computer science we need special methods, tools and techniques which are provided by CSCW system. Followings are some benefits and limitations of video in collaboration [3]. To express the understanding and agreement different visual channels are used by the participants. For example by nodding the head while some one was speaking is the sign of agreement about something which cannot be done through phone. Forecast responses: telephone is not the best choice to predict expression and gestures; it needs to explicitly express their reactions verbally which require more efforts. Enhance verbal descriptions with gestures: Many people gesture while talking because it helps to express them verbally. If people cannot see each other they cannot express. Conveying purely nonverbal information: People could not convey purely non verbal information on phone e.g. during discussing any problem if two persons are not in the condition to resolve it, after this they show each others facial expression by looking at others

2.3 Limitations of video conferencing:
Floor control, turn taking, workspace awareness, peripheral cues, network load management and manipulation of real world are difficult or impossible in interacting remotely through video [3]. To manage turn taking in video conferencing is little bit different from that of the face to face communication. Turns are shorter in duration and the participants are able to more tightly coordinate utterance in face-to-face which enhances their ability to reach mutual understanding [3].

Using peripheral cues: user cannot detect any change in other’s body position using peripheral cues in desktop video conferencing. Have side conversations: People can not address particular person and every one share a single audio channel so side conversation is impossible in data video conferencing. Manipulating real world objects: Participants cannot manipulate or built an object jointly on screen, they can use only hard paper. Some of these limitations are covered by VR environments but also many issues exist for CSCWS e.g. computational load and networking, Seat arrangement, Floor management, workspace awareness, Speaker recognition etc.
3. PROPOSE FRAME WORK FOR SOFTWARE DESIGNERS:

3.1 Work-space awareness:

Conversations can not be fully conveyed in electronic meeting due to lack of mutual awareness [15]. One person understanding of others interactions with in share environment is called work-space awareness and is key to rich collaboration. The key factors of work-space awareness are [13] gestures, speech and gaze. In collaborative interactions the large number of activities contains gestures including pointing to object drawing objects, grasping and pushing, moving and picking objects. Verbal communication/speech is a fundamental part of human communication where languages are use to support this communication. It is very easy to manage gaze in face-to-face meeting but very difficult in electronic meetings because every one is looking to camera. The seating arrangement should support gestures, consequential communication and gaze awareness.

3.2 Seat arrangements:

There are many types of seating arrangements appropriate for particular situations e.g. U-shape, hollow square and octagon seating arrangement [4]. U-shape seating arrangement is the ideal environment for the minimum number of 50 people for discussing, learning, training, and speaking purposes.

Hollow Square seating arrangement is like 'U' shape setup but the arrangement of the people are in the square shape and their main focus is always in front of them and is suitable for the less than 30 people. Octagon is the most important conversational environment of about minimum of 17 and maximum of 30 people and their main focus is in interaction between the participants. It is a variation of the hollow square. Similarly some other seating arrangements are classified on the basis of four types of meetings [5] i.e. Convey meeting: For conveying information and making participants understanding. Here need of one presenter opposite to participants, so A Type seating arrangement is appropriate (figure 1.4). Creative meeting: For solving problems and analyzing matters, it is necessary to make an atmosphere that enables participants to make idea easily. For this purpose they need sit around a round table as in Fig 1.5 B Type seating arrangement. Coordination meeting: Like creative meeting and for checking activities of each department and discovering overlap. Coordination meeting's seat arrangement should be B Type seating arrangement but A Type seating arrangement is better when each department conveying information to another. Decision meeting: For deciding intentions and activities of a company, B type seating arrangement is better for decision meeting when all participants discuss one theme and they have the same rank. In case of a debate as two groups discuss a theme to reach a conclusion, it is necessary to distinguish between two groups then C Type seating arrangement is better. In case of panel discussion D Type seating arrangement is better. Research shows that there is no need of seeing the lower body of a person so it is easy to communicate to others in sitting position and human body should be presented in its actual size and 27 degree of visual angle is natural and appropriate[5]. So we are also going to use B type seating arrangement for CSCW for geographical dispersed collaborators because all the software engineers are at same level and in collaborative meetings each participant has to fully involve for solving a problem. This seating arrangement will support gestures, consequential communication and gaze awareness.
3.3 **Floor management:**

Floor management: Management of interaction between participants in meetings is called floor control [6]. Floor control is a technology that helps to coordinate joint activities and resources among people and their interactions e.g. preserving coherency of local and remote information and controlling turn-taking in conversations or write-updates on shared files [15]. Floor control issues arise is CSCW because participants share the common resources (like work space). In face- to-face meeting eye gaze are used to addressing the others but in desktop video conferencing every one see through same camera so it is difficult to floor control in video conferencing [3]. For the cooperating people electronically the gaze awareness and gestures cannot be produced by commonly used input devices because people can only be seen indirectly through a common screen [13]. Floor control policies have three primary independent dimensions: [16]

1. How people give up or release control that can be explicit release, implicit release and explicit loss?
2. How people acquire control when control is available, there can be various options for this i.e. moderator is responsible for deciding who gets control, explicit request, implicit request and rule based.
3. What happens to requests if control is not available?

Based on these releases and requests many floor control policies are constructed but no one is ideal for specific application. Protocols implementing floor control for CSCW need to observe system and network performance constraints and patterns of interaction between
individuals [15]. A generic protocol is proposed by [15] but not a precise account of how to implement floor control optimally for any given application and many issues exist e.g. accuracy, consistency etc. Audio and video resource sharing can temporal, spatial and functional. Temporal sharing is due to time conflicts. A spatial sharing conflict is due to shared work space. Functional sharing of the same application functions. In this modal, floor control protocol for software engineers will be an efficient and consistent protocol and will based on cues given by input devices e.g. voice-activated, mouse-triggered, or gesture-based floor requests or releases and time limits are set on the holding duration.

3.4 Computational load and networking:

Co-located users need communication media e.g. audio, video, text and require network structure and transmission channel. A network structure for minimizing traffic for CSCW application is important. The amount of the traffic in the network varies over a wide range and the video image packet is greater than the size of static image audio packets and size also depend on quality of image. Video conferences are accomplished through point-to-point connection or by broadcasting the video but these modals are not effective, because by server all the traffic has to pass one particular point. If the number of packets increases the server will go down but it can support hundreds of users if only light-weight-media are used and in point-to-point no benefit of multicasting [14]. Our modal is based on following: 1. two different channels are used one for video and one for audio and other interactions data. 2. At a time static pictures of participant are sent through medium and only the video of one or two interactive participants is sent so the load can be minimize because the video image packet is greater than the size of static image and audio packets. 3. The high quality video and audio is needed for collaboration but network traffic size increases through high quality video and audio, so low quality image and voice should be send on medium and there quality should be enhance on receiver side through converters.

4. CONCLUSIONS AND FUTURE WORK:

In this paper some issues are discussed regarding the involvement of multimedia in collaborative systems. Issues that are mainly focused are of workspace awareness, seat arrangement, floor management, computational load and networking arises when multimedia technologies are used. A collaborative modal for supporting software engineers that will overcome the problems related to the issues mentioned above are proposed here. In this modal we have used B type seating arrangement for geographical dispersed collaborators because in collaborative meetings, each participant has to fully involve for solving a problem. The floor control protocol for software engineers that is an efficient and consistent protocol must be based on cues given
by input devices e.g. voice-activated, mouse-triggered, or gesture-based floor requests or releases and time limits are set on the holding duration. To overcome the problem of computational load, this modal is based on following. 1. Two different channels are used one for audio/video and one for other interactions data. 2. At a time static pictures of participant are sent through medium and only the video of one or two interactive participants is sent so the load can be minimize because the video image packet is greater than the size of static image audio packets. 3. The high quality video and audio is needed for collaboration but network traffic size increase through high quality video and audio, so low quality image and voice should be send on medium and there quality should be enhance on receiver side through converters. Our future work is to implement this modal and our second write-up will be focus on the results and comparisons between propose modal and existing tool available for CSCWS.

REFERENCES:

[10] David N smith IBM Social Computing group august 1998...
SESSION

THEORETIC APPROACHES

Chair(s)

TBA
Model-based Object-oriented Requirement Engineering and its Support to Software Documents Integration

1* William C. Chu, 2 Chih-Hung Chang, 2 Chih-Wei Lu
1* Dept. of Computer Science and Information Engineering, Tunghai University, Taiwan No.181, Sec. 3, Taichung Port Rd., Situn District, Taichung City 407, Taiwan (R.O.C.)
2 Dept. of Information Management, Hsiuping Institute of Technology, Taiwan No.11, Gongye Rd., Dali City, Taichung County 412, Taiwan (R.O.C.)
1* cchu@thu.edu.tw, 2 {chchang, cwlu}@mail.hit.edu.tw

Abstract

Abstract - Maintaining uniformity of software requirement documents with artifacts from other phases of software life cycle is a very important, however, still a difficult and time-consuming task. Most requirement documents were written in ambiguous natural language which is less formal and imprecise. Without proper modeling, the requirement knowledge is laboriously captured and thus the following formal integration with other artifacts is infeasible. In this paper, we propose a Model-based Object-oriented approach for Requirement Engineering to support and improve the maintenance and consistency of software requirement documents, as well as the consistency with other artifacts through software life cycle. By applying modeling and Object-Oriented technologies to requirement, the domain knowledge can be captured in a well-defined model. The completeness, consistency, traceability and reusability of requirement and its integration with the artifacts of other phases thus can be improved.

Keywords: Software Maintenance, Object-oriented, Requirement Document, UML.

1 Introduction

By the inundation of computers and networks from the end of the 20th century, informatization has generally acknowledged as the main trend toward modern business operations and enterprise management. Yet this population brings more challenges to the software systems development. Software now needs to be more flexible for enterprise mutability, and quicker but with more accurate analyses, as well as the designs, for the least time-to-market commercial essentiality. In practice, most of software requirement documents are represented in natural language, which are usually ambiguous, imprecise, incomplete, and informal, in spite of containing some concrete descriptions such as use-case diagrams. Without proper modeling, the knowledge of the requirement is hard to be captured formally and its integration with other artifacts is infeasible. Moreover, the modeling process of software requirement needs to face a challenge of representing these informal and incomplete requirement documents into a well-defined model, so the modeled requirements can be integrated with other much formal software artifacts, such as the diagrams in UML from the upcoming phases analysis, design, implementation, and testing. By observation to the companies who have practiced CMMI and passed level 2 and 3, one of the biggest costs is to maintain the traceability and the consistency of related documents.

On the other hand, for the last decade the growing influence of object-oriented programming (OOP) has led to the rise of new paradigms for software development, generally known as object-oriented analysis and design (OOA/OOD). The most promise of OOA/D and OOP is that the software development process can be efficient and simplified through having the common building blocks [1] (e.g., classes, objects, methods, and inherences), used in phases of software development, from software analysis to implementation. These paradigms adopt concept from object-oriented programming and blend it with semantic data modeling and knowledge representation into modeling frameworks such as UML, which makes the software development process more concrete and manageable and therefore OOA/D have been accepted as the general formal methodologies used in current software development.

However, there are still insufficient of corresponding consistency that OO complies for the requirement process. Defining and specifying the requirements of a software system from the very early stage is essential of the software development lifecycle since meeting stakeholder’s needs is always the principal part for any project. Requirement phase activities, which indeed need the professional and formal treatment the most, usually were treated the less in an informal way. Practically, in requirement elicitation and elaboration stages, natural language has still remained as the general choice for the software developers to specify the software requirements for higher flexibility, since most of the stakeholders...
involved were either poor-trained or lack of domain knowledge and experiences[2].

In this paper, we propose a Model-based Object-oriented approach of Requirement Engineering (MORE) to support and improve the maintenance and consistency of software requirement documents and its consistency with other artifacts of software life cycle.

Applying modeling and OO technologies to requirement, the domain knowledge can be captured in a well-defined model, so the completeness, consistency, traceability and reusability of requirement and its integration with the artifacts of other phases can be cost effectively improved.

This paper is organized as follows. In section 2, we list the related works and the summaries of our earlier research. Next, the XML-based unified model (XUM) and MORE integration is shown in section 3. In the section 4, we conduct an industrial case study to demonstrate the mechanism. Conclusion and future works are given in section 5.

2 Related work

XML [3] is a standard language supported by W3C (World Wide Web Consortium). It offers application neutrality (vendor independence), user extensibility, ability to represent arbitrary and complex information, validation for data structure scheme, and human readability. In our earlier research [4], we apply XML meta-model to describe and integrate related software paradigms of a system into an XML-based unified model (XUM). XUM is a mechanism to integrate and unify sets of models/paradigms (e.g., UML or design patterns) in well accepted software standards (e.g., OOA, OOD, or OOP). The mechanism utilizes a XML-based unified meta-model (XUMM) to construct the corresponding XUMs which present the schema of adopted models in software development process. The mechanism also creates explicitly the relations between these adopted models. The connective models assist the maintenance of consistency by reflecting the changes to all affected models. The mechanism handles ripple effects systematically to reduce the difficulty of software maintenance.

Many approaches have been proposed for requirement engineering. Object-oriented requirement engineering, which takes the good features of object-oriented paradigms, has been proposed to assist the activities of requirement phase, such as requirement elicitation, analysis, negotiation, documentation, verification and validation, and management. Many researches focus on use cases and scenarios to systematically model the requirement artifacts. In our early work of MORE [5], it demonstrated the feasibility of the OO modeling for informal text-based documents.

In past decades, design patterns [6] propose solution skeletons for common design problems. The solution skeleton is described the design can be reuse for other projects. Based on the concept of design pattern, more researches are aware of patterns applicable to other parts of the software development process. Gross and Yu [7] discussed the relationship between non-functional requirements and design patterns. Sutcliffè et al. [8] described how scenarios of use cases can be investigated to identify generic requirements for different application classes. Konrad et al. [9][10] applied patterns to requirement for embedded system.

For official requirement specification, IEEE presented several sample about SRS outlines [11]. However, they still used natural language to describe the requirement of system.

3 The XML-based unified model and MORE

In MORE, information of requirement documents are generally collected and written in naturally language, and then requirement engineer can objectize key concepts into corresponding objects and class, which are called ROMs (Requirement Object Models). The objects are kept in reusable repository, which can be reused for further requirement definitions. ROMs and the objectization of requirement documents lead to a formal system representation which can assist precise specification of the requirement semantics and then prevent incompleteness and inconsistency.

By the construction of ROMs and the related requirement domain knowledge, a set of model-based requirement templates are accumulated. A requirement template consists of actors, scenarios, constraints, and effects. The templates encapsulate the experiences and knowledge of domain experts, which can help and guide requirement engineer to elicit requirements more correctly and effectively.

![Figure 1: The concept process of MORE](image)

Figure 1 shows the concept process of our approach. With the support of ROM, the process of the design phase is assisted with reusable requirement objects, so the related design paradigms such as the class diagrams, activity diagrams, …etc, will be associated and consist with each other in the XML-based Unified Model (XUM).

To making up the space that XUM approach left in the requirement phase, for the following software development phases, software paradigms are represented and integrated as models of XUM. Figure 2 shows the relationship between the software paradigms and XUM.
During software maintenance, modification to any sub-model should be detected and should reflect the effects on the related sub-models; the semantics in each sub-model can then be updated appropriately according to the modification. This assists the consistency checking of modeling information of views. In addition, impact analysis can be applied to the entire software system, including the impact on related source codes, the impact on related design information, and the impact on related requirement information.

4 Case study

In order to demonstrate our approach from document-centered requirement development to MORE development, we conduct an industrial case study. The subject of the case is about developing the software requirements of a Book/Magazine distribution system.

Table 1: A set of questions support collecting original needs from customers

<table>
<thead>
<tr>
<th>Category/Questions</th>
<th>Example (B Corp)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Goals</strong>/What are the major goals of the project?</td>
<td>B Corp. has an ERP purchase system but its document approval procedure is executed manually. The company wants to exploit the approval application of book/magazine to increase the performance and reduce error rate.</td>
</tr>
<tr>
<td><strong>Functional Needs</strong>/What are the major functionalities of the targeted application?</td>
<td>1. The book distribute system can support automatic distribute procedure. 2. The book distribute system can support adjusting flexibly the authorized amount of book/magazine of each position.</td>
</tr>
<tr>
<td><strong>Non-functional Needs</strong>/What are the non-functional expectations of the targeted application?</td>
<td>The user must be authorized before he/she distribute.</td>
</tr>
<tr>
<td><strong>Operation concepts</strong>/What are the operation procedures?</td>
<td>3. The staffs of B Corp. interact with the book distribute system to distribute the new book or magazine to each pathway by different levels of bookstore.</td>
</tr>
<tr>
<td><strong>Constraints</strong>/What are the constraints of the targeted application?</td>
<td>1. Before book distribute, the related data must be input into database. 2. If the difference between the amount of distribute and the amount history is the large, system must email to notify related user.</td>
</tr>
</tbody>
</table>
| **Environment** | Hardware: Server Intel 2.6G/1G RAM  
Software: Database SQL Server 2000  
Network: Intranet  
Legacy system(ERP): Database Informix 7.3 |

In the development of customer requirement, developers elicit customer needs with the questionnaire, as shown in Table 1. The replies got from the questionnaire imply implicit software needs and explicit hardware demands. The design of software needs elicitation is emphasized on the understanding of project goals, functional or non-functional needs,
The design implies that the customer needs could include the concepts, which is informal at the early stage.

The case study involved with developing the software requirements of a workflow application adopted the requirement development process shown in Figure 5.

![Figure 5: requirement development process](image)

**Table 2: The detailed identification of book distribution requirement**

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Date</th>
<th>Business process ID</th>
<th>Designer</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Book distribution</td>
<td>5.30.06</td>
<td>P3.2</td>
<td>Eric</td>
<td>1.0</td>
</tr>
<tr>
<td>Actor</td>
<td>Bookman</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Module Description</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. The distributor input the book_id, to check the data is existed?</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Select using fixed or no,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Select the mode of distribute</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pre-condition</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>User had finished the basic materials of of new book in I1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Post-condition</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. successful: goto 1.2Transformation of pathway</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. unsuccessful: show the message “The data of new book are not exist”, and disable the button of 1.2Transformation of pathway or 1.3Select the pathway</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Main process</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. User input the ID of new book which want to distribute</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. User input the amount of book</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. User select the mode of distribute</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. User click the [OK]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. Check the data of new book are alread final in I1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. If it has been existed, goto 1.2Transformation of pathway, 2.0 Distribute the level of new book, 3.0 Distribute by area or 4.0Distribute of magazine</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7. Else show the message “The data of new book are not exist”</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alternate process</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assist:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Screen:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The customer requirement is developed from the needs of customers. The customer needs generated from the questionnaire are used to roughly identify the components in the application, including which business processes should be automated (workflow application), which documents are managed/executed on the processes (relevant data), and which legacy systems (ERP purchase system) will be integrated with. Table 2 shows the results elicited from the original needs.

**Figure 6: The corresponding Use Case diagram of system requirement specification**

Based on the development procedure, the customer requirement of the project generated in this stage might contain the identifications of all coarse components in the workflow application (e.g., the business processes, extra applications, and relevant data). So, the components of workflow application might be the fundamental entities of the customer requirement. The design of customer requirement model is composed of the same entity model as customer needs model. The customer needs model and customer requirements model associating with each other are linked with association relation. The detail design is shown in Figure 4.

Figure 6 shows the corresponding Use Case diagram derived from system requirement specification. And Figure 7 is the corresponding activity diagram of system requirement specification.

![Figure 7: The corresponding activity diagram](image)
According to the integration links, any modify of requirement, developer can extract related paradigms easily. For example, if we change the content of customers_need, according to the requirement: 1.0 Distribute pre-operational maybe need to modify.

```xml
<requirement id="R1.0" name="1.0 Distribute pre-operational">
  <unifiedLink link="Abstraction" xlink:type="simple" xlink:href="R1.0"/>
  <goals>
    <goal>The pre-operational before distribute the books or magazines to retailers</goal>
  </goals>
  <nonFunction>
    <nonFunction>1. The rate of crash \&lt; 0.000001</nonFunction>
    <operationConcepts1 User input the Book_IDs</operationConcepts1>
    <environment>
      <requirement id="R1.1" name="1.1 Book distribution" date="2006/05/30">
        <designer>Eric" version="1.0" businessProcessID="P3.2">
          <unifiedLink link="Integration" xlink:type="simple" xlink:href="A1.1"/>
          <actor id="UA001">
            <description>1. Distribute input the Book_id, to check the data is existed</description>
          </actor>
          <operationConcepts1 User input the Book_id, to check the data is existed</operationConcepts1>
        </designer1>
        <pre-condition>User had finish the basic materials of new book in 11</pre-condition>
        <......
      </requirement>
    </environment>
  </nonFunction>
</requirement>
```

Figure 8: The correspond XUM

In order to prove our methodology, we implement a prototype called MOR Editor. MOR Editor provides classification to organize the defined objects in class hierarchy. Requirement engineer can apply inheritance techniques to increase the reusability. MOR Editor also provides association function to allow requirement engineer to link these informal and formal objects and classes together. Therefore, related document segments (informal ones) and formally defined objects can be associated with.

Through MOR Editor, the documents in requirement are modeled and candidate reusable documents are made as template. Our approach also adapts the concept of pattern, successful solutions to recurring problems, to the requirement modeling. Requirement engineer can define reusable requirements into templates. For example, a requirement specification about any data processing needs concern about its functionality the creation, deletion, modification, and query. Without considering them all, in practice, we have found many requirements are incomplete, which may cause the cost estimation wrongly. The template contains the experience and knowledge of domain expert, which can help and guide requirement engineer to solicit requirements more correctly and effectively. Java is adopted as the implementation language for the tool prototype, so the MOR Editor can be run across platforms such as Windows, Linux, or UNIX. There are five major toolsets collaborating one another in MOR Editor.

A) The toolbar of MOR Editor, to manipulate functions of the editor.

B) The requirement/object structure browser, to list all element objectization of the requirement in the MORE structure.

C) The requirement engineer viewer, to display the contents needed for requirement engineer assistance.

D) The requirement workplace, to edit requirement documentation. In here, requirement engineer can define the requirement details, requirement states, and requirement relationships.

Figure 9 shows the layout of MOR Editor toolsets in Windows.

Figure 9: Layout of the tool MOR Editor

5 Conclusion

Software requirement is the cardinal importance for developing and maintaining an information system because it is what a system is needed for. Requirement documents are usually written in natural languages. Most of them are informal and that leads to the difficulty for capturing complete/consistent domain knowledge. As a result, the following processes of software development and maintenance are costly and error prone.

Many disparate software analysis/design methods and tools promise fast, efficient software design/evolution, yet they are generally incompatible and therefore suffer from a lack of communication and integration, especially to the paradigms of requirement
process. Problems of implicit inconsistency, which are caused by making changes to requirement specifications, components of the models and designs, significantly increase costs and errors for maintenance.

In this paper, first we proposed a model-based requirement development framework to improve the quality of requirement analysis. Second we proposed a mechanism to integrate system paradigms from requirement to implementation. If system developers/maintainers modify any software paradigms, he/she can easily spot for the of related paradigm updates. Third we implement a requirement editor prototype MOR Editor.

For the future work of this research, we will improve MOR Editor more complete to support XUM which serves as a mechanism for the integration and maintenance through the whole process of software life cycle.

6 References


[6] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley.


Integrating Z in DEVS: a case study
Lift Control System

Mohamed Wassim Trojet¹, Maâmar El-Amine Hamri¹, Claudia Frydman¹
¹LSIS UMR CNRS 6168 Université Paul Cézanne
Avenue Escadrille Normandie Niémen, 13397-Marseille cedex 20, FRANCE
{wassim.trojet, amine.hamri, claudia.frydman}@lsis.org

Abstract
We present a case study to illustrate the usefulness of formal methods (FM) for improving Verification and Validation processes (V & V) of Discrete Event Specification (DEVS). The formalisms employed is Z. Although DEVS is well-established Modeling and Simulation (M & S) framework, DEVS models lack the possibility to be verified formally by developing rigorous proofs of their properties. Integrating FM in DEVS, will cover this lack. In fact, it will permit to check consistency of DEVS models with formal specification and theorem proving techniques. The case study employed is a lift control system. We present a model for the system described by the DEVS formalism. In addition, we formalized required safety properties for the system with a predicate logic assertion, we specified the model with Z and formally verified that the behavior of the control conforms the safety requirement.

Keywords: DEVS, Z, Formal reasoning, Proof obligation.

1. INTRODUCTION
Modeling and simulation (M & S) becomes crucial in analyzing dynamic system behavior. Many formalisms and methods are introduced in order to improve this process. The effort is twofold: a) reach more realistic models and results and b) obtain “simplified, flexible” models. DEVS is a discrete event system modeling formalism. Efforts are conducted to adapt this formalism to several domains and features. In our work, we are interested in formal reasoning (logical analysis) on DEVS models. Formal reasoning consists in checking the consistency of the system (all specified properties taken together don’t lead to contradiction) and finding a proof of property from the system expressed as logic-based formulas in terms of axioms and inference rules. Formal reasoning is a basic step in “Formal Methods” (FM) which requires a formal specification to be performed [Clarke and Wing 1996]. Our approach consists in translating DEVS models into Z specification schemas to permit logical analysis on them. In fact, Z is a formal specification based on set theory and first-order predicate logic [Spivey 1992] [Woodcock and Davies 1995]. There are available powerful tools which support Z (notably Z-Eves, ProofPower,...), these tools facilitate formal reasoning about Z using the theorem proving technique to enable formal verification. The advantages of our approach are:

- Giving logic semantics to DEVS models enabling formal verification and proof of their properties.
- Exploiting the Z proof tools in the logical analysis of DEVS model.
- Enlarging the scope of DEVS framework by integrating formal methods.
- Automatic translation of DEVS models into Z specification schemas by defining rules which allow for the implementation of the approach.

There are few works dealing with giving logic semantics to DEVS models: In [Taoré 2005] [Traoré 2006] Traoré worked on making DEVS Models amenable to formal analysis. The main idea of his work was to establish a new framework he called “Z-DEVS” in which he integrated the “Z-Object” to DEVS framework. In fact, he defined different Z-Object schema classes for specifying the fundamental entities of DEVS framework (i.e., models and simulators). However, Traoré did not state the rules of transformation from DEVS models to equivalent Z-DEVS specifications. Hence, his approach requires a good knowledge of Z-Object specification. Moreover, He did not explain how to develop rigorous proof to check the consistency of the model, an important step in formal analysis.

In [Cristia 2007] [Cristia 2008] Maximiliano established the equivalent TLA+ specification of a DEVS model (atomic and coupled) under some rules. The motivation of his approach consists in bridging the gap between DEVS and other formal notation to make DEVS known by the formal methods community of Computer Science. Nevertheless, Maximiliano didn’t exploit the
potential of FM in the formal reasoning on DEVS models.
In our work, we try to cover these lacks and provide a general approach. We aim not only to make DEVS known by the FM community of Computer Science but also to make FM beneficial to DEVS framework. This paper is organized as follows. In section II we review the DEVS formalism, atomic and coupled model. In section III we present a brief introduction to Z specification. In section IV we state the main idea of our approach. The case study is given in section V. Finally, we conclude the paper in section VI.

2. DEVS REVIEW

DEVS[Zeigler 1976][Ziegler et al. 2000] is a modular formalism for deterministic and causal systems modeling. It allows for the modeling of the behavior of systems. A DEVS model has a time base, inputs, states (with functions of transition from one state to another) and outputs. Complex models are built from atomic models connected together in a hierarchical fashion. Interactions are mediated through input and output ports. That allows for modularity.

2.1. Formal specification of an atomic DEVS model

AtomicDEVS= < X, S, Y, δint, δext, λ, ta > - The time base is continuous and not explicitly mentioned: T= ℜ
- X={ (p,v) | p∈InPorts, v ∈ Xp } set of input ports and values.
- Y={ (p,v) | p∈OutPorts, v ∈ Yp } set of output ports and values.
- S: set of sequential states.
- δint: S→S. Internal transition function, allowing the system transit from one state to another autonomously.
- δext: (Q,X)→S. External transition function allowing the system transit from one state to another just after reception of an external event on the input port.
- λ: S→Y. Output function.
- ta: S→R⁺∪∞. Lifetime function of the state in the model.
- Q={ (s,e) | s∈S, 0≤ e≤ta(s) } set of total states.

The system reaction to an external event depends on its current state, the input value and the elapsed time.

2.2. Formal specification of a coupled DEVS model

The coupled DEVS formalism describes a discrete event system in terms of a network of coupled and atomic components.

CoupledDEVS = < X, Y, D, Md | d∈D, EIC, EOC, IC, Select >
- X: the set of input ports of the coupled model.
- Y: the set of output ports of the coupled model.
- D: Set of names associated to the model components, self is not in D.
- EIC: Set of input connections which connect inputs of the coupled model to one or many components inputs.
- EOC: Set of output connections which connect one or many component outputs to outputs of the coupled model.
- IC: Set of internal connections between components of the coupled model.
- Select: Defines a priority between simultaneous events that concern different components.

3. Z SPECIFICATION

Z is a formal state-based specification[Potter 2001][Bowen 2003]. A main ingredient in Z is the way of decomposing a specification into small pieces called schemas. Schemas are used to describe both static and dynamic aspects of a system. The notation of the schema is the following:

```
Schema name
declarations (state space)
predicates
```

3.1. Z Specification using schemas

Generally, the structure of Z specification document[Spivey 1992][ISO 2002] consists in:

1. Declaration of basic types used lately into the specification.
2. Definition of the global abstract state of the model.

```
State
declarations of the variables describing the state of the model
predicates (constraints on states variables)
```
3. Definition of an initial state of the model

 InitializingState
 State
 Initialization of states variables

4. list of operations, each one is presented by the following schema:

 Operation

 $\Delta$State($\Delta$: to say that the state of the system is changed) OR
 $\Xi$State($\Xi$: to say that the state of the system is the same)
 Eventual declaration of input variables ("?" has to be placed after input variable)
 Eventual declaration of output variables ("!" has to be placed after output variable)

 Pre-condition(condition on values of state variables and eventual input variables just before the operation)
 Post-condition(values of state variables and eventual output variables just after the operation)

3.2. Proof obligation in Z

In traditional Z-based specification methodologies, designers must conduct a set of formal proofs to verify incrementally the consistency of the system being modeled [Jacky 1997] [Peschanski and Julien 2003]. In state/transition approaches like Z-based model this mostly consists in (1) initialization theorems to ensure that initializations preserve state invariants and (2) precondition calculations to enforce the consistency of the operations modifying the state space. Establishing the list of all preconditions ensures that either the state invariant is completely preserved by operation "effects" or that some other condition must be fulfilled.

4. PROPOSED APPROACH

DEVS is a formalism which permits description of dynamic systems and their desired properties. These properties might include functional behavior, timing behavior, or internal structure. However there are not techniques which allow checking the consistency of DEVS models, (all specified properties taken together do not lead to a contradiction). Besides, DEVS does not involve rules for inferring useful information from the specification (the proof theory). In this section, in the order to cover these lacks, we propose a transformation of DEVS models to Z specifications. The clear advantage of this approach is that is possible to reuse all the previous work on the FM for analyzing DEVS models. Our approach (Figure. 1) is based on two techniques:

- Adding an element to the algebraic structure of DEVS which allows describing invariant properties called safety properties in a formal way using predicate logic and set theory, we have called this element "$\phi$". Hence, an extension of DEVS is born: $\phi$DEVS.
- Transforming $\phi$DEVS models to equivalent Z specifications. Next, we will detail our transformation algorithm.

5. CASE STUDY : LIFT CONTROL SYSTEM

5.1. Description of the lift control system

A lift control system is required to control a single lift in a multi-storey building. Requests for the lift can either be made by a customer waiting on a floor of the building, or by a customer traveling in the lift. The task of the controller is to service these requests by moving the lift to required floors. If the lift is empty, it serves always the first request else it serves the request of the customer inside. An important safety property of the lift control system is that when the lift is not stopped, the lift door must be closed. The $\phi$DEVS model of the system is divided into two components: DEVS model shown in the figure 2 and the set of assertions $\phi = \{engine \neq stop \Rightarrow door = closed\}$.

5.2. DEVS model of the lift control system

In this subsection, a DEVS model for the lift control system is introduced. This model has an input with event to indicate the requested position of the lift. The output event of the model shows the current position of
Figure 2. DEVS model of the lift control system

the lift. The state space of the DEVS model comprises
the phase variable, a variable “Engine” to represent the
lift movements \{up, down, stop\}, a variable “Door” to
describe the state of the door \{open | closed\}, a variable
“ReqPos” to remember the requested position of the lift,
and a variable “CurrentPos” to update the current po-
sition of the lift.

The lift control system is assumed to be initially
stopped, positioned over the floor number 0, and with
the door closed. Also, the lift control maintain the last
position requested and it is not obliged to return to the
initial state.

In the figure 2, there is the DEVS model, represented
by a phase transition diagram, for the lift control system
[Song and Kim 1994]. An output event is specified on a
dotted line by an output port followed by a message
name with output operator’!’. Similarly an input event
is specified on a solid line by an input port followed
by a message name with input operator’?’. The behav-
ioral description of the atomic model is represented by
an activity transition diagram or a state transition di-
agram, which consists of nodes and two-colored edges.
Each node represents an activity or a state, dotted arc
denotes an internal transition and solid arc an exter-
nal transition. Optionnally, transition condition can be
specified after input or output event with a separator
notation @. The time advance in a state is attached to
the state node because it represents a sojourn time to
fulfill its activity.

5.3. The equivalent Z specification of the
lift control system

The equivalent Z specification will be the following:
First the basic types of the system are introduced:
Action ::=moveUp|moveDown|stop
StateDoor::=opened | closed
PHASE::=Init | Waiting |EvtRecep | MoveUp | MoveDown | ReachUpper | ReachUnder | Stop | OpenDoor | CloseDoor

The finite sets of the possible values of the state vari-
ables phase, Engine and Door of the Lift ϕDEVS model
are represented by the basic types: PHASE, Action and
StateDoor.

\[
\begin{align*}
X & \quad \text{in} : \mathbb{R} \\
Y & \quad \text{out} : \mathbb{R} \\
S & \quad \text{phase} : \text{PHASE} \\
& \quad \text{Engine} : \text{Action} \\
& \quad \text{Door} : \text{StateDoor} \\
& \quad \text{CurrentPos} : \mathbb{R} \\
& \quad \text{ReqPos} : \mathbb{R}
\end{align*}
\]
The schema $S$ identifies the state variables of the Lift $\phi$DEVS model.

\[
\begin{align*}
\text{InitLift} & \\
\text{Lift} & \\
\delta_{\text{ext}} & \\
\delta_{\text{int}} &
\end{align*}
\]

The Z abstract state schema of the Lift contains state variables of the Lift $\phi$DEVS model. These variables have to satisfy the property $\phi$.

The schema $\text{InitLift}$ describes the initial state of the Lift $\phi$DEVS model by affecting initial values to state variables of the Lift schema.

\[
\begin{align*}
\text{InitLift} & \\
\text{Lift} & \\
\delta_{\text{ext}} & \\
\delta_{\text{int}} &
\end{align*}
\]

The schema $\lambda$ identifies states of the lift $\phi$DEVS model which have outputs.

\[
\begin{align*}
\text{InitLift} & \\
\text{Lift} & \\
\delta_{\text{ext}} & \\
\delta_{\text{int}} &
\end{align*}
\]

The schema $\delta_{\text{ext}}$ contains all external transitions of the lift $\phi$DEVS model.
The schema \( \delta_{int} \) contains some internal transitions of the Lift \( \phi_{DEVS} \) model.

5.4. Formal reasoning on DEVS model of the lift control system using Z

The proof obligation as mentioned in section 3.2 consists in (1) initialization theorems and (2) precondition calculations.

1) Initialization theorem proof of the Lift controller.

Theorem CanInitLift

\[ \exists Lift' \bullet InitLift \]

⇔

\[ \exists s': S_{\delta_{DEVS}} \bullet \]

\( s'.phase = Init \land s'.Engine = Stop \land s'.Door = closed \land s'.CurrentPos = 0 \land s'.ReqPos = 0 \land s'.Engine = stop \lor \]

\( s'.phase = Init \land s'.Engine = Stop \land s'.Door = closed \land s'.CurrentPos = 0 \land s'.ReqPos = 0 \land s'.Door = closed \lor \]

\( s'.phase = Init \land s'.Engine = Stop \land s'.Door = closed \land s'.CurrentPos = 0 \land s'.ReqPos = 0 \land s'.Door = closed \lor \]

\( s'.phase = Init \land s'.Engine = Stop \land s'.Door = closed \land s'.CurrentPos = 0 \land s'.ReqPos = 0 \land s'.Door = closed \lor \]

\( s'.phase = Init \land s'.Engine = Stop \land s'.Door = closed \land s'.CurrentPos = 0 \land s'.ReqPos = 0 \land s'.Door = closed \lor \]

Thus, the initial state of the lift is true.

We use here the regular definition expansion technique to rewrite the proof obligation, using the concerned state and operation schemas. Then we simplify our calculation by putting all conditions at the same level(i.e. removing the local existential quantifiers) and replacing the dotted variables by their values.

2) Precondition calculations[Saaltink 1999]

Z provides a reference to the precondition of an operation schema; for a schema \( Op \equiv [\Delta S; \ \text{in?} \ : \ \text{IN} \ ; \ \text{out!} \ : \ \text{OUT}] \) the schema reference pre \( Op \) is equivalent to \( \exists S'; \ \text{out!} \ : \ \text{OUT} \bullet Op \), and describes the initial states for which an output and a final state are possible. A precondition theorem, of the form \( \forall S; \ \text{in?} \ : \ \text{IN} \ | \ \text{pre Op} \), where \( P \) gives the precondition. In the example of the lift controller, we have to calculate Pre \( \delta_{int} \) and Pre \( \delta_{ext} \) and prove that they are true. For example, the \( S'(\text{target}) \) in \( \delta_{ext} \) satisfies the invariant property(\( \text{engine} \neq \text{stop} \Rightarrow \text{door}=\text{closed} \)). Hence, precondition will be the “Source” state and the \( \text{COND}_{\delta_{ext}} \). Thus, the precondition theorem will be true and the operation can be executed. In fact,

\[ \text{pre } \delta_{ext} \equiv \exists Lift' \bullet \delta_{ext} \]

Expansion:

\[ \text{pre } \delta_{ext} \equiv \exists s,s': S; \ \exists x?: X \bullet \]

\( (s.phase = \text{waiting} \land s.Engine = \text{stop} \land s.Door = \text{closed}) \lor \]

\( (s'.phase = \text{EvtRecept} \land s'.Engine = s.Engine \land s'.Door = \text{opened} \land s'.ReqPos = x?.in) \) [from \( \delta_{ext} \) schema]

\( (s.Engine = \text{stop} \land s.Door = \text{closed}) [\text{Invariant}] \land (s'.Engine = stop \land s'.Door = closed) [\text{Invariant}] \)

Reducing:

\[ (s.phase = \text{waiting} \land s.Engine = \text{stop} \land s.Door = \text{closed}) \lor \]

\( (s'.phase = \text{EvtRecept} \land s'.Engine = s.Engine \land s'.Door = \text{opened} \land s'.ReqPos = x?.in) \)

Precondition schema:

\[ \text{pre } \delta_{ext} \equiv \{ \text{Lift} \mid (s.phase = \text{waiting} \land s.Engine = \text{stop} \land s.Door = \text{closed}) \} \]

Proof:

Theorem \( \delta_{ext} \) Precondition

\( \forall \text{Lift}; !?: X \mid s.phase = \text{waiting} \land s.Engine = \text{stop} \land s.Door = \text{closed} \land (s.Engine = \text{stop} \land s.Door = \text{closed}) \bullet \]

\[ \{ \text{lift} \mid s.phase = \text{waiting} \land s.Engine = \text{stop} \land s.Door = \text{closed} \} \]

Result:

True

6. CONCLUSION AND FUTURE WORK

In this paper, we showed how to express, in a simple way, invariant (safety) properties on DEVS atomic model by adding an element to the classic structure, \( \phi \), and then translating the extended DEVS model(\( \delta_{DEVS} \) model) to Z specification schemas. This translation is beneficial to DEVS since it offers to it a logic semantics. Having a Z specification of DEVS model enables the theorem-proving techniques with the tools already available. These tools could be used to explore properties, often detecting conflicts between different requirements or missing assumptions. A such job improve the V&V process. Furthermore, our approach will help the users of DEVS to validate their models not only with simulation, but also with formal proof especially when the system, modeled with DEVS, contains critical properties called safety properties.

DEVS supports also complex models (or coupled models). These models are built from DEVS atomic models connected together in a hierarchical fashion. If the
DEVS atomic models are valid, then the global model is valid too. We are working to verify if the φDEVS coupled model can be valid (coherent) when the φDEVS atomic models are validated by using our approach. In addition, we will study the possibility and the interest of translating a Z specification to DEVS model.

7. REFERENCES


Mostafa Madiesh and Guido Wirtz
Distributed and Mobile Systems Group, University of Bamberg
Feldkirchenstraße 21, 96052 Bamberg, Germany
{mostafa.madiesh | guido.wirtz}@uni-bamberg.de

Abstract

A business-to-business (B2B) process is a process, in which different business partners interact with each other to achieve a common goal. In this paper we propose a top-down method to design B2B process in the context of service oriented architectures (SOA). In this method the Web Services Choreography Description Language (WS-CDL) is used to describe the global behavior of a B2B process, which serves as contractual basis for the collaboration between business partners. The local behavior for the individual business partners can be generated from the WS-CDL specification using the abstract Web Services Business Process Execution Language (WS-BPEL). WS-BPEL is used in this step solely to describe externally observable message exchange behavior of each business partner involved, without revealing their internal implementation. These generated abstract WS-BPEL processes are the starting point for implementing new local business processes. Consequently, before deployment the implemented WS-BPEL processes have to be proved consistent with the generated abstract WS-BPEL processes.

Keywords: B2B, SOA, WS-CDL, WS-BPEL, BPMN, compatibility, Petri nets

1. Introduction

Nowadays companies tend to become bigger and bigger through mergers and takeovers, which leads to ever more complicated processes to integrate different branches and aspects inside modern companies. Despite their size, businesses ties with other companies are still inevitable. Both kinds of development lead to the same sort of (integration) problems caused by incompatibilities ranging from, e.g., different contract handling and overall processes to different software environments, data formats and infrastructure. Although the former case – different branches of the same enterprise – theoretically may be open to uniform processes and systems, this is hard to achieve in practice due to the speed of changes and size of the problem. So, using the standard characterization of Business-to-business (B2B) processes as processes, in which different business partners interact with each other to achieve a common goal, these partners may be different companies as well as different branches or departments of the same business. The common point being that normally there is no detailed knowledge about the process as a whole at a single partner. In fact, each of the partners has detailed knowledge about her own part but only some insight in those she interacts with.

A suitable abstraction technique is essential here to determine which knowledge should only be known internally and how much information has to be published among partners to ensure an overall successful collaboration. Hence, each B2B process has (at least) two aspects: a global process view and different local process views. In the global process view, interactions between the business partners and the dependencies between these interactions should be described and no information about the internal process logic of business partners is included. In the local process view each business partner involved in a B2B process describes the interactions shared with other business partners, and additional internal steps between these interactions, which are invisible to the other business partners.

For the underlying basic technology, standards for interaction and description of messages and functionality like those provided by the web service standards for implementing service oriented architectures (SOA) are of great help which explains their enormous success over the last years [1].

Unfortunately, this does not hold to the same degree for standards for the higher levels of business processes. To design the more technical levels of a B2B process using SOA standards, the Web Services Business Process Execution Language (WS-BPEL) [2] can be used to describe the local process views and the Web Services Choreography Description Language (WS-CDL) [3] can be used to describe the global process view, which may serve as the contractual basis for the collaboration between business partners. However,
Section 2 discusses some closely related work. Section 3 describes and illustrates our top-down method in more detail.

In this paper we propose a complementing top-down method to design the technical levels of B2B processes using SOA standards that is part of our efforts to built a workbench for implementing B2B processes using SOA technologies.

In this method (see figure 1 for an overview) the design begins with the description of the global behavior of the B2B process using WS-CDL. Out of this information, an abstract WS-BPEL process for each business partner involved is automatic generated from the WS-CDL specification. After generating the abstract WS-BPEL processes, the compatibility between the generated abstract WS-BPEL processes has to be ensured. Compatibility means that the local business processes interact properly, i.e., they are free of deadlocks and there are no messages being sent that cannot be properly received or vice versa. After ensuring the compatibility between the abstract WS-BPEL processes, each business partner implements the detailed local executable WS-BPEL process for herself. Based on the implementation of all abstract WS-BPEL processes, it is verified if each implemented WS-BPEL process is consistent with its abstract WS-BPEL process. After ensuring the consistency between the abstract WS-BPEL processes and the executable WS-BPEL processes, the executable WS-BPEL processes may be deployed in a suitably BPEL engine. Note, that although the discussion here is confined to the (simpler) top-down case, many of the techniques and tools discussed here may also be deployed in bottom-up and mixed-mode development which is beyond the scope of this paper.

The remainder of this paper is structured as follows. Section 2 describes some closely related work. Section 3 describes and illustrates our top-down method in more detail. Finally, section 4 summarizes the main results and discusses routes for ongoing and future work.

2. Related Work

There are lots of approaches to design processes in a service-oriented setting; [14] provides a valuable overview of a spectrum of methods and techniques used. Proposals more closely related to our work are the following: In [15], a three-level approach is presented to map RosettaNet Partner Interface Processes (PIPs) to WS-BPEL processes: First, PIPs are transformed to abstract WS-BPEL processes templates capturing the message exchange and behavioral pattern of each business partner involved, then full valid abstract WS-BPEL processes are created from these templates representing a specialization of that pattern, and finally each business partner involved implements the full valid abstract WS-BPEL processes independently. In [16], a top-down approach is presented to modeling global behaviors of composite web services. In this method, first the desired global behavior of composite web services is described with a so-called conversation protocol specified by a realizable Büchi automaton, then system goals on the global conversation protocol, which are described with linear temporal logic (LTL), are verified. Finally, the implementation of each business partner involved in the conversation protocol is synthesized from the Büchi automaton via projection. An informal approach to design inter-organizational workflows based on a workflow setting is presented in [17]. This approach consists of three steps. In the first step the common public workflow is specified with workflow net, which serves as a contract between the organizations involved. In the second step the workflow net of the common public workflow is partitioned the over the organizations involved. In the third step a private workflow is created for each organization which is a subclass of the respective part of the common public workflow. In contrast to these approaches, our work puts its focus on using SOA standard description languages and to provide integrated tools to generate and/or to check consistent process descriptions which obtain an overall ‘compatible behavior’.

There are lots of approaches focussing on notions of ‘compatibility’ between processes and their verification in the context of B2B interactions. For a recent overview, we refer to [18].

In [19], a process-algebraic approach is described to verifying process interactions for business collaboration described in BPMN [20]: First, a semantic model for BPMN described in Hoare’s CSP and then the semantic model is used to verify the compatibility between the business processes involved in the collaboration.

In [21] a method is proposed to verify the compatibility between existing WS-BPEL processes. In this method first each WS-BPEL process is transformed to a BPEL annotated Petri net called BPN, then all BPN nets are composed and then the composed BPN is transformed into a communication graph which is a directed, bipartite graph representing the external visible behavior of a process. Finally the communication graph is analyzed to verify the compatibility between existing WS-BPEL processes.

In [22], an approach to verify, if there is a partner process for an existing WS-BPEL process, such that both can interact properly, is presented. Here, the WS-BPEL process is transformed into a Petri net model, too. Based on this model, the existence of a partner process is decided.

Although we use similar techniques and even adopt some
of them for checking compatibility (cf. section 3), none of these approaches provides an integrated environment to support the entire development process.

3. The Top-Down Method

The top-down method (see figure 1 for an overview) comprises the following six phases which may be iterated.

**Phase 1:** Describing the global behavior of the B2B process with BPMN for WS-CDL

The business partners involved agree on a common global behavior, which serves as a contract for the overall B2B process. This common global behavior is provided after this phase in WS-CDL. As a graphical language to provide this specification, however, we use the Business Process Modeling Notation [20] as it is a language much closer to the business analysts that are assumed to provide this kind of global information. To support this, we have developed a new tool called "WS-CDL Creator" (Figure 2) which supports modeling of WS-CDL using BPMN by generating a complete WS-CDL specification code from the BPMN model provided afterwards; for a detailed description of the mapping, refer to [23].

**Phase 2:** Generation of abstract WS-BPEL processes from WS-CDL specification code

For each business partner involved, an abstract WS-BPEL process is generated from the WS-CDL specification built in phase 1. To this aim, a mapping from WS-CDL to WS-BPEL like that presented in [24] is used, in which a prototype of this mapping is implemented as an XSLT transformation.
program. The basic idea is that for each role in the WS-CDL specification, a WS-BPEL process is generated and each element in the WS-CDL specification is added only to WS-BPEL processes of those roles for which it is relevant. The input of this transformation is a single WS-CDL document and the output consists of two or more abstract WS-BPEL processes for each business partner involved in the WS-CDL specification, one or more partnerLinkTypes representing the communication relationships in WS-BPEL processes, multiple property and propertyAlias definitions related to WSDL interfaces (see figure 3). At the end of this phase each business partner has an abstract WS-BPEL process, which is the starting point for implementing the detailed new business process.

Phase 3: Verifying the Compatibility between the abstract WS-BPEL processes

After generating the abstract WS-BPEL processes, the compatibility between the generated abstract WS-BPEL processes has to be ensured. Compatibility means that the local business processes interact properly, i.e., they are free of deadlocks and there are no messages being sent that cannot be properly received or vice versa. To verify that the abstract WS-BPEL processes are compatible with each other, we adapt the method proposed in [21]. Starting from two or more WS-BPEL processes, the following steps are performed (see figure 5):

1) Each WS-BPEL process is transformed into a so-called BPEL-annotated Petri net (BPN). In this transformation each element of the WS-BPEL process is mapped into modular Petri net pattern which are grouped together according to the structure of the WS-BPEL process and the result of this transformation is
Fig. 5. Verifying Compatibility between WS-BPEL Processes

Fig. 4. Communication Graph (cf. [25])

called BPEL-annotated Petri net (BPN). The mapping of WS-BPEL to Petri net abstracts from data aspects.

2) All resulting BPNs are reduced to speed up analysis and then the reduced BPNs are composed as a single BPN, which is again reduced.

3) The composed BPN is transformed to a communication graph (c-graph) [25] representing the external visible behavior of the composed BPN. A c-graph (see figure 4) is a directed, bipartite graph containing visible nodes (drawn as white ellipses) and hidden nodes (drawn as black circles). Visible nodes model the reachable states of the workflow net. Each outgoing edge from a visible node is labeled with a message which the workflow net is able to receive in at least one of those states. The hidden nodes model the intermediate state of the workflow net. Each outgoing edge from a hidden node is labeled with a message which the workflow net is able to send.

4) Finally, the communication graph is analyzed and the result is visualized. If the WS-BPEL processes are compatible, the user is just informed. Otherwise, erroneous system traces are provided for the user to analyze the obtained problems.

The four steps are implemented in an eclipse plug-in and interfaced to the rest of our tool set.

Phase 4: Implementing the abstract WS-BPEL processes

After ensuring the overall compatibility between the abstract WS-BPEL processes, each business partner has to implement the local WS-BPEL process by adding private behavior and details of execution that are hidden in the abstract WS-BPEL process. At the end of this phase each abstract process is implemented by an executable WS-BPEL process. However, such an implementation may be not consistent with the corresponding abstract view. Hence, only after the fifth phase, where for each involved business partner the consistency between the behavior of his executable WS-
Wombat4ws checks are performed automatically by the tool BPEL processes (see figure 7). All transformations and consistency analysis between abstract and executable WS-BPEL processes are performed successfully. Note, that here is also potential for a more global redesign step back.

**Phase 5: Verifying consistency between executable and abstract WS-BPEL Processes**

We adopt the method proposed in [26] to perform the consistency analysis between abstract and executable WS-BPEL processes (see figure 7). All transformations and checks are performed automatically by the tool Wombat4ws [27].

1) The abstract as well as the executable WS-BPEL process are transformed into so-called Workflow nets (WF-net) [28] which is a class of the Petri net including special places modeling the sending and receipt of messages. Figure 6 shows a simple workflow net, in which first the message "a" is received and then the messages "b" and "c" are sent, where "?" and "!" is used to indicate receiving (sending) a message in CSP notation.

2) After the transformation, the WF-net of the abstract WS-BPEL process and the executable WS-BPEL are transformed into a communication graph (c-graph) [25]. Figure 4 shows the c-graph of the workflow net shown in figure 6.

3) Now, the c-graph of the executable WS-BPEL process and the c-graph of the abstract WS-BPEL process are compared to verify compatibility rules, like, e.g., that the executable WS-BPEL process accepts at least those messages the abstract WS-BPEL process accepts. If the c-graph of the executable WS-BPEL process simulates the c-graph of the abstract WS-BPEL process, then the executable WS-BPEL process is a consistent implementation the abstract WS-BPEL process.

If all behaviors of executable WS-BPEL processes are consistent to behaviors of the abstract WS-BPEL processes then the executable WS-BPEL processes interact properly, since we have ensured in phase 2 that the abstract WS-BPEL processes are compatible with each other. Otherwise, the executable WS-BPEL processes must be adapted until they are consistent with their abstract counterparts.

**Phase 6: Deployment of the executable WS-BPEL processes**

After ensuring that all executable WS-BPEL processes are consistent with the abstract WS-BPEL processes, the executable WS-BPEL processes are deployed in a suitable WS-BPEL engine. Then the set of interacting executable WS-BPEL processes is executed.

### 4. Conclusion and Future Work

In this paper we represented a top-down method to design B2B process using SOA standards, so that the design begins with the description of the global behavior of the B2B process in BMPN resulting automatically in a WS-CDL specification that is step by step checked and transformed into a set of compatible executable WS-BPEL processes. For doing so, we adapt Petri net-based methods from [21] and [26] and combine them with our own tool set based on Eclipse. This tool is part of a workbench for B2B process design based on SOA standards that is interfaced to other verification methods like, e.g., model checking [10], and also tries to support bottom-up and mixed integration-style development.

Ongoing and future work is dedicated to extend the functionality of the system, to support consistency issues on all levels more comfortably, to provide a better integration of the different aspects and tools as well as to conduct more rigorous case studies for B2B integration in real life scenarios like, e.g., those from [7], [8] and [9].

### References


Fig. 7. Verifying Consistency between an executable and an abstract WS-BPEL process.

- **Step 1:**
  - WS-BPEL to WF-net transformation
  - WF-net to c-graph transformation

- **Step 2:**
  - WS-BPEL to WF-net transformation
  - WF-net to c-graph transformation

- **Step 3:**
  - Verifying consistency

References:


Software Complexity for Computer Communication and Sensor Networks Using Binary Decision Diagrams
Harpreet Singh, Adam Mustapha, Arati M Dixit, Kuldip Singh, Grant R. Gerhart

Abstract—Software complexity has been a topic of interest for software engineers for the last three decades. Various definitions of software complexity have appeared in literature from time to time. This has resulted in the development of software metrics by different investigators. With the new developments in the area of computer communication and sensor networks, software complexity metrics need an improvement so that they can be used for internet, computer communication networks, and sensor networks. It is difficult to apply the existing definitions of software complexity to internet applications. The objective of the present paper is to propose a definition which could be used for internet and sensor networks applications. In this paper a new definition of software complexity is utilized for the proposed applications. It is suggested to use Binary Decision Diagrams for this purpose as these diagrams result in disjoint expression which is needed in the calculation of the software complexity. Simple cases of software complexity for series, parallel, series-parallel and non series-parallel networks are investigated. The simulation of the proposed algorithm is given in this paper. The complexity between two nodes of a graph can be determined with the help of path expressions between the two nodes of interest of a graph. Binary Decision Diagram approach is then utilized to determine the complexity between two terminal nodes of a graph. The approach is applicable to any number of nodes and branches of a graph. It is hoped that this approach will be extensively used by software engineers who are developing software for internet, computer communication networks, and sensor networks.

Harpreet Singh, hsingh@eng.wayne.edu
Adam Mustapha, df5088@wayne.edu
Arati M Dixit, as7623@wayne.edu
Department of Electrical and Computer Engineering, Wayne State University Detroit, MI 48202

Kuldip Singh with Department of Electrical and Computer Engineering, IIT Roorkee, India, 247667

Grant R. Gerhart, gerhart@us.army.mil, U.S. Army Tank Automotive Command Research, Development and Engineering Center, Warren, MI 48088

Index Terms—software complexity, software reliability, fuzzy logic, binary decision diagrams, boolean algebra, MATLAB.

I. INTRODUCTION

Software complexity is a very important metric used in software engineering. While developing software it is important to predict the complexity of software in advance. If the complexity is more than the desired complexity, changes can be made in the development process to obtain the prescribed complexity. It is important to know this metric in the early stages of the software development. Knowing complexity at later stages results in high development costs of the software. Because of these reasons there has been on ongoing research in determining various definitions of complexity. With increasing popularity of internet, the complexity definitions have to be modified so as to meet the increasing demand of software development for internet applications. The existing definitions of complexity are not suitable for internet application. In this paper, a new definition of software complexity is suggested so that this definition can be applied to internet software development [1]. Here in the software complexity is defined from zero to one rather then the previous definition of number of lines of code, number of operators and operands, number of loops, etc. Halstead [2], [28] defined software complexity in terms operators and operands. McCabe [3] defined the software complexity in terms of the number of loops in the software. This was followed by a number of other workers [4]-[6] who gave different definitions of software complexity. Table I. has various definitions of software complexity.

In order to define software complexity between zero and one, we suggest Boolean algebra approach here. We essentially suggest the same approach which has been used by several investigators in determining the reliability of computer communication network [7]-[16]. The same approach can be useful for internet and
sensor network applications, so long as the problem can be represented as a graph, these approaches can be applied. In this paper we suggest Binary Decision Diagrams approach to determine the complexity of sensor network. Binary Decision Diagrams became popular when the conventional truth table approaches could not be applied for a large number of variables. With the coming of internet and sensor network, a number of nodes and branches of under consideration have become extremely large. Because of the fact that Binary Decision Diagrams approach can be used for a large number of variables, hence such an approach is suggested in the present work.

### TABLE 1

**SOFTWARE COMPLEXITY**

<table>
<thead>
<tr>
<th>Authors/Year</th>
<th>Software Complexity</th>
<th>In Terms of</th>
</tr>
</thead>
<tbody>
<tr>
<td>McCabe 1976</td>
<td>Cyclomatic complexity</td>
<td>Loops</td>
</tr>
<tr>
<td>Halstead 1977</td>
<td>Length of program and Effort</td>
<td>Operands and Operators</td>
</tr>
<tr>
<td>Ramamorothy and Melton 1988</td>
<td>Family of measures</td>
<td>Weighted measures</td>
</tr>
<tr>
<td>Lisa Anneberg 1991</td>
<td>Patri net 'A' matrix complexity</td>
<td>Order of 'A' matrix</td>
</tr>
<tr>
<td>Munson and Khoshgoftaar 1992</td>
<td>Relative complexity</td>
<td>Factor Analysis</td>
</tr>
</tbody>
</table>

### III. REVIEW OF SOFTWARE COMPLEXITY OF SIMPLE NETWORKS

The software complexity of circuits has been studied for the last four decades extensively in the literature [2]-[6], [10], [20]-[22]. The software complexity of branches in series, branches in Parallel, and branches in non-series parallel has been the subject of research papers [1].

Currently, sensors networks have become very popular in literature because of their applications in different areas. The purpose of this paper is to utilize how Binary Decision Diagrams can be useful in determining the software complexity for computer communication and sensors networks. For completeness, a review of Binary Decision Diagrams is briefly given first. Fig. 2 illustrates the software complexity of paths in series, in parallel, in series-parallel and non-series parallel. Both paths and binary decision diagrams for each case are given.

More complex software is less reliable and vice versa. The software reliability has a range of zero to one; hence software complexity is also defined as to have a range of zero to one. Further, these definitions are extended so that the complexity of interconnected network can be defined.

Consider two branches in series having software complexity as 0.9 and 0.8. The uncomplexity (reliability) of these branches will be 0.1 and 0.2 respectively. The reliability (uncomplexity) of series path will be equal to $0.1 \times 0.2 = 0.02$. Hence the complexity will be of the series path will be $1-0.02 = 0.98$.

Next consider two parallel paths having complexity 0.9 and 0.8. Their uncomplexity (reliability) will be 0.1 and 0.2 respectively. So the reliability (uncomplexity) of the two paths in parallel will be equal to $0.1 + 0.2 - 0.1 \times 0.2$ which will be equal to 0.28. Hence the software complexity of two parallel paths is $1-0.28 = 0.72$.

Next consider two series parallel paths (x1 and x2 in parallel with x3 and x4) having complexities equal to 0.9 and 0.8 for the series path and 0.7 and 0.6 for the other path in parallel. The complexity of the series path will be 0.98 and 0.88. The
uncomplexity of the two parallel paths are 0.2 and 0.12 respectively. The overall uncomplexity is 
0.02 + 0.12 − 0.02 * 0.12 = 0.1376. Hence the overall series parallel complexity will be equal to 
1 − 0.1376 = 0.8624.

We cannot find out complexity of non-series parallel network by simple calculation. However by using Boolean Algebra technique and by finding a disjoint (non-overlapping) expression one can determine the terminal complexity of the non-series parallel network as 0.835.

The terminal software complexity value of each network is given in a Table 2. These values can be easily determined with the help of conventional techniques given in reference [1]. However, in this paper these values can be determined through Binary Decision Diagrams and with the software developed using Fuzzy Logic [24]-[26] MATLAB toolbox [27].

<table>
<thead>
<tr>
<th>Graph</th>
<th>Terminal software complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_1 \rightarrow c_2$</td>
<td>0.98</td>
</tr>
<tr>
<td>$c_1 \rightarrow c_2$</td>
<td>0.72</td>
</tr>
<tr>
<td>$c_1 \rightarrow c_2 \rightarrow c_3 \rightarrow c_4$</td>
<td>0.8624</td>
</tr>
<tr>
<td>$c_1 \rightarrow c_2 \rightarrow c_3 \rightarrow c_4$</td>
<td>0.835</td>
</tr>
</tbody>
</table>
III. TERMINAL SOFTWARE COMPLEXITY (TSC) USING BINARY DECISION DIAGRAMS (BDD)

The terminal software complexity is the software complexity between two terminal nodes. Given a network, it is important to determine branch software complexity between various nodes of a network. The problem is a general one and has applications in several areas such as communication systems and computer programs. Knowing the complexity of each branch, one can determine the terminal software complexity between any two nodes. The Boolean algebra approach has been used by several investigators [10], [14], in determining the terminal reliability of a network. The Boolean algebra technique basically consists of the following steps:

1. Determine simple paths between two nodes of a graph.
2. Write down the Boolean expression corresponding to the paths where the Boolean variables correspond to the different branches.
3. Determine a disjoint expression corresponding to the Boolean expression given in step 2.
4. Given the disjoint Boolean expression, substitute the corresponding values of branch software uncomplexity.
5. The terminal software complexity = 1 - uncomplexity

V. ALGORITHM FOR TERMINAL SOFTWARE COMPLEXITY USING BINARY DECISION DIAGRAMS

The following steps are used for drawing Binary decision Diagrams.

1. Determine simple paths.
2. Determine the Boolean expressions which correspond to the simple paths.
3. Use Shannon’s expansion theorem to mark the paths.
4. Determine the non-overlapping expressions.
5. Substitute branch software uncomplexity values
6. The terminal software complexity = 1 - uncomplexity.

VI. SOFTWARE IMPLEMENTATION

We have developed software which can determine the software complexity between any two terminal nodes of graph. The approach is applicable for a graph having any number of nodes and branches. The software has been developed based on the concept of Binary Decision Diagrams.

The form below in Fig. 3 is the starting form of the application. This form allows the user to enter

- number of wires
- number of nodes
- number of bidirectional wires
- begin node of the desired path
- end node of the desired path

Desired path is the path for which the complexity is to be calculated.

Fig. 3 shows MATLAB displays for terminal complexity for series parallel network with 4 nodes 4 wires example.
VI. CONCLUSION

In this paper a new technique defining software complexity has been proposed. Contrary to the previous definitions, we suggest a new definition of software complexity where the values of complexity lie between zero and one. Such definition helps in determining the software complexity for internet application. This approach how to determine complexity requires disjoint Boolean expression. In this paper BDD are suggested to obtain disjoint or non-overlapping expression. The proposed procedure has been implemented using MATLAB. The procedure is applicable for any number of nodes. It is hoped that this approach of determining software complexity will be utilized by several investigators who will be developing software for internet and sensor applications. Determining software complexity at the earlier stages of the software development goes a long way in reducing the overall cost of software development.

REFERENCES


Model Checking Consistency between Sequence and State Diagrams

Kuang-Nan Chang
Department of Computer Science
Eastern Kentucky University
Richmond, Kentucky, USA

Abstract – This paper presents a method for detecting consistency defects of UML diagrams with model checking. The method verifies the consistency between state diagrams and sequence diagrams. Sequence diagrams are simulated as system specification, while state diagrams are used to set desired system properties. Model checking is then used to verify the properties with system specification to find any consistency defects between them.

Keywords: UML, Sequence Diagram, State Diagram, Model Checking, and Consistency Defects.

1 Introduction

Although the various diagrams of UML enable UML to describe different aspects of a software system, the diagrams also cause the consistency problems among themselves. Especially when software systems are big, preventing consistency defects can become a slow, laborious task.

In their experiments [10], Lange and Chaudron observed two groups of people how they applied UML to their projects. Their experiment results showed that consistency defects were easy to make, and these defects had a great influence on the later interpretation of the diagrams. Lange and Chaudron studied these defects and classified them into different categories, so that people could use their study to avoid same or similar consistency defects in their UML models. In their other research [9], Lange and Chaudron provided a definition of completeness of a UML model, and set some rules to assess a model’s completeness. The rules were set for checking the relationships of use cases/classes and classes/message sequences. Like many other methods (ex., [3, 5]), the consistency/completeness checking methods in both of the Lange and Chaudron’s papers rely on manual operations, and the focus of these methods is those diagrams (such as use case, domain, and sequence diagrams) that are for the early software development phase.

Boris Litvak et al. [11] introduced an algorithmic method to check consistency between sequence and state diagrams. In their method, the states in a state diagram are associated with the object lifelines in different sequence diagrams, so that their algorithm is able to detect any inconsistencies between the diagrams of the same object.

Alanazi et al. [1] argued that Litvak’s approach could only be applied to one object at a time. Multiple executions of the algorithm must be applied in order to check the consistency of an entire sequence diagram. Alanazi et al. [1] presented a matrix approach to check the consistency between the diagrams. In their approach, a transition matrix is introduced to represent the state of more than just one object. The closure of the matrix is able to determine reachable and unreachable states. These states are used to validate class diagrams, and, then, to validate sequence diagrams.

Some researchers introduce formal methods to detect inconsistencies between state and sequence diagrams of UML. For example, Lam and Padget [8] used the π-Calculus, and Mens et al. [12] used description logic. It needs some mathematic background and skills to learn and use their approaches to specify and validate the UML diagrams.

This paper introduces an approach to check the consistency between sequence and state diagrams with model checking technique. The consistency checking is applied to individual object. The approach collects all the messages (function calls) and their execution sequences of an object from all the sequence diagrams. Then, the state diagram of the same object is used to set desired system properties to verify all possible combinations of the collected execution sequences. With model checking approach, consistency defects between the state diagram and the sequence diagrams can be detected easily and automatically.

The rest of the paper is organized as follows. Section 2 discusses related work of the paper, including the features of sequence diagram and state diagram of UML, and the SPIN model checking tool. Section 3 describes the kind of consistency defects the paper focuses on. Section 4 explains the proposed approach and illustrates its use with an example. Section 5 concludes the paper.
2 Background

2.1 UML State and Sequence Diagrams

In UML, state diagrams and sequence diagrams are used to specify the dynamic behavior of a system. A state diagram shows the behavior of an object through many use cases of a system. It shows all of the possible states of an object as events happen. A state diagram includes states and transitions. A state represents a stage in the behavior pattern of an object. An initial state is a special state in which an object is first created. A final state is also a special state in which no transitions lead out of an object. A transition is triggered by an event to progress an object from one state to another.

A sequence diagram includes parallel vertical lines and horizontal arrows. The vertical lines represent the existences of different processes or objects simultaneously. The horizontal arrows represent the messages exchanged between the processes or objects. A sequence diagram specifies a runtime scenario, in which the interactions between objects and the message passing between objects are specified in a particular order.

2.2 Model Checking and SPIN

Model checking has been recognized as a practical and effective verification technique. Using model checking, the system’s behavior is modeled by a set of concurrent communicating state machines. A desired property to be checked for satisfiability is stated as a temporal logic formula. The verification process exhaustively explores the state space derived from the movements of the state machines. The state space keeps the information about different states and their occurrence sequences. Model checking verifies a desired property with these sequences of states, and tries to find a counterexample to disprove the property. Model checking is effective, because it applies an exhaustive search for a counterexample in the state space. It is practical, because the exhaustive search is done with an automatic mechanism.

The SPIN model checker [6] is a generic verification system. It supports the design and verification of asynchronous process systems. It has been used successfully in industry for verifying both hardware and software systems. For verifying a software system with SPIN, two specifications are needed: the specification of system behaviors in Promela (a language for specifying system behaviors with processes and channels) and the specification of desired system properties in Linear Temporal Logic [7] (LTL, a logic that consists of temporal operators for the sequence consideration of events). With these two specifications, SPIN is able to verify the behavior specification against the desired properties of a system. If the behavior of the system cannot satisfy any of the desired properties, SPIN will show an error message and point the location where the error happens in the specification.

3 Inconsistency between Sequence and State Diagrams

Since the behavior of an object can be described with a state diagram, and its interactions with other objects can be specified with different sequence diagrams, the dynamic behavior of the object must be kept consistently in these diagrams. Although a state diagram may be nondeterministic, the sequences of transitions in the diagram imply the orders of the message passing in different sequence diagrams. That is, tracing the messages of an object in the sequence diagrams should reflect the changes of states for the object in the state diagram. A consistency defect happens when an object reaches some state whose outgoing transitions do not agree with the orders of message passing for that object in any sequence diagram. The example in Figure 1 shows such a defect.

Let’s assume that the state diagram SD1 describes the behavior of some object o1, and SEQ1 and SEQ2 show only the message passing of o1 in two sequence diagrams. If we intend to execute the scenarios in both sequence diagrams, there will be different sequences to execute them. We might execute SEQ1 first and SEQ2 second, or we might execute SEQ1 twice first, and, then, SEQ2. Notice that in both cases, after executing SEQ1 and SEQ2, the object is supposed to be in state S2 according to the state diagram SD1. That means the next event for object o1 to perform must be e2(), which is not the first message call in both sequence diagrams. Hence, a consistency defect happens if we try to execute any one of SEQ1 or SEQ2 after the execution of SEQ2.

This kind of consistency defects may happen between a state diagram and one or more sequence diagrams. Like the one in Figure 1, the defect concerns two sequence diagrams. To detect such kind of defects, this paper suggests using model checking to find the defects and their locations in the diagrams.
4 Consistency Check with SPIN

4.1 Specification with SPIN

To detect consistency defects between sequence diagrams and state diagram of an object, the message passing (function calls) of the object in different sequence diagrams are simulated with as many number as processes of Spin. For example, in Figure 1, the object o1 in the SEQ1 sequence diagram has two function calls. The behavior of o1 in this diagram can be specified with SPIN as follows:

1. proctype SD1 () {
2.   SD11:
3.     SD1chan?ack;
4.   SD12:
5.     o1Event = e1;
6.     o1Event = e2;
7.     if
8.       ::atomic { SD2chan!ack; goto SD11;}
9.     ::atomic { goto SD12;}
10.    fi;
11. }

Line 1 is the name of the process that simulates the diagram SEQ1 in Figure 1. Line 3 represents the beginning of the timeline of the object in the diagram. Lines 5 and 6 show the execution sequence of the messages e1() and e2(). The following if statement randomly picks next scenario (sequence diagram) to perform. Figure 2 is a complete specification for SEQ1 and SEQ2 of o1 in Figure 1.

Figure 2. System Specification of o1 in Promela

Notice that the last part of the code is the main function, init, which initializes the two processes (SD1 and SD2) and picks one of them to begin the whole simulation.

The state diagram of an object is used to set up desired system properties in the proposed method. The properties will be verified with system specification like the one in Figure 2. The SPIN is able to detect any inconsistency between the properties and the specification.

Figure 3. Property Specification of o1 in LTL

Figure 3 shows the properties derived from the state diagram of the object o1 in Figure 1. At first different variables are set to represent different events in the diagram.

#define p o1Event==e1
#define q o1Event==e2
#define r o1Event==e3
#define s o1Event==e4

A desired property shows the desired execution sequence of events. For example, SD1 shows that after executing the event e1(), e2() must be the next event to be executed. To specify this property, the temporal operator U (until) is used to express such an execution sequence as follows:

p -> (p U q)

This expression says that when the event e1() (variable p) has been executed, the next event to be executed is e2() (variable q), and there are no other events happened between them. From the diagram, four desired properties are specified and connected with the AND operators as follows:
Notice that another temporal operator \[ \[] \] is attached to the beginning of the expression, which requests that the properties in the expression need to be always true all the time.

### 4.2 Verification Result

By pressing the Run Verification button in Figure 3, SPIN starts verifying the specified property in Figure 3 with the system specification in Figure 2. Figure 3 also shows the result of the verification, which indicates that there is an error in the verification (in the very last block of the window). Figure 4 shows the trace of the error with the system states generated by the SPIN, and Figure 5 shows the value changes of the variables in the system states.

![Figure 4. Sequence of System States for Tracing Errors](image)

In the state diagram \( SD1 \), the event \( e4() \) must be followed by \( e2() \). The last property, \( s \rightarrow (s \ U q) \), in the property expression specifies such a relationship between the events. Figure 4 shows that there is an infinite loop of system states which prevents this relationship from happening. Figure 5 indicates that, after the execution of \( e4() \), an infinite loop of executing \( e3() \) and \( e4() \) follows, and \( e2() \) could never happen after \( e4() \). Hence, a consistency defect between the state diagram and the two sequence diagrams in Figure 1 has been detected by the SPIN model checker.

![Figure 6. Revised System Diagram for Object o1](image)

One possible correction is to replace \( SD1 \) in Figure 1 with \( SD2 \) in Figure 6. Then, the property \( s \rightarrow (s \ U q) \) will be changed to \( s \rightarrow (s \ U (p||r)) \), and the property expression will become:

\[
[] ((p\rightarrow(p \ U q)) \&\& (q\rightarrow(q \ U (p \ || \ r))) \&\& (r\rightarrow(r \ U s)) \&\& (s\rightarrow(s \ U q)))
\]

Figure 7 shows the verification result, which indicates that there is no error in the verification. That is, there is no inconsistency between the sequence diagrams and the new state diagram.

![Figure 7. Revised Property Specification of o1 in LTL](image)
5 Conclusions

Keeping consistency among the various diagrams of UML is always a great challenge and laborious task. Any consistency defects among the diagrams may cause misinterpretations in the later software development stages. Many researchers have proposed different methods to prevent or detect different types of consistency defects. Most of them are for checking the consistency between the diagrams that specify the static aspects of a software system. A lot of these methods still need manual operations to perform consistency check, which makes them impractical for the systems of large size.

This paper has presented an automatic verification method to detect consistency defects between two UML diagrams that are used to specify the dynamic behavior of a software system: sequence diagram and state diagram. The method uses the SPIN model checker to detect any inconsistency between these two types of diagrams. In the method, a system specification derived from sequence diagrams is verified with a set of desired system properties determined with a state diagram. If an error is detected, SPIN is able to show the cause of the error, which is also a consistency defect between the sequence diagrams and the state diagram. Hence, with SPIN, correcting such a defect becomes an easy job.

This paper has only discussed simple structures of sequence diagrams and state diagrams. For more complicated structures, such as loops in sequence diagrams and nested states in state diagrams, they may need to be specified in different ways with SPIN. A further study about those complicated structures of the diagrams remains the future work.

6 References


An Expert System for Pi-Calculus and Api-Calculus Automated Reduction

S. Rahimi, J. Dillards, and B. Gupta
Department of Computer Science, Southern Illinois University, Carbondale, Illinois, USA

Abstract - As a formal method, π-Calculus is a notation that provides the means to represent the system state and demonstrate behavior thus detecting problems with the system before the system is implemented. In 2002, Rahimi [1] introduced a new multi-process formal modeling language known as Api-Calculus which expanded the functionality of π-Calculus. Both types of Calculi rely on a complex system of interactions between their entities known as reduction to show the evolution, or change of state, of the system. Although these are useful tools for formally modeling multi-process systems, π-Calculus and Api-Calculus are difficult to reduce by hand and there exists no automated tool for their reduction. The goal of this work is to develop a system to perform this reduction regardless of whether the system is in π-Calculus or in Api-Calculus. This paper describes the under taken process to create an expert system to automate the reduction process. The presented system is the first system that solves the complicated syntax and semantics of π-Calculus and Api-Calculus and provides an automated mechanism for enforcing reduction.

Keywords: Formal Methods, Process Calculi, Expert Systems, Pi-Calculus, Reduction Rules

1 Introduction

π–Calculus, introduced by Milner [3], has become one of the most complete and widely used and researched tools for modeling multi-process systems [4]. π–Calculus is a notation that provides a means to not only represent the system state to state, but also demonstrate behavior and thus detect problems before the system is implemented. Since it is a formal modeling language, π–Calculus also provides the mathematical framework to formally represent a multi-process system's behavior as is predicted [2, 5].

π–Calculus represents a multi-process system as a universe, where abstract processes interact with each other by sending and receiving names that represent communication channels or variables. A communication channel can be anything that allows two processes to communicate with one another. A communication channel can be a variety of things from hyperlinks to coordinates of communication satellites.

The primary entities within π–Calculus are processes. Processes are not exclusively what processes are in an operating system sense of the term although they can be in certain implementations. Processes in π–Calculus include any entity that will communicate with other entities. Processes evolve to become different processes by performing actions. Processes interact with each other by sending names to one another across communication channels during interactions.

Any process can communicate with any other process that shares the same communication channel with one exception. π–Calculus has a system of restrictions placed on variables that as the name implies restricts the variables by only allowing other processes with the same restricted variable to use them. So for a process Q to interact with a process P over communication channel C both P and Q would either have to have C restricted or neither would.

In 2002, Rahimi [1] introduced an offshoot of π–Calculus called Api-Calculus. Api-Calculus has the same basic elements as π–Calculus including processes and restrictions. Api-Calculus provided three primary additions to π–Calculus. These contributions were milieus, knowledge units, and terms. For detail information regarding the syntax and semantics of these calculi please refer to [1,3].

So far the capability of π–Calculus and Api-Calculus in showing a snapshot of a mobile environment is discussed, but what about the representation of the behavior of such environment. The behavior of a system is represented by a series of actions, also known as reductions, which are governed by a set of rules known as reduction rules and a set of axioms known as the axioms of structural congruence. Reduction rules and axioms of structural congruence are shown in tables 1 and 2 respectively (where P, Q represent processes, M represents milieu, Oi represent either a process or a milieu, τ represents an internal action, K represents a knowledge unit, and x,u,z represent variables/channels). Reduction rules are designed to show how aspects of the system translate across a reduction which can be thought of as one step of a process's evolution.

Obviously there is a great deal of work already done in developing and utilizing different aspects of π–Calculus and Api-Calculus. They both can be used to successfully define and study the behavior of a distributed system. However, the work of manually running the system by running the reductions without the aid of a computer is not only extremely tedious but is prone to error. Naturally, it is much easier for a human to make a mistake reducing the system by hand due to the human probably knowing how the system should come out and subconsciously expecting it to do so. On the other hand, if a computer could run the reduction rules for the user, the human error is eliminated as is the tedious task of manually reducing the system.
that reduced either work incorporates a Java application to store the system and allow the addition of a graphical user interface. Working in conjunction with the Java application is an expert system used to reduce the system. This system utilizes the JESS [7] libraries to facilitate the expert system. It will integrate into a visualization tool called ACVisualizer [8] and provide an interface module for both the performance evaluation and the validation tools created by Chen and Rias [9, 10].

2 The Implementation Issues

Both $\pi$–Calculus and Api-Calculus are well defined formal modeling languages. However, they are not defined in a matter that makes them conducive to translation to code. Because of this, many issues arose during the implementation not apparent in the definitions of the languages. The first main category of issues is issues not defined which includes whether or not the processes keep variables in memory once they are sent on an outgoing channel and how the languages which are defined nontemporally can be transformed into a temporal program. The second category is issues that are not involving interpretation of the languages, but involve the semantics of its implementation. This group includes how to store the system state for display, where to store the systems structure and how to handle the difference in $\pi$–Calculus and Api-Calculus.

As stated earlier, the developed expert system handles both $\pi$–Calculus and Api-Calculus. The distinction made between the two is invisible to the user. Whether or not the system is treated as $\pi$–Calculus or Api-Calculus depends on whether or not the processes keep variables in memory once they are sent on an outgoing channel and how the languages which are defined nontemporally can be transformed into a temporal program. The second category is issues that are not involving interpretation of the languages, but involve the semantics of its implementation. This group includes how to store the system state for display, where to store the systems structure and how to handle the difference in $\pi$–Calculus and Api-Calculus. In essence the system is always running Api-Calculus, it just does not force the user into doing so explicitly.

There is certainly more than one way that this system could have been created. There were three primary choices considered for organization of the system. The most basic of these choices was to leave all the structure of the system and all of the reduction of the system in the Java code. An all Java system would have some definite advantages. The most obvious is there is no need to import or export any information in or out of the expert system. This would save a great deal of coding complexity as far as setting the system up to execute the reductions is considered. Having the entire system run in Java would also lessen the restrictions placed on naming processes and data since there would be no need to worry about the restricted characters of another system. However, there are also disadvantages to this type of system. Most notably the complexity of the system that processes the reductions would have to be very high. This type of system would necessitate having a method of considering all the verbs on all the processes for comparison to one another via the reduction rules. Basically something would have to scan all the verb data structures in all of the different objects within the system and then repeat this process each time a reduction is executed. Although it is a requirement that all the verbs be scanned it

Prior to this work there was no existing computer system that reduced either $\pi$–Calculus or Api-Calculus systems. This work incorporates a Java application to store the system and allow the addition of a graphical user interface. Working in conjunction with the Java application is an expert system used to reduce the system. This system utilizes the JESS [7] libraries to facilitate the expert system. It will integrate into a visualization tool called ACVisualizer [8] and provide an interface module for both the performance evaluation and the validation tools created by Chen and Rias [9, 10].

### Table 1. Reduction Rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAU</td>
<td>$\tau P + Q \rightarrow P$</td>
</tr>
<tr>
<td>REACT</td>
<td>$\left( u(z).P + P' \right) \mid \left( u(x).Q + Q' \right) \rightarrow P \mid Q$</td>
</tr>
<tr>
<td>PAR</td>
<td>$P_1 \mid P_2 \rightarrow P_1' \mid P_2$</td>
</tr>
<tr>
<td>RES</td>
<td>$P \rightarrow Q$</td>
</tr>
<tr>
<td>RES-K</td>
<td>$P \rightarrow Q$</td>
</tr>
<tr>
<td>MIL</td>
<td>$P \rightarrow Q$</td>
</tr>
<tr>
<td>STRUCT</td>
<td>$O_1 \rightarrow O_1'$</td>
</tr>
<tr>
<td></td>
<td>$O_2 \rightarrow O_2'$</td>
</tr>
<tr>
<td></td>
<td>If $O_1 = O_2$ and $O_1' = O_2'$</td>
</tr>
</tbody>
</table>

### Table 2. Axioms of Structural Congruence

<table>
<thead>
<tr>
<th>Axiom</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match</td>
<td>$[T=T] \pi P \equiv \pi P$</td>
</tr>
<tr>
<td>Summation Associativity</td>
<td>$O_1 + (O_2 + O_3) \equiv (O_1 + O_2) + O_3$</td>
</tr>
<tr>
<td>Summation Commutativity</td>
<td>$O_1 + O_2 \equiv O_2 + O_1$</td>
</tr>
<tr>
<td>Summation Identity</td>
<td>$O_1 + 0 \equiv O_1$</td>
</tr>
<tr>
<td>Composition Associativity</td>
<td>$O_1 \mid (O_2 \mid O_3) \equiv (O_1 \mid O_2) \mid O_3$</td>
</tr>
<tr>
<td>Same Process</td>
<td>$O + O = O$</td>
</tr>
<tr>
<td>Composition Commutativity</td>
<td>$O_1 \mid O_2 \equiv O_2 \mid O_1$</td>
</tr>
<tr>
<td>Composition Identity</td>
<td>$O \mid 0 \equiv O$</td>
</tr>
<tr>
<td>Restriction</td>
<td>$\nu T_1 \nu T_2 P \equiv \nu T_2 \nu T_1 P$</td>
</tr>
<tr>
<td>Restriction Identity</td>
<td>$\nu T 0 \equiv 0$</td>
</tr>
<tr>
<td>Restriction Composition</td>
<td>$\nu T \left( P_1 \mid P_2 \right) \equiv P_1 \mid \nu T P_2$, if $T \neq ft(P_1)$</td>
</tr>
<tr>
<td>Replication</td>
<td>$!P \equiv P \mid !P$</td>
</tr>
</tbody>
</table>

Although it is a requirement that all the verbs be scanned it...
would be very beneficial to find a way that the system would not have to manually do all this scanning.

One solution to this problem is to take advantage of the Rete algorithm [6] which is used in expert systems to very efficiently perform pattern matching. An expert system is a natural fit for processing the reduction rules of our system since it utilizes a system of pattern matching very efficiently. This will enable us to determine which verbs are ready to be reduced without having to perform the entire pattern matching manually. But using an expert system to store the system has downsides as well. The primary issue will be our desire to have a graphical user interface as the interface from the user to our system. Since having the system in Java made the graphical user interface easy to create in comparison to other systems it would be unreasonable to give up this advantage. Likewise the object oriented structure of Java naturally lends itself to the storage of our system so it is necessary to keep Java as the primary way of storing and interacting with the system.

So now the solution seems very natural to have the system stored in Java to take advantage of its object oriented structure as well as to utilize its graphical user interface capabilities. The processing of the reduction rules will be performed in an expert system to take advantage of the Rete algorithm for efficient pattern matching. This solution seems very storing but there is still a large downside. Having the reductions rules executed in a separate system from the storage and user interaction with the system will cause the need to translate this system between the two platforms. Although this problem is not trivial it is much more manageable than the problems created by either having the entire system in Java or the entire system in an expert system. Later in this paper, there is a description of how the system translates itself from Java to the expert system and back again.

Another problem that arose prior to creating the expert system to run the reduction rules was the procedure for handling new verbs coming into the system. Since \( \pi \)-Calculus is a non-temporal system it is not designed to distinguish where new verbs show up in the list of verbs ready to be executed. The first solution to this problem is to try to exactly emulate \( \pi \)-Calculus by recreating the entire list of verbs each time after the reduction rules have been run. In other words treat it non-temporally. Although this approach would be most accurate in showing how the \( \pi \)-Calculus is defined, it is not ideal to try to treat our temporal system non-temporally. Since we want to continue to treat our system temporally a system needs to be designed for executing the verbs left over after the reductions conclude execution.

The first approach is to cache the system each time after the expert system has run. This approach would involve removing all the verbs of all the processes and milieus in the system after the expert system concludes execution and storing all of these removed verbs in some sort of slush data structure. Then after all the new verbs finish reducing during the next running of the expert system all of the verbs are pulled out of cache and added to the system in an attempt to further reduce the system. This approach does have its advantages as the old verbs no longer become a hindrance to the execution of the verbs recently applied to the system. This eliminates the chance of one old verb completely debilitating a processes execution. In addition it does not simply discard the old verbs since the cache system will give them a chance to still be executed. However, this solution certainly is not perfect. One problem with this is that it does not accurately model the system. It allows verbs that have been added to the system following earlier verbs to execute first. For example a process could be attempting to receive a new value for a piece of data in it called A. It needs to receive its new value for A before it sends this to another process. So the user adds the receive verb on one iteration of the system and then adds the send verb on the second iteration of the system. However if in this scenario the receive verb does not get reduced, it will be put into the cache and the send verb will execute first causing an error in the intended execution of the system. Also since in \( \pi \)-Calculus it is possible for one verb to hold up the execution of the process and make it inactive, then it should be made the same way in this system as well.

One solution to solve this problem would be to use a stack data structure to store the verbs. This would work by having the user build up a succession of verbs on top of each process. The process would always simply pop the structure to get the next verb. The next verb would then be at the top of the stack. This eliminates the problem of executing verbs out of order which was present with the cache system. However, this is not an ideal system since if there are still verbs left on a process at the end of the execution of the expert system they will prevent the user from adding verbs to the system since the stack structure mandates that each verb be applied to the top of the stack.

The natural solution to this latest problem is to substitute a queue in place of the stack. This solves the issue of putting new verbs in front of old verbs in the order of execution without having to wait until the process is completely free of verbs. And like with using stacks there is no worry about verbs executing outside of the order desired by the user. Although strong, this solution is not perfect either. It still allows an old verb to clog the system preventing the processes from executing. But this is less of a hindrance to the system than simply eliminating verbs that do not get reduced or worrying about having verbs executing out of sequence and as was mentioned earlier verbs causing processes to go inactive is present in \( \pi \)-Calculus.

Another issue that was not clearly defined within the calculi is what happens to unrestricted variables once they have been sent on an outgoing channel. Obviously there are two possible things that can happen when an unrestricted variable is sent. The sending process either retains that variable in its data or it does not. When considering sending information along a channel it makes sense that the sending process still knows that piece of information and thus has it retained in its database. In contrast, when considering sending a communication channel that only one process at a time is allowed to use along another communication channel it would make sense that the sending process would not still posses the channel being sent. Since in \( \pi \)-Calculus there is no distinction between when a unique
A restriction signifies a variable in the system which has a restricted scope. The name field in the restriction fact is the name of the variable that is restricted. The list of processes for which the variable is restricted is the list of all the processes that have this variable restricted. All of the processes in this list must be in the same milieu since a restriction cannot span more than one milieu.

This also means that the execution of each reduction is only really a problem because the rules are reduction rules and axioms of structural congruence. The flow of execution is very natural as an expert system since its execution would not be randomized. It is not optimally efficient since it does not flow as naturally as it could. Since there is now no need to recreate Java versions of the rule-base is relatively straightforward. In each milieu, process, or restriction presented in the Java application there is a form of a toToString method. These methods print each entity as an assertion into the expert system. When building these entities the system travels to the sub-entities contained within the larger entity such as a verb and call the same type of toToString method on them. The higher level entity then uses the toToString results of its sub-entities to construct itself.

The entire system is pieced together from the toToString printouts of each type of entity into a vector. Each element of the vector is an individual entity. The running program then takes this vector and asserts each entity as an individual fact to the expert system. After all of these are asserted, the expert system runs the reductions. As it was mentioned earlier, randomizing this process eliminates the problem of the same fact always being chosen first in the execution of a rule.

3 The Implementation of the Expert System

Now that the issues have been discussed and solutions have been identified, this section deals with the detailed semantics of the presented expert system.

The first step of the flow of execution of the expert system, outputting the π-Calculus or Api-Calculus system into the rule-base is relatively straightforward. In each milieu, process, or restriction, as presented in the Java application there is a form of a toToString method. These methods print each entity as an assertion into the expert system. When building these entities the system travels to the sub-entities contained within the larger entity such as a verb and call the same type of toToString method on them. The higher level entity then uses the toToString results of its sub-entities to construct itself.

The entire system is pieced together from the toToString printouts of each type of entity into a vector. Each element of the vector is an individual entity. The running program then takes this vector and asserts each entity as an individual fact to the expert system. After all of these are asserted, the expert system runs the reductions. As it was mentioned earlier, randomizing this process eliminates the problem of the same fact always being chosen first in the execution of a rule.

3.1 The Main Entities

The expert system has three primary entities. These entities are the restriction, milieu, and process. The structure of these entities in the expert system is detailed below.

A restriction has the following form:

(restriction (name restricted-variable-name) (processes list-of-processes-restricted))

A restriction signifies a variable in the system which has a restricted scope. The name field in the restriction fact is the name of the variable that is restricted. The list of processes is the list of all the processes that have this variable restricted. All of the processes in this list must be in the same milieu since a restriction cannot span more than one milieu.
A milieu has the following form:

\[(\text{milieu} (\text{name} \ milieum-name) (\text{processes} \ list-of-processes) (\text{milieus} \ list-of-milieus) (\text{verbs} \ verb-list))\]

A milieu represents a conceptual grouping of processes and restrictions and is the integral addition to \[\pi\]-Calculus forming API-Calculus. The name field of the milieu represents the name of the milieu. The processes field contains a list of all the processes directly underneath this milieu. The milieu field contains a list of all the milieus directly underneath this milieu. The verb field contains a list of verbs that can be applied to the milieu. JoinMilieu and LeaveMilieu are the only two verbs which can be applied. They do not exist in the calculus because the calculus is designed to represent a snapshot of the system which will show the exact scope of the milieus. However, in our system there needs to be transitions so these verbs were created to signify when a milieu enters or leaves another milieu. A milieu needs to have at least one process or sub-milieu in it in order to exist. If a processes or sub-milieu leaves the milieu resulting in no processes and no sub-milieus then the milieu is destroyed.

One problem that needed addressing in this system was what happens to a process's restrictions when it leaves a milieu. Obviously the process cannot be under the exact same restrictions as before since a restriction cannot completely encompass a milieu. However, this does not address whether or not a process should join any restrictions in its new milieu that it was in under its own milieu. For instance consider the Api-Calculus system \[M_1/ [ xP | M_2/ [ x(\text{Leave}.Q \mid R) ] ]\]. It could either reduce to \[M_1/ [ x(P \mid Q) | M_2/[ x.R ] ] \text{ or } M_1/[ xP | Q | M_2/[ x.R ] ]\]. The former of these shows the process would join any restrictions in its new milieu that it was a member of in its previous while the latter shows that it would not join any new restrictions. The latter is the one the system operates under since although the user could certainly be try to represent the same term \(x\) being restricted there is nothing in the language that formally links the two. Also the former could be represented by using a supplementary verb **BindRestriction** which combined with **UnBindRestriction** adds the functionality of a process entering and leaving restrictions in the same way as milieus. Using **BindRestriction** the system would begin like \[M_1/[ xP \mid M_2/[ x(\text{Leave}.\text{BindRestriction} \ x.Q \mid R) ] ]\], then after one reduction would change to \[M_1/[ xP \mid M_2/[ x(\text{Leave}.\text{BindRestriction} \ x.Q \mid R) ] ]\] and after a second reduction would become \[M_1/[ x(P \mid Q) | M_2/[ x.R ] ]\]. With this added functionality, the system will not have processes joining restrictions that are named the same as the ones they belonged to inside their previous milieu when changing milieus.

A process has the following form:

\[(\text{process} (\text{name} \ process-name) (\text{verbs} \ verb-list) (\text{data} \ data-list))\]

A process represents an entity in the system with the capacity to store and transmit information to the other processes in the system as well as perform internal actions. The name field contains the name of the process. The verb list contains all the verbs yet to act on the process. These verbs are in sequential order starting with the leftmost. Any of the verbs which will be described later are legal on processes. The data field contains all of the data the process has. This data is unordered. The data contains regular data such as a variable with its value as well as communication channels with nothing differentiating them. There is no indication in the process's data field what data is restricted and what is not. This is indicated by the restriction entities.

### 3.2 Execution Stages and the Verbs

Overall the expert system is broken up into 3 stages of execution. The three stages are in order of highest to lowest priority are the reduction stage, selection stage, and collection stage. The selection stage is the primary stage and occurs as the middle of the three in priority. The selection stage is where verbs are chosen to be reduced. In the selection stage a verb is chosen from any process or milieu and is sent into the reduction stage where it is determined whether or not the verb is ready to be reduced. Once no more verbs remain or no possible reductions remain among the remaining verbs the expert system goes into the collection stage.

The collection stage which has the lowest priority of the three stages is simply where the expert system bundles its results to send back to the Java application after finishing a reduction. This stage gathers all the restrictions, milieus, and processes and binds them all into one large string separated by @'s.

By far the most complex stage is the reduction stage. This stage has the highest priority but it is not really one single stage but a collection of sequences grouped together. This stage is specific to the type of verb that was selected during the selection stage. Each verb has a specific method of execution as is detailed below.

The third argument of each verb in the system is the condition field. The condition field represents the match and mismatch actions from the calculi. No verb will be removed until its condition evaluates as true. The condition is of the form \(<\text{Condition} \ \text{sign} = \text{datum}1, \text{datum}2>\). In addition to representing match condition, the expert system has the added functionality over \(\pi\)-Calculus of mismatch. Mismatches are the same as matches except they use "!=" as opposed to "=".
Whether the condition is a match or a mismatch is signified by the sign. Sign is always a zero, referring to not equals, a one, referring to equals, or a three representing no condition. Datum1 and datum2 are pieces of data in the data list of the process the condition applies to. For example “0A,B” would translate to Java code as “A != B”, likewise “1A,B” would translate to “A == B”. In π–Calculus “InternalAction IA 0A,B” would be “[A ≠ B]” and “InternalAction IA 1A,B” would be “[A = B]”.

The Kill verb is the verb which terminates a process. It is of the form <Kill Kill condition>. Once this verb is evaluated the process terminates and is subsequently removed from all restrictions and milieus. This verb does not have a direct equivalent in the π–Calculus. However, it does give a method of representing 0. Like 0 it prevents the process from continuing execution. However, it does give some added functionality to 0. By transforming 0 from a process as it is in the calculi to a verb gives the user the ability to add a condition to it. This condition could be designed to always be false and thus leave the process in the system unlike in the calculi where all 0 processes are reduced out of the system.

The InternalAction verb has no perceptible affect on the system outside of the process it is applied to. It represents an action taken within the individual process and is of the form <InternalAction IA condition>. This verb operates identically to the τ prefix from π–Calculus.

The Replication verb acts identically to the “!” prefix in π–Calculus. It is of the form <Replication count condition>. The only field that is not straight forward is the count field. It represents how many children have been replicated so far by this process. The process replicating itself is unchanged by the replication process except for the count which increments by one. The process created is the same as the parent except it has the replication verb removed and its name is the same as the parent with a decimal followed by the count of the parent. For instance the first child of Process P would be Process P.0. Likewise P.0’s third child would be P.0.2. Once the child is created, it joins all the milieus and restrictions its parent belonged to.

The OpenMilieu verb is used to represent a milieu being destroyed. It is of the form <OpenMilieu OM condition>. When a milieu is opened all of its children are placed into the milieu’s parent. All of the restrictions that applied to the child processes of the milieu remain and if any restrictions exist with the same name the two restrictions continue existing side by side. These restrictions do not join together since they originally represented two separate terms albeit terms with the same name.

The JoinMilieu verb is used to represent either a process or milieu attempting to join a milieu. It is of the form <JoinMilieu milieu-name condition>. Before executing the verb on a milieu the system checks to verify that adding the milieu to the milieu it wishes to join would not cause it to become its own descendant. If it were to become its own descendant there could be a loop in the system which would cause problems in other verbs. This check is not computationally expensive since it only looks at the descendants of the milieu trying to join making the check takes only O(number of milieus). Of course before a milieu or process joins a milieu it needs to leave the milieu it is currently in. Once it has done that and in the case of processes exited all restrictions on its variables it is free to join the new milieu.

LeaveMilieu is the verb responsible for notifying the system of a process or milieu’s desire to leave the milieu it is currently in. It is of the form <LeaveMilieu LM condition>. It always takes the process or milieu out of its parent milieu and deposits it in the parent milieu of the milieu or process attempting to leave if one exists. So for a process or milieu to leave all of the milieus it is under it must have multiple LeaveMilieu verbs in a row. If this turns out to be a problem in the future a new verb could be added to make this process simpler. Unlike joining a milieu, leaving a milieu causes no risk of loops to the system.

The Receive verb represents the act of a process trying to receive a piece of data across a channel. It is of the form <Receive channel,variable condition>. To receive a variable a process must have the channel variable in its data list. Also it must have another process with a send verb that is trying to send along the same channel the receiving process is receiving on. Also for the transaction to work if the channel is restricted in the sending or receiving process, it also must be restricted in its counterpart. Once the system pairs up these two processes the system can execute the transaction. Upon receiving a variable, the process sets up a substitution replacing the value of the received variable in the data list or adding the variable to the data list. When the substitution executes, all of the names of the variable in the receiving process are changed to the names of the verb that was sent. This includes the data portion of the process along with any verbs that are set to run on that process.

The send verb represents the act of a process trying to send one of its pieces of data across a channel. It is of the form <Send channel,variable condition>. To send a variable a process must have the channel variable and the variable to send in its data list. The conditions needed to send match the conditions needed to receive which were detailed in the section on the Receive verb. Upon sending a variable the variable remains in the processes data list.

BindRestriction is the verb signifying binding of a variable in a process. It is of the form <BindRestriction variable-bound,value condition>. It signifies the action of a process joining a restriction. Although it is not an actual part of π–Calculus or Api-Calculus it does add to the functionality of the system as addressed earlier. If the user wants to use a “purer” form of the calculus the user simply would not use BindRestriction or its corollary UnBindRestriction.

UnBindRestriction is the verb signifying release of a variable in a process. It is of the form <UnBindRestriction variable-unbound condition>. UnBindRestriction is the antithesis of the BindRestriction, there is also no equivalent of UnBindRestriction in π–Calculus much like BindRestriction. However like BindRestriction, UnBindRestriction is a useful addition to the system since it signifies the opposite actions of BindRestriction. UnBindRestriction enables the system to be able to reduce the scope of an existing restriction or eliminate a
restriction. The combination of UnBindRestriction and BindRestriction give the system the capacity to make these changes in the scope of restrictions.

Using BindRestriction and UnBindRestriction in conjunctions gives the user another piece of added functionality to the calculi. The user is now able to introduce variables during the reduction of the system. The user simply needs to run a BindRestriction and specify the variable being brought in along with its value. This will bring the variable into the system and bind it in a restriction. If the user does not wish to have the variable in a restriction the BindRestriction can be followed immediately by an UnBindRestriction freeing it from the restriction.

4 Summary and Future Work

Although π−Calculus and Api-Calculus are formal modeling languages that are difficult to understand, the difficulty of this expert system is not in the understanding of these calculi. The real difficulty comes in the complication of defining it within the limitations of an expert system as can be noticed by requiring 80 rules to implement it.

Some of the difficulties encountered in the creation of this expert system certainly could have been solved another way. For example, having processes retain the information of the variables they send on outgoing channels was not explicitly defined in the definitions of the calculi. This decision and others like it were made trying to keep the best interest of the users of this expert system in mind and trying to make it an overall better product. Although these solutions may not be perfect answers, given the scope of this project, they were the best solution available.

Both π−Calculus and Api-Calculus are powerful tools that can be used to formally model systems. This paper presented an expert system that provides for these powerful languages to reduce complex systems automatically. This is currently a powerful tool for researchers interested in these formal modeling languages. However, there are multiple extensions of these calculi as different researchers adapt them to fit their needs. Naturally, there would need to be changes to this expert system to accommodate these different versions of the calculi.

An obvious direction for future work would be to attempt to implement the Summation Operator and Knowledge Units present in the calculi. Adding these elements would make expert system a more complete albeit more complex system. Also, this expert system was not created to act alone as an application. Its primary need rests in its interaction with applications being developed by various graduate students under Dr. Shahram Rahimi [8, 9, 10].

5 References


Qualitative Comparison of B, VDM and Z in Specifying Requirements of Safety Critical Systems

Ishrat Rahman Sami  
eShopWorks Ltd.  
39/43 Putney High St  
London, SW15 1SP, UK

Dr Brian Dupée  
School of Computing & Communications  
Southampton Solent University  
East Park Terrace  
Southampton, SO14 0YN, UK

Abstract Successful development of a safety critical system is difficult considering the high risks associated with it. Because of the enormous cost of safety, and the eliciting of all the appropriate safety requirements, it is essential to specify them correctly and verify the development of the system. The safety requirements of the system must take priority. Considering the extreme cost of safety and finding early injected errors for preventing failure formal methods of specification must be considered in the development process to assure quality of the developed system. B, VDM and Z are three popular formal specification languages in the industry.

This paper investigates the quality of the specifications developed in Z, VDM and B with respect to given quality criteria.

Keywords: Software Specification, Z, VDM, B

1 Introduction

Safety critical systems are sensitive because of their vulnerability to catastrophic accidents causing massive financial loss, harm to human life, environmental pollution and business viability. Researchers from NASA identified two major reasons for failure of safety critical systems: hardware failure and design flaws [1]. Design flaws can be dealt with through extensive testing, design density such as fault-tolerance, recovery blocks etc. and fault avoidance such as formal specification and verification, automatic program synthesis, etc. [2].

System specification is the reflection of user requirements used to implement a correct and safe system and to verify the developed implementation for dependability. Correct and complete specification thus has significant importance in minimizing design flaws through accurate representation of functional activities and structured representation of requirements necessary for easy verification, validation and reuse [3]. Poor specification causes system failure, even though the system performs according to it [4]. Moreover, specification errors are very costly if they are detected later during development [5, 6].

Considering the extreme cost of finding early injected errors, formal specification must be considered in the development process. B, VDM and Z are three popular formal specification languages. This paper will compare the performance experience of these formal specification languages in a case study and evaluate their strength with respect to identified criteria: ambiguity, correctness, consistency, completeness, dependability (availability, reliability, security and safety), reusability, traceability, modifiability, presence of over specification, duplicate specification and unrealistic specification following language independence during research on specification of a safety critical system.

Real-time computing systems can be specified as systems with explicit timing requirements on its functional behavior [7]. Therefore, to be acceptable, these systems must possess logical and timing correctness [8]. Safety Critical Systems are a special type of real-time systems where failure to meet critical timing and functional requirements will result in one or more of death, injury, damage and loss [7]: Railway Traffic Systems, Nuclear Power Plants, Chemical Plants, Medical Systems are well known examples. Therefore correct operational performance is the prime quality criterion, the nature of safety critical systems enforces correctness to be its major criterion [9] followed by dependability, the degree of user confidence in a system [5].

Fault, error and failure are inevitable products of software development. A safety critical system fails because of system faults and the introduction of unexpected behavior in the operational environment [8, 10]. Therefore, in order to minimize damage in the presence of failure, Fail Operational, Fail Soft, Fail Hard and Fail Safe conditions of the system and their remedies must be defined. Safety critical systems are inherently complex and therefore traceability and modifiability are also important quality characteristics [5]. These qualities also facilitate reusability.
2 Background

The main purpose of a specification document is to convey what the system should do to the later phases of software development in a structured format [5]. If the requirements are not organized in a structured way then finding inconsistency, conflicts and missing information will be difficult and that will result in an incorrect and incomplete specification. The abstract nature of specification is important as unnecessary and/or early design and implementation decisions may negatively affect the performance of the system and it is its abstract nature that makes it possible to verify the specification before implementation [11].

For a safety critical system’s specification it is crucial to consider all the necessary safety requirements. Formal specification languages provide guidelines and rules for formal specification development uniformly which are often supported by tools [12]. In the last two decades different model-based formal specification languages gained popularity for their effectiveness in developing complex dependable systems among which Abstract Machine Notation (AMN) Specification Language used by the B-Method, Z and VDM has had remarkable success [9]. The B-Method is based on weakest precondition and set theory where weakest precondition is used to prove the specification is correct and set theory is used as the major concept of specification; VDM and Z are also based on set theory and first-order logic and Z also considers schema calculus [13, 14].

The B-Method supports a layered development approach providing notation for rigorous requirements modeling, software interface specification, design, implementation and maintenance promoting incremental verification and validation of the produced portable software [3, 15]. It inherits the merits of its predecessor VDM and introduces preconditions, guarded commands, stepwise refinement, refinement calculus and data refinement and brought advancement in formal methods maintaining simplicity [16]. Increasing modularity through an object-based style, it aids simplicity and better understanding in developing complex systems reliably through extensive verification [13, 17]. B is supported through the entire development life-cycle by the B-Toolkit [15]. The B-Method adopted the Abstract Machine Notation as the formal language for development that uses predicate logic, set theory, language of generalized substitutions, sequences, relations, functions and other abstract data types [13, 18]. The Abstract Machine is its core building block which is a specification of a system or a partial system or a component of the system [16]. Composition of these defined Abstract Machines representing the desired behavior of partial systems gradually builds the whole system specification.

VDM provides a formal notation for expressing specification, an inference mechanism for constructing proof of correctness and a framework for developing specification in a verifiable manner and has become a British standard [19, 20]. VDM specification through step-wise refinement can be transformed into a more implementable version suitable for some target programming language [21]. Although concurrent processes cannot be expressed in VDM, it provides types and values to define data structures and functions and operations to define constrained functional requirements [20]. VDM specification can be done in either an analogous or modular fashion [21]. In this comparison, modular VDM is used.

Z is descriptive in nature and comprises informal text and formal description to explicitly specify task, data used and data input to and output from a function, pre-conditions and post-conditions for the function [19, 14]. In Z specification sequence is important as the definition becomes visible immediately after it is declared [20]. Z provides type definition, global constants definition and schema definition [22, 20, 14].

3 Case Study: Railway Crossing Control System & Criteria for Evaluation

Comparisons were performed on a case study of a Railway Crossing Control System [23] consisting of two subsystems: a Driving System (DS) that will be installed inside the train and a Crossing Control System (CCS) situated in crossing centers. For securing safe crossing these two systems must communicate with each other through a number of messages. This system was specified using each of the three specification languages.

3.1 Criteria for Evaluation

Ambiguity Domain knowledge and functionality of the system must be reflected unambiguously.

Correctness An incorrect specification will produce a wrong and unsuccessful system [19]. So, the necessity of a correct functional specification is essential.

Consistency Commonalities and conflicts among the requirements will introduce design flaws if not resolved during specification.

Completeness A complete specification promotes better design and a better system [19]. Completeness is a relative measure and it is extremely difficult to produce a complete product. Therefore, the evaluation is relative.

Availability To be dependable a system must be available when required [5]. Availability cannot be ensured by specification but a dependable system will have availability requirements fully specified.

Reliability Reliability of the system is related to the correct functionality of the system verified through testing [5]. A function needs to perform the right action at the right time. For a reliable system following aspects must be specified:

Interface Defining the interfaces for a safety-critical system is extremely challenging as the system's activity will
be decided upon the human-computer or computer-computer interaction and there is no room for operator error [24].

**Performance** Required performance of the system must be specified in some way as performance failure may lead to disaster [6].

**Timing Constraints** In hard real-time requirements it is necessary to specify when the task will take place and what the deadline of the task is to perform the correct task on time.

**Task Conditions** Some tasks are dependent on others and must be executed sequentially [19]. On the other hand some tasks are needed to be executed in parallel. It is possible that functional dependency may introduce deadlock [19]. So care is needed to have a deadlock free specification.

**Security** The security of safety critical system needed to be specified to save the system from intruders [5]. Security requirements are mostly functional requirements.

**Safety** Safety requirements are the major concern of every safety critical system.

**Environmental Conditions** An otherwise correct system developed in an ideal environment may not work properly in a particular operational environment [8]. Moreover, the system must take actions based on its current environmental conditions.

**Error Conditions and Remedies** Error conditions, and actions to be taken in the presence of error conditions, are needed to be investigated and recorded in the specification. If some conditions fail unexpectedly then the actions necessary to avoid failure must be specified for a safe system [10].

**Fail conditions, type and actions** Any system must start in a safe state and in case of abnormal conditions the system must end in a safe state i.e. in any abnormal situation the system must fail safe [6].

**Reusability, Traceability and Modifiability** This system is a part of a larger system and the specification may change (such as contents and/or sequence of transmitted data) with system enhancement.

**Duplicate, Unrealistic or Over-specification** A good specification is free from these [5].

**Language Independence** Although language independence of the specification language is not necessary for safety critical systems, it adds an extra benefit in international projects for resolving conflicts in the system [20].

4 Observations and Preliminary Analysis

Parallel specification of Railway Crossing Control System has been developed using B, VDM and Z. The Railway Crossing System is a safety critical system so it must be initialized in a safe state and in any abnormal situation to avoid catastrophic damage the system must stop in a fail safe state.

---

Module BarrierController

MACHINE BarrierController

SEES
TrafficLightController, CCS, GlobalData

INCLUDES
ErrorHandling

SETS
BARR_STATUS := {open, closed}

VARIABLES barrier

INVARIANT barrier : BARR_STATUS

INITIALISATION
barrier := closed

OPERATIONS
openBarr =
SELECT light=red & train_cond˜[true]={}
THEN barrier := open
ELSE ANY e WHERE e:ERROR
    THEN handleError(e) END
END;

closeBarr =
SELECT light=red & train_cond˜[true]/={} 
THEN barrier := close
ELSE ANY e WHERE e:ERROR
    THEN handleError(e) END
END;

failSafeBarr(a) =
PRE a:ALERT & a = on
THEN barrier := close END
END

Figure 1. Specification in B

4.1 Specification in B

The BarrierController module is used to control the barrier for the CCS. SEES and INCLUDES constructs define the dependency of this module to the TrafficLightController, CCS and others. This identifies the changes to the dependent modules For example: if the definition of train_cond changes in CCS then the specification of BarrierController must be modified. But seeing only this module does not provide the information that train_cond belongs to CCS not TrafficLightController or others. These features show B’s traceability. The module’s dependency on other modules is explicitly specified by USES and SEES. Such is important to reuse a module successfully and to remove conflicts regarding inter-module dependency to ensure consistency. The sequence of events can be explicitly specified in B through preconditions. Interfaces of modules and operations are well specified in the specification.

Preconditions of the operations enclose the environmental and operational conditions to decide activity. For specifying error conditions in openBarr and closeBarr functions nondeterministic SELECT and ANY statement are used to defer details until enough information becomes available. In this way, any occurrence of errors in these two functions must be resolved to avoid catastrophic danger. There-
fore, successful inclusion of environmental conditions, errors and associated actions, safe start state and fail safe state is enough to say that B provides mechanisms to specify a safe system considering safety measures.

B fails to record explicitly that after receiving a train status message, CCS must secure the crossing within a given time. This is a performance requirement essential for a valid system that depends on combined performance of several operations, but any particular instruction about when the task should start can be easily be incorporated in the specification document using a precondition. B also fails to specify any priority of the tasks. In an alert situation, the system should prioritize the failSafe operations to activate the red light and close the barrier. Security requirements of the systems are functional requirements that can be specified correctly in B. B inherits some notational ambiguity. For example, the symbol \( \lor \) stands for both set union and logical OR and the symbol \( \land \) stands for both sequence concatenation and logical AND. But this is not a major problem for the experienced developer.

### 4.2 Specification in VDM

VDM, like B, supports specifying a system in a modular fashion. VDM is also extremely rich considering built-in types. Definition of types can be defined or deferred according to available information represents the abstract nature of VDM avoiding over specification.

Inter-module dependencies are more efficiently specified in VDM than in B. A module can use other modules’ features through the ‘imports’ feature of VDM if the use is allowed by that module through the ‘exports’ feature. Therefore, a module can explicitly and consistently define its interface to other modules. This promotes better modifiability, traceability, verifiability and reusability. Moreover, inside the implicit operation’s definition it is explicitly specified which state variables will be accessed in which mode (read/write mode) and in case of using imported operations with the help of ‘using’ keyword. So, VDM more powerfully specifies dependencies increasing modifiability and reusability.

Start safe and fail safe concepts essential for safety critical systems are preserved efficiently in the specification. Environmental conditions required for a particular functionality is efficiently preserved explicitly or implicitly in functions and operations. The sequence of operations is extremely important for the safety of the system. Preconditions and post conditions ensure that the functions and operations transfer the system to a correct system state. When error conditions arise, and it is important to attend to them, is explicitly indicated in the specification. Therefore, efficiently incorporating environmental constraints, erroneous situations and failure conditions, VDM provides mechanisms to specify a safe system, but it fails to specify task’s priority.

### 4.3 Specification in Z

Z does not specify systems in a modular fashion. So, the two major subsystems are separately specified in Z. Formal specification is Z is assisted by natural language description. So, it is comparatively easy for a domain expert to understand the specification.

The meanings of the variables are unambiguously defined in further natural language descriptions. The schema FailSafeCenter specifies the requirements of fail safe mode successfully. Each operational schema encloses the environmental conditions which must be satisfied for performing that operation, post conditions that must be preserved in the environment after the operation and constraints that

```plaintext
module BarrierController
imports
  from GlobalData
types
    Train, Alert;
from ErrorHandling all;
from TrafficLightController
types
  LightStatus
operations
  LIGHT_COND: () -> LightStatus;
from CCS
operations
  TRAIN_COND: Train -> B
exports all
definitions
types
  BarrierStatus = CLOSED | OPEN
  state br of barrier: BarrierStatus
init mk_br(b) == b = CLOSED end
operations
  DARR_COND() b:BarrierStatus
  ext rd barrier: BarrierStatus
  post b = barrier;
  FAIL_SAFE_BR(a:Alert)
  ext wr barrier: BarrierStatus
  pre a = ON
  post barrier = CLOSED;
  OPEN_BR(t:Train)
  ext wr barrier: BarrierStatus
  pre (\((\text{LIGHT_COND()}\ \text{using}\ \text{light} = \text{RED})\ \\text{TRAIN_COND(t)} = \text{false}\))
  post barrier = OPEN
  errs b1:barrier /= OPEN -> (dc1 a:Actions-set = handleErr(b1));
  CLOSE_BR(t:Train)
  ext wr barrier:BarrierStatus
  pre (\((\text{LIGHT_COND()}\ \text{using}\ \text{light} = \text{RED})\ \\text{TRAIN_COND(t)} = \text{true}\))
  post barrier = CLOSE
  errs b2:barrier /= OPEN -> (dc1 a:Actions-set = handleErr(b2))
end BarrierController
```

Figure 2. Specification in VDM
must be maintained by the schema variables. OpenBarrier & CloseBarrier efficiently preserve environmental conditions critical for the safety of the system. By specifying environmental conditions, errors and remedies and start safe and fail safe modes, Z proved its ability to specify a safe system.

Dependencies among operations are well preserved promoting better modifiability, reusability and traceability through schema calculus. But inter-system dependency cannot be properly specified. Natural language description must then be incorporated. Sequential operational requirements of tasks are well preserved but Z fails to specify timing requirements within the schemas. The contents of the messages to be send and received are correctly and unambiguously specified using inputs and output features of Z. Interfaces are well specified through outputs and inputs of the schemas. The requirement that correct information must be always available and updated regularly is not incorporated in the formal part of the specification. As common

information shared by the example systems are specified in two places, duplicate specification exists which can introduce conflicts during changes in documents. Security requirements can be specified efficiently.

5 Comparison

It is clear from the research that formal specification provides unambiguous, correct and consistence specification with a better possibility of completeness. Functional requirements are correctly, consistently, unambiguously and relatively completely preserved in formal specification. As most of the security requirements are assumed to be functional so security requirements can also be efficiently incorporated in the specification document. All safety requirements can be well preserved which makes it attractive for safety critical system. Formal specification makes it more reusable, traceable and modifiable and thereby more consis-
tent, unambiguous, correct and complete. Although Z may suffer from over specification and there may be duplication in case of shared information, it is not a major problem and if care is taken during specification, consistency can be easily verified. B and VDM resist over specification, duplicate specification and unrealistic specification confidently. B, VDM and Z can be easily translated to different languages using look-up tables and automated tools. Among dependability requirements (availability, reliability, safety and security) all three formal languages fail to specify availability and reliability requirements completely. Formal specification can efficiently preserve most of the criteria. Therefore for excellent quality and safe systems, formal specification is necessary.

Although Z fails to preserve availability requirements formally, an informal description of Z can be used. B and VDM can express availability requirements but, for clarity, comments must be enclosed with the formal specification. In the case of reliability, some requirements can be incorporated in the specification efficiently and some cannot. It is clear that interface, dependency and sequence of operations can be efficiently preserved in formal specification but not timing constraints, performance and priority requirements. However, VDM, with specific extensions for expressing timing constraints like VDM++ and Z, incorporating RTIL (Real-Time Interval Logic) are now widely used for the development of safety critical systems [19]. Performance and priority requirements are also needed to be incorporated for the safety and success of the safety critical system. Formally, it is not possible as they depend on specific implementation. Informal natural language description or comments must assist formal specification to incorporate them.

6 Conclusion

It is clear that among the three formal specification languages Z contains core features for specifying functionally correct systems. Therefore, if there is a safety critical system that does not require timing correctness but requires only functional correctness, Z is sufficient. B and VDM are both very powerful specification languages; B has an object oriented nature while VDM follows structured programming style. The parallel specifications show that both VDM and the B-Method have sufficient and necessary features, the easier modular style of VDM wins, due to the transparency of imports.

References

Table 1. B, VDM and Z’s Ability to Specify Evaluation Criteria

<table>
<thead>
<tr>
<th>Criterion</th>
<th>B</th>
<th>VDM</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ambiguity</td>
<td>Common terms are defined explicitly promoting uniform use through the whole specification. Variables and constants can be unambiguously typed and constrained. Some notational ambiguity which may be well understood by developer’s experience.</td>
<td>Common terms are defined explicitly promoting uniform use through the whole specification. Variables and constants can be unambiguously typed and constrained. VDM has a rich set of built-in types providing correct and unambiguous specification of limits.</td>
<td>Common terms are defined explicitly promoting uniform use through the whole specification. Variables and constants can be unambiguously typed and constrained. Z’s built-in type is less rich than VDM but richer than B.</td>
</tr>
<tr>
<td>Correctness</td>
<td>All the functional requirements recorded in the requirements document are correctly specified. Functions, their operating conditions and their sequences are correctly preserved. All assumptions are explicitly preserved.</td>
<td>VDM specifies not only inter-module dependency but also the features of modules on which a it is dependent. Consistency in defining messages and their sequence is preserved.</td>
<td>Inter-system dependency is not preserved in the formal portion of Z. Consistency in defining messages and their sequence is preserved.</td>
</tr>
<tr>
<td>Consistency</td>
<td>Checking inter-module consistency is simpler in B than in Z. Consistency in defining messages and their sequence is preserved.</td>
<td>VDM provides features to specify more complete specification. Formal specification aids in finding incompleteness by picturing alternative situations through combinations of conditions.</td>
<td>Inter-system dependency is not preserved in the formal portion of Z. Consistency in defining messages and their sequence is preserved.</td>
</tr>
<tr>
<td>Completeness</td>
<td>B provides features to specify more complete specification. Formal specification aids in finding incompleteness by picturing alternative situations through combinations of conditions.</td>
<td>VDM provides features to specify more complete specification. Formal specification aids in finding incompleteness by picturing alternative situations through combinations of conditions.</td>
<td>Z provides a core subset of features to specify correct functional requirements. But Z is less powerful than B and VDM.</td>
</tr>
<tr>
<td>Availability</td>
<td>Considering the display example, needed for up-to-date information is preserved but refresh rate is not.</td>
<td>Considering the display example, needed for continuous refresh and recalculation is preserved but refresh rate is not.</td>
<td>Not preserved in the formal part of the specification but preserved in the natural language part.</td>
</tr>
<tr>
<td>Reliability</td>
<td>Some but not all timing requirements can be specified. Fails to specify performance and task priority requirements. Other reliability requirements can be incorporated efficiently.</td>
<td>VDM provides features to specify more complete specification. Formal specification aids in finding incompleteness by picturing alternative situations through combinations of conditions.</td>
<td>Formal part of Z fails to incorporate performance, timing and task priority requirements. Other reliability requirements can be incorporated efficiently.</td>
</tr>
<tr>
<td>Security</td>
<td>Security requirements can be well preserved.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Safety</td>
<td>Safety requirements can be efficiently specified.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reusability, Traceability &amp; Modifiability</td>
<td>B and VDM introduce a good reusable, traceable and modifiable specification.</td>
<td>Z is much less efficient in this sector but it also produces reusable, modifiable and traceable document.</td>
<td>Z is much less efficient in this sector but it also produces reusable, modifiable and traceable document.</td>
</tr>
<tr>
<td>Over specification &amp; duplicate specification</td>
<td>Over specification and duplication are efficiently avoided.</td>
<td>Over specification and duplication are efficiently avoided.</td>
<td>No duplication and over specification are found in a single document but large systems may have duplications due to the size of documentation.</td>
</tr>
<tr>
<td>Unrealistic specification</td>
<td>No evidence of unrealistic specification was found. But, as logic is the underlying tool for formal specification, incorporating unrealistic specification is extremely rare.</td>
<td>Over specification and duplication are efficiently avoided.</td>
<td>No duplication and over specification are found in a single document but large systems may have duplications due to the size of documentation.</td>
</tr>
<tr>
<td>Language independence</td>
<td>As formal specification is based on declared common terms, using a look-up table it can be easily translated to other languages.</td>
<td>Over specification and duplication are efficiently avoided.</td>
<td>Formal part of Z can be easily translated but the informal part cannot.</td>
</tr>
</tbody>
</table>
Queue Design and Implementation Based on Service Level of NQS

Young-Joo Lee, Chan-Yeol Park, Sung-Jun Kim, Jin-Woo Sung, Sang-Dong Lee, Joong-Kwon Kim

Korea Institute of Science and Technology Information, Daejeon, Korea

Abstract - A job management system is used in order to allocate a limited system resource to many users. A queue is designed and embodied to support various service levels. Users can select a suitable queue according to urgency or importance of their works. Based on the characteristics and configuration of the system and the patterns of user works, network queuing system (NQS) is selected as a job management system for NEC SX-5/6 systems at KISTI supercomputing center. By using this queue, users can select a suitable queue according to characteristics and purpose of their jobs, and system resources can be allocated efficiently. Our design and implementation of NQS queue guarantee the turnaround time based on service level and show the best performance at around 90 percentages of CPU usage.

Keywords: NQS, Job management.

1 Introduction

A job management system is a utility that allocates limited system resources to many users efficiently. Through the job management system, one can enable to define a resource policy and establish user interface for job execution management. There are several kinds of a job management system such as LoadLeveler, NQS or PBS. Because a job management system assigns whole system resources that each work requires, it is important to choose a job management system and to design the job execution queue.

Job management system can be marked out with characteristics of system and patterns of jobs. It is important in large system where lots of people are using, because a job management system influence entire efficiencies.

In this paper, we analyzed the property of NQS, the job management system, so we can design various queues by service level. And we have looked into the results of the operating queues.

NQS, originated from NASA, is a tool in order to schedule batch jobs. It is a system which uses ‘multi batch queue,’ limits resources by their properties, and this operates queues with batch job. We can define the various queues which can limit CPU Time, data, stack, resident set, file size and so on.

2 NQS queue design

2.1 System

The NEC system at KISTI Supercomputing center consists of two systems SX-5 and SX-6 that have a node and 2 nodes respectively. The NEC system is using the Gigabit network between nodes and the IXS(Internet Crossbar Switch) between SX-5 and SX-6. It uses the GFS file system and one can access all nodes through this GFS from one’s home directory.

<table>
<thead>
<tr>
<th>Table 1. Technical Summary of NEC system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content</td>
</tr>
<tr>
<td>Model</td>
</tr>
<tr>
<td>OS</td>
</tr>
<tr>
<td>CPUs/Node</td>
</tr>
<tr>
<td>Nodes</td>
</tr>
<tr>
<td>R_{peak}(GFlops)/CPU</td>
</tr>
<tr>
<td>Memory(GB) /Node</td>
</tr>
</tbody>
</table>

2.2 Queue Priorities

NQS allows users to submit batch jobs to queues for execution and provides facilities for remote queuing, queue status controls, batch request resource quota limits, and so on. How early one’s job runs compared to other jobs and how much one is charged are determined by the priority one use. We assigned different priorities to several queues, and the cost to use each queue is graded.
The charge is calculated from the charge factor which is related with a queue of NQS.

\[ \text{Charge} = \text{CPU Time} \times \text{Charge factor} \times \text{Machine factor} \]

Table 2. The charge factor

<table>
<thead>
<tr>
<th>Queue name</th>
<th>Charge factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Realtime</td>
<td>2</td>
</tr>
<tr>
<td>Express</td>
<td>1.5</td>
</tr>
<tr>
<td>normal</td>
<td></td>
</tr>
<tr>
<td>large</td>
<td>1</td>
</tr>
<tr>
<td>small</td>
<td></td>
</tr>
<tr>
<td>Economy</td>
<td>0.5</td>
</tr>
</tbody>
</table>

2.3 CPU Scheduling Algorithms

This system schedules processes using the CPU priority formula that includes variable parameters. Process priorities are recalculated at the time of followings:

- Each decay interval (except for processes running on the CPU) (1)
- At the return of an interrupt (2)
- At the time of a timeslice up (3)

\[
\text{(Execution priority)} = (\text{CPU counter})
\]
\[
> (\text{modification value}) + (\text{nice value}) + (\text{base priority}) + (\text{constant}),
\]
where \( (\text{CPU counter}) = (\text{CPU counter}) + (\text{tick quantum}) \)

or

\[
(\text{CPU counter}) = (\text{CPU counter}) >> (\text{decay factor})
\]

at the time of priority recovery (each decay interval)

Table 3. NQS Scheduling Parameter

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Explanation</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basepri</td>
<td>This variable is used by the system administrator or NQS administrator to adjust the execution priority</td>
<td>20</td>
</tr>
<tr>
<td>Modcpu</td>
<td>At the time of recalculation of the execution priority, the CPU counter is shifted with this</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 1 shows an example that 3 jobs (A, B and C) are assigned CPU resources and are executed via NQS, where the constant is 40, executing on CPU (for each tick; 1 tick = 1/HZ seconds; HZ = 200).

```
0 Second : start
1 Second later
A: (CPU counter) = 200
  (Execution priority) = 200 >> 2 + 20 + 40 = 110
B: (Execution priority) = 20 + 40 = 60
C: (Execution priority) = 20 + 40 = 60
2 Second later
A: (CPU counter) = 240 >> 120
  (Execution priority) = 120 >> 2 + 20 + 40 = 90
B: (Execution priority) = 20 + 40 = 60
C: (Execution priority) = 20 + 40 = 60
A: Sleep B: Run, C: Sleep
3 Second later
A: (CPU counter) = 120 >> 1 = 60
  (Execution priority) = 60 >> 2 + 20 + 40 = 75
B: (Execution priority) = 200
C: (Execution priority) = 20 + 40 = 60
4 Second later
A: (CPU counter) = 60 >> 1 = 30
  (Execution priority) = 30 >> 2 + 20 + 40 = 68
B: (CPU counter) = 240 >> 1 = 120
  (Execution priority) = 120 + 20 + 40 = 90
C: (Execution priority) = 20 + 40 = 60
```
Figure 1. Example of Process Dispatching

3 Queue design

3.1 Tests

We injected a set of sample jobs about three hundreds times on one node with real situations, and then we monitored and obtained the optimums of the priority, base priority, time slice, and aging range. Each job was executed by using various queues such as realtime, express, large, small and economy. We analyzed these properties and obtained suitable values for NEC system at KISTI supercomputing center.

Table 4. The parameter values

<table>
<thead>
<tr>
<th>Queue name</th>
<th>Charge factor</th>
<th>Priority</th>
<th>Base Priority</th>
<th>Time slice</th>
<th>Aging Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>realtime</td>
<td>2</td>
<td>30</td>
<td>60</td>
<td>2000</td>
<td>160</td>
</tr>
<tr>
<td>express</td>
<td>1.5</td>
<td>30</td>
<td>70</td>
<td>500</td>
<td>109</td>
</tr>
<tr>
<td>normal</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>large</td>
<td>1</td>
<td>29</td>
<td>85</td>
<td>1000</td>
<td>109</td>
</tr>
<tr>
<td>small</td>
<td>1</td>
<td>30</td>
<td>82</td>
<td>1000</td>
<td>109</td>
</tr>
<tr>
<td>economy</td>
<td>0.5</td>
<td>30</td>
<td>140</td>
<td>1000</td>
<td>160</td>
</tr>
</tbody>
</table>

In Table 4, Base Priority, Time slice, and Aging Range influence the performance of job processing. These parameters are directly influenced to CPU scheduling by increasing values and started to operate queue by queues. Base Priority is the most important parameter. It run the processing job and concerned when those process sleeps. Time slice is the constant time, in default value we set the realtime queue with doubled value. Aging Range is the point where we redefined the priority. To keep the Priority, realtime and economy didn’t change the default value. Figure 2 is obtained by using whole data.

The suitable values are summarized in Table 4. In Figure 2, large and small queues are normal. When they operated, they keep same level and same circumstances, and make small queue’s Priority higher than that of large queue so small queue can be operated faster. Because small queue’s operating time is shorter than large queue. When we design Queue, we thought that the memory is enough, so we didn’t define memory separately but just on small queue for faster job.

Figure 2. Test performance of queue

3.2 Node queue design

Users can submit batch queue from any node. The queue is distributed through load balancing pipe queue that distribute a job on a suitable node based CPU usage condition of each node. Each queue has a limit (Run, Group, User) of job in order that one user can not use the whole NEC systems. Figure 3 shows the procedure of a submitted job.

Figure 3. Example of Process Dispatching
In order to maintain the performance of Queue and uses a limited system resource efficiently, user should submit their jobs by using various queues. Jobs are using various queues are induced by using the upper limit of number of jobs for each queue and each user.

Table 5 is obtained by using NQS for one week. Figure 4 shows the performance of queue. This performance is related with Elapsed Time/CPU Time and CPU usage. This data comes from averages of NQS results obtained when CPU Usages are 50%, 60%, 70%, 80% and 90% respectively.

This figure can be reproduced in case that many jobs are treated by using NQS simultaneously.

Table 5. Statistic of NQS

<table>
<thead>
<tr>
<th>Queue Name</th>
<th>Jobs</th>
<th>Wait Time (sec)</th>
<th>Elapsed Time (sec)</th>
<th>CPU Time (sec)</th>
<th>Elapsed/CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>realtime</td>
<td>30</td>
<td>12</td>
<td>579,791</td>
<td>573,471</td>
<td>1.01</td>
</tr>
<tr>
<td>express</td>
<td>99</td>
<td>304</td>
<td>98,840</td>
<td>94,655</td>
<td>1.04</td>
</tr>
<tr>
<td>large</td>
<td>154</td>
<td>233,525</td>
<td>8,742,684</td>
<td>305,377</td>
<td>1.19</td>
</tr>
<tr>
<td>small</td>
<td>7</td>
<td>9</td>
<td>160</td>
<td>153</td>
<td>1.10</td>
</tr>
<tr>
<td>economy</td>
<td>14</td>
<td>5</td>
<td>1,450,125</td>
<td>58,005</td>
<td>2.50</td>
</tr>
</tbody>
</table>

Figure 4 shows an Idle Time rate as a function of CPU usage. The Idle Time rate represents the processing performance about a job and is calculated from an (Elapsed Time/CPU Time). In order to obtain the statistics, we had run the jobs more than six months. Single job had run mainly on the NEC system which used single node SMPs of vector multiprocessors. NQS distributes jobs very efficiently and the jobs compete to being allocated the computing resources after 75 percentage of CPU usage. As CPU usage increases, the completion becomes intense.

4 Conclusion

We executed jobs and monitored with various situations on NQS queue designed. The designed NQS queue guarantees the turnaround time based on service level. Therefore, users can select a suitable queue according to urgency or importance of their works, and they can execute their jobs promptly if necessary. The more competitive of CPU usage, our design showed the better performance. In order to get the best performance of NQS queue, users are recommended to use various queues.

In the future, the research will be conducted with another method to analyze the performance of job processes on the massive parallel machines.

5 References


SESSION

SOFTWARE MANAGEMENT

Chair(s)

TBA
Leveraging a Common Development Framework and Delivery Approaches through Effective Software Management

Gavin O’Brien
Robert Snelick
Lynne Rosenthal
Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg MD, USA

Abstract – The discipline of software engineering establishes a set of principles for designing and developing software. Organizations have applied these principles to deliver high-quality, cost-effective software products in a timely manner. However, one aspect of software development that is often overlooked is the management of groups of independent developers working on separate projects within an organization. This leads to groups using similar but equivalent technologies, creating duplicate support tools and infrastructures, and thus limiting the productivity of the organization. Using a common development framework and set of technologies reduces duplication, promotes reusability, and leverages staff expertise and knowledge. Software development becomes streamlined resulting in shorter development cycles while improving product flexibility. Moreover, it may be possible to easily and quickly adapt products to satisfy new customer demands. We describe how leveraging a common development framework and delivery approaches through effective software management optimized our resources and increased our productivity. The decisions and choices we made to establish our development framework are explained. Finally we demonstrate the effectiveness of our approach with a case study describing the development and delivery of our healthcare test tools.

Keywords: Development Environment; Project Management Issues; Project Management Tools; Software Engineering Methodologies; Software Reuse.

1 Introduction

When organizations set out to create software solutions, the development framework in which the work is conducted and the software technology used to develop the solutions are generally decided independently. In fact, in many organizations it is not unusual for the software technology to be explicitly chosen and the development framework to evolve as the project progresses. Nevertheless, the choice of development tools, technologies, support programs, etc. depend on factors related to the project such as: customer needs, customer requirements, legacy systems, project management and configuration control, developers’ expertise, and delivery approaches to name a few. If multiple software development groups exist within an organization, these choices are often made by each group based on the above factors. This can result in groups using similar but equivalent technologies, creating duplicate support tools and infrastructures, and potentially limiting the effectiveness of the groups and their products.

Using a common development framework and set of technologies can help reduce duplication, promote reusability, and leverage staff expertise and knowledge. Product development can become streamlined resulting in completing products sooner as well as providing more flexible products. Moreover, it may be possible to easily and quickly adapt products to satisfy new customer demands. For example, we were able to take an existing software tool, reuse the tool’s code generation and validation components, modify the user interface, change the delivery approach from a desktop application to a web application and produce a new, customized tool in less than one month.

Common development frameworks work well in organizations where there are multiple groups developing software independently of each other. For existing groups, step back and assess the technology and tools being used across the groups to determine where efforts can be combined or shared. For new groups, be aware of the technologies and development frameworks being used within the organization and let this guide the technology decisions. However, a common development framework should not be the primary mechanism for technology selection; rather it can be a differentiator when a choice must be made. A common development framework and technologies may not be appropriate for all organizations or projects. Reasons for why shared frameworks and technologies are not implemented include:

- The development timeline is usually compressed and once the technology decisions have been
made development groups tend to start developing rather than analyzing their decisions.

- Group fragmentation and lack of communication within organizations.
- The competitive nature of organization does not promote reuse and sharing.
- If a group is not maintaining a tool, they do not feel comfortable using it.
- The cost of many software packages is cheap and the maintenance and training costs can be underestimated.

This document describes how leveraging a common development framework and delivery approaches through effective software management optimized our resources and increased our productivity. It presents the decisions and choices we made in to establish our development framework as well as explaining the components that comprise the framework. The paper concludes with a case study of our healthcare test tools.

2 Background

The Software Diagnostics and Conformance Testing (SDCT) Division at the National Institute of Standards and Technology (NIST) is developing software tools, applications, and services to improve the quality and implementation of healthcare information systems. Specifically, these software products generate, validate and/or facilitate the exchange of messages and documents between healthcare systems.

Initially, our software products were being developed by several groups within the SDCT Division, independently of each other. Each product encompassed a wide range of technologies and required a different delivery approach (i.e., mechanisms for accessing and using the tools). The user interfaces of the products were different as was the underlying software for development. When we looked more closely at what each group was doing, we realized each group had its own methods for developing and sharing code, documentation, and issue or-to-do lists. A few groups maintained a wiki to share information and bug tracking was done in a variety of different ways.

We soon realized that we were targeting the same healthcare information systems and same customers, but were addressing different, although related aspects of the messages produced and exchanged by these healthcare systems. Moreover, we saw opportunities to increase our customer base if we enhanced our products, provided consistent user interfaces, and offered several delivery approaches. To optimize the usefulness of the tools and broaden their applicability, it made sense to combine them into an integrated testing toolkit. Although this required us to adjust and re-engineer some of our code and working habits, it provides synergy among our programmers and produces more flexible, usable software products. Thus, by analyzing the needs across these groups, we were able to optimize our resources and improve our productivity as well as to select the “best in breed” for our solution.

3 Development Framework

A development framework is a re-usable design for a software system (or subsystem). A development framework may include support programs, code libraries, a scripting language, software management tools or other software to develop the different components of a software project. [15]

SDCT’s development framework technologies were selected using a set of selection criteria described in section 3.1. These criteria were used as differentiators in determining our development environment described in section 3.2 and project management tools described in section 3.3.

When software developers make project based decisions, their focus tends to be on the project or problem at hand. Often times the organizational resources are not considered. The following observations were made as common pitfalls of the project focused developer versus the organizational focused developer.

Pick a set of technologies and tools and stick to it.

Research and development organizations are filled with curious and creative thinkers. These individuals want to work with the latest technologies. This tendency may create a variety of technologies that are difficult to maintain, may not be mature for prime-time use, and/or may not take into account legacy software.

Reuse of tools and applications whenever possible.

Server based tools and applications can be excellent candidates for this type of reuse. For example, a large group of developers working on different projects across an organization can share servers for: wikis, source code control, web servers, databases, libraries, and APIs.

Create an environment of knowledge sharing by keeping the organizations’ set of technologies small.

As the development framework becomes larger, the complexity of the framework grows. If there are no constraints on what is in the development framework, developers will continue to add tools and applications they like and are familiar with.

In contrast, keeping the development framework small forces all developers to learn and share the same set of
tools. This fosters an environment of reuse and avoids “reinventing the wheel”.

3.1 Software Selection Criteria

Whenever software packages are selected an important step is to create a list of criteria used for selecting the software. The software we are interested in relates to the software development process and must include common factors across the organization. The following is a list of our criteria:

Flexible and maintainable paradigm: The development technology must be flexible and maintainable to allow the development team to build a wide range of programs.

Open source software: Open source software is low cost, readily available, and is usually well documented.

Streamlined development process: The development process must not be burdensome to the developer. Turn around time cannot be hindered by a complicated development process.

Reusability: The development team shares libraries, graphics, files, directory structures and install packages

Optimization of resources: Every project does not host their own web server. Use one server for source code control, wiki, web, etc. Use the wiki for information sharing, document sharing, and bug tracking.

Cross training: With optimization of resources, cross training can be achieved by reviewing and studying documentation. Have developers proof read each other’s documentation to acquire knowledge.

Multiple delivery approaches: For maximum flexibility and software use, provide multiple ways for delivering software to an end-user, e.g. API, desktop application, web service, or web application.

Multiple operating systems, Windows/UNIX/Mac: All tools must run on all the various operating systems which we support and use.

3.2 Development Environment

Given the selection criteria listed in section 3.1 the SDCT healthcare team selected the following technologies for the development environment. Fortunately, all the groups within the SDCT Division were generally using object oriented technologies with some flavor of Java and every group was using XML. Specifically, the development environment consists of:

- Base Technologies
- Java, XML, and database technology
- J2EE
- XML, XSLT, XML Schema, Xpath, etc.
- MySQL
- Web Server
  - Apache (Tomcat/httpd)
  - Dynamic web content
  - JSP, Ajax, Axis2, HTML

3.3 Development Tools

Project management tools comprise the development framework, facilitating the development, tracking and management of software and resources. Although project management tools address a vast array of software development activities, we have selected a minimum set of tools that allowed us to deliver, control, document, test, and fix our software. Specifically, the areas for which we selected tools and the specific tools selected are:

Source Code Management is the management of multiple revisions of the same unit of information: CVS Server, Tortoise CVS (as PC client)

Integrated Development Environment (IDE) is a software application that provides comprehensive facilities to computer programmers for software development: Eclipse.

Coding Conventions are set of programming style guidelines that developers follow to help ensure that their source code is consistent, maintainable, and easy to read.

Bug Tracking is a system to document, assign, and track issues and problems with software: Trac

Documentation: wiki, javadoc, information sharing

Web Server: Apache, Tomcat

4 Delivery Approaches

It is often advantageous to provide users with alternative ways to access software products. In this section, we describe several different delivery approaches and present advantages and disadvantages of each. The approaches include application programming interfaces, desktop applications, web services, and web applications. The use of a web portal is also discussed as a way to effectively organize and publish the available services.

The delivery approach is influenced by the characteristics of the service and the requirements of the end user. In some cases, only a single delivery approach is appropriate. Yet, in other cases multiple approaches need to be considered. It may also be the case where more than one approach is suitable. We discuss the circumstances that lend one
Another consideration in choosing a desktop application is the delivery approach, which is an important consideration when selecting a deployment method. Desktop applications and browser applications see differences in how functionality can be delivered. For a discussion on when to use each type of application, it is crucial to understand their characteristics and advantages.

4.1 Application Programming Interfaces

A common method of providing services to users is through Application Programming Interfaces (APIs) [8]. APIs are a collection of related method calls that provide developer’s with basic building blocks to create applications. APIs are abstract notions that describe the interface and behavior; they do not describe how the interface is implemented. This is a powerful concept in that implementations may change but the user is given a consistent view of the set of services offered by the API. Often APIs become standardized beyond a single organization and when implemented by vendors enables interoperability between vendor’s products. APIs are heavily used in information technology projects today. Well-known APIs include the Java development environment [9] for general purpose program development and Microsoft’s DirectX for handling tasks related to multimedia and game programming [10].

4.2 Desktop Applications

Desktop applications are simply defined as programs that run on your personal computer or laptop. Historically, desktop applications have been the predominate choice of developer’s and vendor’s for delivering products. Common desktop applications include word processing, e-mail clients, and games. Desktop applications offer rich graphical user-interfaces (GUI) and often take full advantage of the system resources offered by the desktop system. In contrast to web-based applications, desktop applications are not as susceptible to network problems or browser failures. Since desktop applications reside on a user’s machine, they are readily accessible and are a good choice when there is a need to tightly integrate functionality within the desktop, e.g., drag-and-drop features. Desktops and desktop applications inherently offer better security which is an important consideration when selecting a delivery approach.

Another consideration in choosing a desktop application is the cost of deployment and maintenance. Desktop applications have to be installed on every computer system that will use it. When new versions or updates are available, the application has to be reinstalled or patched on every computer. This can be labor intensive and costly, especially in applications that require frequent updates. Moreover, the timeframe in which the updates are completed are unpredictable and there is no guarantee that they are completed. In contrast, web applications can be maintained and updated locally on the server and offer an alternative to desktop applications when the same level of functionality can be delivered. For a discussion on when to deploy a desktop applications and browser applications see [11].

4.3 Web Services

A growing trend is the proliferation of delivering application services via web services. A web service provides APIs to users through access over a network, such as the Internet [12]. The user invokes a method which is executed on the remote system hosting the requesting service. Web services share business logic, data, and processes through the API but unlike web applications do not provide a GUI or require a browser. The APIs provided can be integrated into user’s applications or the information retrieved can be directly rendered. Examples of web services include storage management, customer relationship management, stock quotes, and checking of bids for auction items [13]. Characteristics and advantages of web services include being self-describing, distributed, loosely coupled, portable, and self-contained units of functionality. In addition, they are language and platform independent and are published, located, and accessed over a network, such as the Internet.

Web services allow for remote access to application functionality and data across networks. The technology relies on a related set of Internet protocol standards that include XML, SOAP, HTTP, WSDL, ebXML, and UDDI. XML provides a mean for standardizing data formats and exchanging data. The SOAP (Simple Object Access Protocol) specification is used to transfer the data via XML messages. WSDL (Web Services Description Language) describes the available web service operations. WSDL is based on XML and is machine readable. Platform independent XML-based registries such as ebXML and UDDI (Universal Description, Discovery, and Integration), can be used to list the available services.

One important use of web services is that it enables organizations with legacy applications to easily integrate with other systems in a standard way. For example, an organization may have its own unique system of managing information (such as inventory data) and they need to expose the application logic to a heterogeneous client-base. Since web services are standard-based clients running on disparate systems can programmatically access application functionality exposed using web service technology, without regard to the implementation details of the service or the runtime environment [14]. In our own experience, we provide a healthcare message validation web service in which users with testing environments implemented on different development platforms utilize the services seamlessly. We elaborate on this topic in the case study presented in section 5. Another case where this proves useful is bridging .net and Java applications or UNIX and Window applications.

4.4 Web Portal

A web portal is a site that functions as a point of access to information on the World Wide Web. Portals present...
information from diverse sources in a unified way. Aside from the search engine standard, web portals offer other services such as e-mail, news, stock prices, and various other features. Portals provide a way for enterprises to provide a consistent look and feel with access control and procedures for multiple applications, which otherwise would have been different entities altogether. At NIST, we use a web portal as a means to deliver a wide range of healthcare testing products to our users. It is one stop shopping for obtaining all NIST’s healthcare tools, applications, and information.

5 Case Study: Healthcare Test Tools

NIST is developing testing tools to support conformance and interoperability of healthcare messaging and document standards. These tools include services for message and document validation, message generation, reference implementations, and communication utilities. The tools are open-source and are placed in the public domain.

Early on in the project it became apparent that the organizational resources and the requirements of the intended users were very diverse in the healthcare IT community. Some sought simple utilities to validate message instances, some needed the utilities to be local because of privacy concerns of healthcare data, yet others had the resources to incorporate the services into their existing products, while still others needed to integrate the services into a testing platform that was built using a different development platform. Thus, it was imperative that we designed a flexible and efficient architecture that could make the service accessible and support the requirements of the end users. In developing our toolkit, we employed the principles and utilities described earlier in this paper. This allowed us to quickly and efficiently develop and deploy the services in numerous ways.

Figure 1 illustrates the overall design and services of our HL7 Version 2 testing toolkit. At the center of the toolkit is a collection of core services that are implemented as APIs. The APIs and their implementation provide the foundation upon which the other applications and utilities are built. This design promotes reuse and contains the core functionality of the toolkit to the definitions of the APIs.

Figure 1. HL7 Testing Toolkit: Design and Services

The APIs are written in Java and provide the building blocks to create desktop applications, web services, and web applications. Using the project management and software engineering techniques described earlier, we developed a number of desktop applications. Each application has the same look-and-feel and reuses many graphical components. A cornerstone application of the toolkit that utilizes the message generation API is HL7 Message Maker [5, 6].

We have also developed web services that expose the message validation and message generation interfaces [13]. In one case, a developer used the web service to bridge the DICOM validation toolkit (DVTk) [7] to our HL7 toolkit. This enabled expanded interoperability testing of healthcare testing scenarios as described by the Integrating the Healthcare Enterprise (IHE) initiative [3]. DICOM is a healthcare message imaging standard and HL7 is a healthcare message clinical data standard. The DVTk environment is written in the .net development platform in C++. Using web services was a quick and efficient way to bridge the diverse platforms (i.e., .net/C++ and J2EE/Java) to create a testing environment capable of evaluating scenarios involving DICOM and HL7 messages.

Web applications have also proven to be an effective delivery approach for providing validation services for conformance testing. For example, the message validation web application [1] allows users to validate message instances against a given conformance profile. The user can select or upload a conformance profile and upload a message instance to the web site and immediately view a validation report. Very little requirements are placed on the user. The message validation web application has preloaded conformance profiles that are used by test scenarios at interoperability connect-a-thons [3, 4]. This allows vendors to pretest messages produced by their applications prior to the connect-a-thon testing.
Users can also make use of the APIs by developing their own applications (e.g., a message generation utility) or by incorporating the services into their testing environments or applications. This further demonstrates the flexibility and utility of the design approach.

Another set of tools developed at NIST focuses on the Clinical Document Architecture (CDA) used in HL7 Version 3. This tool, the NIST CDA Validator, determines if an XML instance document is correct with respect to the specifications identified within the CDA/CCD-based specification and with respect to HITSP/IHE constructs.

The NIST Validator uses schemas and schematron rules to test the documents with respect to the CDA R2 XML Schema, CCD XML Schematron rules, and IHE XML schematron rules. The tool is a web based application using the NIST framework user-interface and web service infrastructure. The developers of this tool were different from the messaging toolkit developers, yet they were able to leverage the framework to build a tool with the same look and feel in only four months.

The HIT-Testing [2] web site is an example of a portal for delivering a wide range of information and test tools. This web site, developed in partnership with the Office of the National Coordinator for Health Information Technology (ONC), Healthcare Information Technology Standards Panel (HITSP), and the Certification Commission for Healthcare Information Technology (CCHIT) provides HIT implementers with access to the tools and resources needed to support and test their implementation of standards-based health systems. Ultimately users can obtain test tools developed using the wide range of delivery approaches explained above. The development of this site was also leveraged under the framework and principles we described earlier.

7 References


6 Summary

NIST has leveraged a common development framework and delivery approaches to optimize development time, development resources, and maintain a common look and feel throughout their tools and applications. In addition, the common development framework creates synergies across the organization helping developers to communicate and optimize their time and efforts.

Understanding software development organizations can have a profound effect on their efficiency. NIST realized that one of the strengths of software developers is their curiosity and thirst for new technologies, but they also recognized the pitfalls in always transitioning to something new from a product point of view.
Towards an Information Driven Software Development Life Cycle

Dr. Ernest Cachia, Mr. Mark Micallef
Department of Computer Science, Faculty of ICT, University of Malta, Msida, Malta

Abstract - Abstract--- Although software engineering has matured greatly over the years, a large number of ICT projects continue to fail[1][2]. Studies continue to identify non-technical issues such as poor communication, shifting requirements and poor executive involvement as the main causes of these failures. This paper identifies such well known causes and poses the question as to why currently available software development life cycles fall short of dealing with them. Drawing on results from a research exercise carried out by the authors, a link is made between the quality of information used throughout the development life cycle and the quality of the resultant product. Consequently, it is proposed that organisations knowingly or unknowingly maintain a knowledge context and the quality of this knowledge context has direct impact on product quality. Furthermore, it is proposed that a software development life cycle be developed in which participants do not focus explicitly on the traditional phases of software development. Rather, a conscious decision is made to focus instead on possibility of developing an altogether different way of thinking by which high quality systems could be engineered within budget and on time.

Keywords: Quality Assurance, Software Development Life Cycles, Software Engineering

1 Introduction

It could be said that the research area of development life cycles is indeed mature. Since the early days of software engineering, this area has seen the development of a number of models and methodologies ranging from the generic waterfall model [3] to the more recent agile techniques [4][5]. Different approaches function to varying degrees of success depending on the scenario at hand. However, given that ICT projects persistently continue to be late and even of insufficient quality [1], one is compelled to consider the possibility that the software engineering community may have taken a wrong turn at some point. One must explore the possibility of developing an altogether different way of thinking by which high quality systems could be engineered.

Traditionally, software development life cycle has been perceived as a structured process imposed on the development of a product. In so doing, the development process focuses explicitly on the product thus putting it through a number of phases before finally delivering it in its finished form. At its core, a particular life cycle differs from others in the way it guides a product through transitions between these different phases. Throughout this paper, such life cycles will be referred to as product-oriented life cycles. Due to the fact that the primary goal is usually that of delivering a product, the thinking behind product-oriented life cycles inherently seems to make sense. However, perceiving software engineering as simply being all about the product may be misleading. Software is after all, an intangible artifact conceived entirely from knowledge and at its core, exists solely to facilitate the use of information and knowledge. Furthermore, the nature of modern software engineering environments gives rise to a whole new genre of problems which directly or indirectly affect product quality and project timeliness. Due to issues such as high expectations of software, constraining time lines, increased staff turnover, engineers' intra-project mobility and the dynamic nature of all information related to a product, problems such as cognitive overload, information anxiety and duplication of effort amongst others have been observed. These problems are discussed further in section 2 but are being mentioned here to highlight a problem which is not explicitly dealt with by product-oriented development life cycles.

In this paper, it is being proposed and hypothesized that every organization, knowingly or unknowingly maintains a knowledge context. We define this knowledge context as being the knowledge, technical or otherwise, held by any of the organization's stakeholders at a particular point in time. It is being proposed that the quality of a product is directly related to the quality of the knowledge context used to create it. Consequently, this knowledge context should be nurtured and maintained so as to ensure the timely delivery of high quality products. Finally, it is being proposed that a new type of software development life cycle be developed whereby the focal point is the development and maintenance of a high quality knowledge context. If the proposals put forward here are true, it is felt that a high quality product will naturally follow.
2 Modern Software Engineering Environments

This section describes typical characteristics of the modern work place which give rise to problems effecting project timeliness and/or product quality. The problems described here tend to go beyond the problem domains handled by traditional development life cycles and serve to illustrate the benefits of maintaining a high quality knowledge context within an organisation.

Over the years, expectations of ICT systems have gone from storage and retrieval of data to complex functionality which automates and complements business processes in an attempt to gain a competitive edge. Due to market pressures, this increasing functionality is being demanded in shorter spans of time [6]. Whereas in the past it may have been common to have software development projects go on for over a year, today delivery dates of between four to twelve weeks are more common place [6]. Compounding this increased complexity and time restrictions, modern systems are also highly susceptible to an onslaught of external factors manifested in the form of changing requirements, conflicting decisions, changing directions, experimental technologies, and so on. In essence, the software engineering process no longer exists in a convenient bubble which enables engineers to ignore an evolving world whilst engineering a product which caters for a freeze-frame of that dynamic world.

Putting technical merits aside for the time being, this constant onslaught of new or changing information in a diversity of formats from across the spectrum of quality has lead to the observation of cognitive overload in the work place [7]. A study amongst Fortune 1000 workers indicates that workers now work in environments of increased complexity, saturated with multi-tasking, interruption, and profound information overload [8]. A number of studies claim that consequences of such environments include information anxiety, social tension, job dissatisfaction, ill health, increased staff turnover, and consequently poor quality of work [8].

Another characteristic of the software engineering environment resulting from all this is the increase in inter-project mobility. An engineer can expect to be shifted between projects on a regular basis depending on a number of factors such as customer priorities, project schedules, funding and so on. When an engineer switches projects in this way, she needs time to adjust to the new context. This may involve familiarising herself with the project, technologies being used, design architectures being utilised, decisions which were taken, and so on. During this adjustment period, the engineer may also distract other employees from their work because of her need to ask questions and understand project-specific issues. All being said, one realises that there is a certain amount of time after a switch during which the engineer is minimally productive at best or counter productive to the team's efforts at worst. Given the shortening project schedules, this is not a desirable situation.

A somewhat related concept refers to staff turnover, a recurring concern with ICT companies where annual turnover rates can rise above 10% [9]. With this regular flow of staff leaving and new staff joining, one's challenge is two-fold. Firstly, one must somehow retain the knowledge held by departing staff for use in current and future projects. Secondly, one needs a strategy for transferring all required knowledge to new staff as quickly and effectively as possible so as to enable them to be productive.

Finally, we examine a situation stemming from the independant way in which teams within the same company seem to operate. It is not uncommon for a development team to spend a considerable amount of time (typically days) solving a problem with (for example) a third-party component only to realise a few weeks later that the same problem had already been solved by another team in the same company. This discovery would understandably result in frustration on the engineers' side for having wasted time reinventing the wheel, as well as on the management's side due to the waste in time and money that unnecessary duplication of efforts causes.

With all this information and knowledge being created, modified, used, and retired on a daily basis, one needs to develop ways to effectively manage this information and focus it towards achieving the goals at hand. It is the opinion of the authors of this paper that the formalisation of the concept of a knowledge context would be a concrete first step in dealing with these situations. Take the example whereby engineers are likely to be shifted around projects regularly. In this situation, the organisation in question would do well to somehow ensure that all engineers had a certain minimal knowledge about most (if not all) ongoing projects in the company. If this was achieved, switching engineers between projects would be smoother. Similarly, the concept of duplication of work would be virtually eliminated if an engineer could be notified that the problem which he is currently working on has already been solved and was somehow pointed to the solution.

When considering the whole concept of knowledge context and how it may be used, one is undoubtedly inundated with questions about how a number of issues would be solved. For example, in the case of duplicated effort, one must certainly be aware of the difficulties inherent in keeping everyone informed about everything all the time. This would surely only compound the problem of cognitive overload. At this point, the scope is to put forward the concept of the knowledge context and the benefits which its formalisation would bring. It is beyond the immediate scope to delve into the details of how to actually build, maintain and use such a context.
3 The Knowledge Context

In section 1, a knowledge context is defined as being the knowledge, technical or otherwise, held by any of the organisation's stakeholders at a particular point in time. This definition, although concise, illustrates the importance of three particular issues. Firstly, it puts forward the concept that all relevant knowledge, be it technical or not, is important to a project's success. That is to say that although sound technical knowledge (specifications, design, programming language knowledge, etc) is essential when delivering quality software, non-technical information is just as essential. Examples of non-technical knowledge include things such as the client's future business aspirations, legislation relating to the product being developed, staff vacation plans and so on. Secondly, the definition makes reference to all stakeholders of the company. This is important because communication problems have been shown to considerably influence the success of a project [10]. Therefore, there needs to be a constant flow of relevant information between all levels of the organisation's hierarchy as well as any external stakeholders. Finally, the definition makes reference to the temporal aspect of knowledge and information. Different information may be required by the same person at different points in time. The temporal information requirements may be as obvious as the engineer needing specifications during the design phase and needing design documents during the development phase. However, it is often the case that one may need access to the same knowledge albeit it from a different perspective or maybe using information with different characteristics (finer granularity, different media, etc). People will accumulate a certain level of knowledge over time and placing the right information in the right peoples' hands at the right time will facilitate better product quality in all its aspects.

At this point, it is useful to explicitly distinguish between knowledge and information. This is necessary because these two terms are sometimes used interchangeably and the difference between the two is key to the concepts presented in this paper. Knowledge refers to one's acquaintance with facts, principles, concepts, theories and so on. Information on the other hand, refers to the transfer of knowledge in some way, shape or form.

4 Knowledge used in Software Engineering

In order to delve deeper into the abstract concept of a knowledge context, the authors of this paper carried out a research exercise with the participation of development professionals, management professionals and entrepreneurs who have had experience commissioning ICT systems. The scope of this exercise was to identify information which is used throughout software development, classify it into a number of manageable knowledge areas and discuss the impact which the quality of this information would have on a finished product.
After further discussion and analysis, it resulted that these knowledge items could each be placed in one of three categories. The first category is the Technical Knowledge category. This refers to knowledge which is related to the technical aspect of building a software product. Examples from this category include product requirements, architectural designs, test plans, metrics readings, and solutions to past technical problems. Thirteen (41%) of the items identified fell into this category.

The second category is the Resource Knowledge category and refers to knowledge related to the resources required to carry out a project. This includes knowledge such as staff training needs, staff project allocation, staff vacation plans, hardware availability, staff tendency to be sick, and so on. Ten (31%) of the items identified were classified as being in this category.

Finally, a third category emerged and was named the Constraining Knowledge category. As the name suggests, knowledge in this category would lead to stakeholders having to make decisions and take actions within certain boundaries, even if this sometimes means going against sound technical principles. Some examples of knowledge in this category include company goals and policies, decisions, time lines, market status, legislation and project budgets. Nine (28%) of the identified items were deemed to be in this category.

5 Information and product quality

One of the original goals of the research leading up to this paper was that of establishing a link between the quality of information used throughout product development and the quality of the resulting product. Results from the research exercise discussed in section 4.1 indicate that this is indeed the case. At this point, our research is only concerned with linking information quality to product quality. Although a future research goal would involve quantifying what aspects of product quality are influenced by particular aspects of information quality, this is not yet within our scope. As such, instead of analysing each individual knowledge item and the information associated with it, it suffices to analyse the three knowledge categories identified in section 4.2. This section categorised all knowledge information as being technical, resource-related or constraining. Each of these categories is examined in turn below.

Participants in our research exercise claimed the quality of the technical information used throughout a development process was paramount to the resulting solution. It may sound obvious that, for example, creating code based on a design which was in turn based on conflicting and inaccurate specifications will result in a product of questionable quality. However, participants highlighted a number of interesting situations which may not seem so obvious. One such example involved a team encountering a problem with a third-party library used to develop a product. This problem was a show-stopper and took three days to solve. Considering that the team was working within a twenty day iteration, this resulted in the loss of 15% of the total iteration time. During a postmortem meeting, it was frustrating for the team to discover that one of the other teams said they had encountered and solved the same problem in a previous project. Had there been adequate knowledge transfer between teams, the 15% of iteration time spent fixing the problem would have instead gone towards adding more functionality and/or improving overall quality.

With regards to resource-related information, opinions initially varied as to the actual impact this had on product quality. Beyond staff-project allocation, participants seemed to be used to a fire-fighting approach when it came to resources. If someone took some unplanned days off or was out sick, the other team members would cover for him or the person involved would work late nights to make the deadlines upon returning to work. The same approach seems to be applied to hardware availability. If for example an important test server develops a fault, participants claimed they simply do the best with whatever resources were left until the server was fixed. These arguments seem to indicate that human resourcefulness and sheer effort makes the need for high-quality resource-related information unnecessary. However, further discussion revealed otherwise. It transpired that in the case of the sick engineer who worked late nights in order to make up for lost time, the resulting module for which that engineer was responsible for a large number of problems discovered by the testing team. Similarly, in the case of a test server failing, this sometimes resulted in a product release being delayed or products being released without adequate testing. Eventually, participants agreed that having high-quality resource-related information at hand would facilitate better project planning which in turn would have a positive impact on product quality.
Finally, issues related to constraining information are analysed. In this regard, participants acknowledged that not having the right information at hand in this area would affect product quality although there seemed to be a certain aura of helplessness in the discussion and scenarios put forward. One participant complained that he had worked for a company which kept changing the priority of projects which were worked upon. Consequently, she was forced to switch between projects on a very regular basis. Project priorities are a result of company goals and company policies, both of which were identified as being types of constraining information. This is because even though on a technical or project management level, it makes more sense to finish an item of work before moving on to the next, if project priorities change you may be constricted to do otherwise. Another participant described a scenario where a product had to be considerably restructured because of a change in financial legislation. It turns out that this change in legislation had been announced more than a year before it actually came into effect. Had this knowledge been available to engineers, the product would have been done right the first time round.

From the research exercise carried out, it is clear that the presence or absence of required information with the required level of quality will impact the quality of the finished product. Hence a development process should ensure that all stakeholders have the all the information, with the right characteristics (quantity, representation, and so on) at the right time. The following section identifies a number of challenges involved when maintaining a knowledge context in this regard.

### 6 Challenges involved when maintaining a Knowledge Context

Having shown the need for development processes to maintain a knowledge context within an organisation, it is worth exploring what challenges one is likely to face when attempting this. Seven key challenges where identified and are discussed in this section. Given that systems grow increasingly larger in terms of the functionality they offer, the amount of information associated with such systems is also bound to grow. These circumstances, along with the temporal properties of information which were discussed in section 2, leads to the natural conclusion that electronic tool support would be needed when it comes to maintaining a knowledge context. This immediately gives rise to the challenges of how one would capture and manage increasing amounts of dynamic information relating to a project and the organisation as a whole. Typically, a chunk of information would need to be captured, associated or related to other information and somehow tagged with attributes so that it can be easily accessed in future.

Even if one develops a way of capturing and managing information, there is still a matter of quality which needs to be addressed. Before one allows a chunk of information to somehow influence the development of a product, one needs to be sufficiently sure of its quality. Lee et al [11] identified fourteen attributes relating to the quality of information. The third challenge surfaces here. How does one evaluate the quality of information in a reliable manner without being overly intrusive? Although knowing the quality of information is important, one must strike a balance whereby information quality can ascertained with a reasonable degree of certainty without being counter productive to development effort. On a related note, it would be desirable to quantitatively link information quality to product quality. That is to say, by evaluating the quality of an organisation's knowledge context one would be able to reason about, or even measure the quality of a product which is being developed at a particular point in time. The establishment of such a link would enable an organisation to take corrective measures from a knowledge perspective should the product quality not be desirable.

Finally we identify three challenges related to the temporal and dynamic quality of information. Over time, certain events will occur which will result in one or more people needing particular information. Such events may include a particular milestone being reached, a change in requirements, a change in relevant legislation, someone leaving the company, and so on. The challenges here refer to knowing when a particular information asset is needed, knowing who needs it and knowing what characteristics it needs to exhibit. The latter requirement is important because the same body of knowledge may be represented in different ways. Furthermore, the chosen representation has the potential to positively or negatively influence the effective use of that knowledge. Representations of a body of knowledge may differ in a number of ways such as format, level of abstraction, type, and so on.

### 7 Product Focused Models

The reasoning behind product focused development life cycles is indeed logical and, at face value, completely correct. In such models, the emphasis is on building a product of a certain level of quality, usually within a stipulated time frame. Typically, a product would go through a number of phases (specification, design, development, testing, etc) before finally being delivered. The main difference between different mainstream development life cycles is the way the product transitions between these phases. In fact, existing life cycles are classified into four groups: sequential, incremental, evolutionary, and agile. The naming of these classifications illustrates the way in which a product will be built if it were to be developed using a life cycle in a particular group. Initially, this makes perfect sense. A software development team is in fact meant to develop software products. Hence it should follow that such teams follow a process which is focused on delivering products. However, having identified the need for a knowledge context to be maintained by an organisation, how well do existing development life cycles actually cope with the challenges identified in section 6. Having conducted research into a number of development life cycles, the authors
of this paper conclude that these challenges are not comprehensively addressed by mainstream models. Even though more recent life cycles such as extreme programming (XP) [4], Scrum [5] and DSDM [12] cushion the effect changing information by introducing iterations or sprints, they still do not address most of the challenges identified in section 6. Of all existing life cycles, XP comes closest to achieving what we are looking for. It acknowledges the existence of institutional knowledge and promotes communication in feedback so as to facilitate its dissemination among members of a development team [4]. However, the development team by no means constitutes all stakeholders of an organisation. Also, although the constant communication and feedback loop will likely have a positive effect on maintaining a knowledge context, it does not protect stakeholders against cognitive overload and other pitfalls identified in section 2. That being said, one is compelled to explore the possibility of there being a better way to handle the challenges presented when maintaining an information context.

8 The Information-Driven Approach

It is being proposed that an information driven software development life cycle be developed. This life cycle should effectively tackle the challenges identified in section 6 and produce a high-quality knowledge context as well as a high quality finished product. Broadly speaking, the life cycle should provide capabilities in two areas: Knowledge capture and evaluation and Knowledge utilisation. The knowledge capture and evaluation capabilities involve capturing real-world knowledge, relating it to other knowledge, evaluating its quality and storing it for later use. This aspect of an information driven life cycle has the potential of being tedious and prone to error so care must be taken to devise techniques which utilise automation as much as possible and minimise the risk of human error. Once knowledge of known quality is stored, the life cycle will utilise it over time to achieve the development of a high quality product. This will involve disseminating the information to people who need it at the time they need it, monitoring the quality of the information as it changes over time, allowing querying of information. Finally it would be useful to be able to predict product quality based on the quality of the information being used by the development process. It is therefore perceived that the life cycle consist of the components shown in the figure 1.

It is the intention of the authors of this paper to carry out further research in this area so as to develop such a life cycle. Preliminary work carried out in this area suggests that such a life cycle will require interdisciplinary contributions from areas such as computer science, psychology and educational theory.

9 References


[10] DeMarco T., Lister T., Peopleware - Productive Projects And Teams, 1999


Metamodel based Model Transformation Framework

Xiaoping Jia, Hongming Liu, Lizhang Qin, and Adam Steele
School of Computer Science, Telecommunication and Information Systems
DePaul University, Chicago, Illinois, USA
Postal Address: 243 S Wabash, Chicago, IL, 60604 email: {xjia, jordan, lqin, and asteele}@cs.depaul.edu
Phone: 312-362-6251, 8756, 8265, 6247
If admitted, presenter: Hongming Liu

ABSTRACT
Model Transformation is the key aspect of Model Driven Engineering (MDE), a model-centric software development approach aiming at improving the quality and productivity of software development processes. This paper presents a metamodel based model transformation framework, which is based on a formal modeling notation – Z-based Object-Oriented Modeling notation (ZOOM). It includes a set of supporting tools aiming at delivering the benefits in practical applications of model driven engineering. The potential benefits of the proposed model transformation framework include: 1) readability and rigorosity of meta-model definitions; 2) simplicity of transformation definition; and 3) extensibility of transformation templates. The architecture and design of the framework is discussed and comparisons with related research work are provided to show the benefits of this framework.

KEYWORDS
Model Driven Engineering, Modeling, Metamodeling, Model Transformation

1 Introduction
Model Driven Engineering (MDE) tackles the elusive problem of system development by promoting the usage of models as the primary artifact to be constructed and maintained [1, 2]. MDE shifts software development from a code-centric activity to a model-centric activity. Accomplishing this shift entails developing support for modeling concepts at different levels of abstraction and transforming abstract models to more concrete descriptions of software. In another word, MDE reduces complexity in software development through modularization and abstraction [3].

Because of MDE’s potential to dramatically change the way we develop applications, companies are already working to deliver supporting technologies [4]. However, there is no universally accepted definition of the requirements for a MDE infrastructure and many requirements for MDE support are unclear or even unspecified. Notwithstanding the lack of standards, with careful reading of related researches [5], we argue that the main issues MDE infrastructure is facing are: (1) incompleteness in existing modeling notations; (2) lack of effective model transformation mechanism. Considering above issues, we provide an alternate solution, which is a new model-driven engineering framework including a formal modeling notation and a set of supporting tools aiming at the realization of the benefits of true model-driven engineering. We have developed a new formal modeling notation called Z-based Object-Oriented Modeling notation or ZOOM, which is based on the formal specification notation Z [6–8], and several key components of UML-2. ZOOM is a simple, precise, and easy to use modeling language. It has dual representation textual and visual. The syntax of textual representation is defined precisely in EBNF grammar. The formalism of modeling notation provides a solid foundation for metamodeling which is an important factor in model transformation. With a simplified metamodeling design, we are able to develop an extensible model transformation process.

In summary, our work objective is to apply such an overall research approach in realization of MDE focusing on model transformation. The notation and metamodel design lays the foundation of my tool development and the tool development in turn demonstrates the validity and advantage of the design.

This paper is organized as follows: Section 2 presents the ZOOM architecture and notation. Then section 3 covers the characteristics of our model transformation approach and the model transformation process. Section 4 discusses related research work and compares them with our approach and finally Section 5 concludes the paper.
2 ZOOM Architecture and Notation

While UML-2 is widely used as a visual modeling language to support MDE, it has several weaknesses. UML-2 is not specifically designed for MDE, so its models are generally informative without providing definitive views. Also, while UML-2 provides multiple visual views to present similar aspects of a software design, it lacks an inherent mechanism to enforce consistency between these views. Additionally, UML-2 lacks user and system interface design notations. To overcome these obstacles, we propose to enhance the UML-2 models and meta-model to include support for formal syntax and semantics and to provide a new UI model. The result is a new formal notation called ZOOM. ZOOM stands for Z-based Object-Oriented Modeling. It is based on the formal specification notation Z [7–9], which is in turn based upon set theory and mathematical logic. ZOOM provides a human-readable syntax to specify the mathematical model in Z. A complete description of how ZOOM supports Z notation can be found in [10].

Figure 1 shows the overall architecture of the ZOOM models for a software system. The functional requirements derive the structural, behavioral and UI models. ZOOM provides a pre-defined event model, which is processed by an event-driven framework, to bind the structural, behavioral, and UI models together [11]. The integrated ZOOM model will be processed by the Knowledge-based Model Compilation Tools resulting in different implementations of the software system based on the specific platform and knowledge base.

3 Metamodel based Transformation Approach

Figure 2 shows the basic structure of our model transformation framework. A transformation engine takes the Hierarchical Relational Metamodel(HRM) defined source model as input, and use a template comprise of a set of transformation rules to produce output model in a format specified by the templates. In another word, the output from the transformation engine is a transformation of the input model. We will discuss the characteristics of our approach in the following subsections.

3.1 Source Model Representation

We use ZOOM notation to represent Platform Independent Model(PIM). ZOOM notation has a textual syntax defined by BNF, which gives us a simplified way to define and use ZOOM metamodel. Since ZOOM provides the textual syntax in a form that most programming languages have, we are able to build an internal representation of ZOOM models in a structure similar to Abstract Syntax Tree(AST), only the node in the tree will be constructs of the modeling language instead of constructs of programming language. However, to capture more complicated modeling language constructs like association. We adopt mathematical collection to depict the relationships of different constructs. Considering it’s tree structure and such relationships, we name our metamodel Hierarchical Relational Metamodel(HRM).
3.2 A Metamodeling Language

Metamodeling is a critical part of our transformation approach. It provide a mechanism to unambiguously define modeling languages - ZOOM in our case. It is the prerequisite for a model transformation tool to access and make use of the models. We will now look into the design of our Hierarchical Relational Metamodel (HRM).

3.2.1 Hierarchical Relational Metamodel

The fact that ZOOM notation has a textual syntax defined by BNF gives us a simplified way to define and use ZOOM model’s metamodel. From implementation point of view, metamodel defines the internal representation of models. In programming language, this internal representation often takes the form of Abstract Syntax Tree (AST) that can be processed by interpreter or compiler [12,13]. Since ZOOM provides the textual syntax in a form that most programming languages have, we are able to build an internal representation of ZOOM models in a structure similar to Abstract Syntax Tree. The only difference is the nodes in the tree are constructs of the modeling language instead of constructs of programming language. To capture more complicated modeling language constructs like association, we also adapt mathematics collection to depict the relationships of different constructs.

3.2.2 HRM Definition

We provide the following definition of HRM:

**Definition 3.1.** Hierarchical Relational Metamodel is a 3-tuple: HRM = (N, C, R)

- N is a set of nodes: N = \{n_1, n_2, ..., n_j\}
- C is a relation on N × N, which forms a tree structure that has one root and no unconnected nodes. Each node may have zero or more children. In another word, a node is either a leaf (i.e. with no children) or can be decomposed as one or more children and each child forms a subtree
- R = \{r_1, r_1 ... r_k\} is a set of relations between nodes, where r_i is a relation on N × N.

Figure 3 shows a simple class diagram that has four classes: Student, Graduate, Undergraduate and Course. The corresponding HRM diagram is also show in Figure 3 in the middle. This metamodel can be represented as (N, C, R) according to Definition 3.1. More specifically, we can elaborate the contents of its three components as:

<table>
<thead>
<tr>
<th>Component</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>{ ClassDiagram, Student, Graduate, enroll, x, y, Student.name, Graduate.advisor, ... }</td>
</tr>
<tr>
<td>C</td>
<td>{(Student, Student.name),(Graduate, Graduate.advisor), (Graduate, Graduate.thesis), ... }</td>
</tr>
<tr>
<td>R</td>
<td>{ superClass, subClass, leftAssociationEnd, rightAssociationEnd }</td>
</tr>
<tr>
<td>superClass</td>
<td>{(x, Student), (y, Student)}</td>
</tr>
<tr>
<td>subClass</td>
<td>{(x, Graduate), (y, Undergraduate)}</td>
</tr>
<tr>
<td>leftAssociationEnd</td>
<td>{(enroll, Student)}</td>
</tr>
<tr>
<td>rightAssociationEnd</td>
<td>{(enroll, Course)}</td>
</tr>
</tbody>
</table>

The components superClass, subClass, leftAssociationEnd and rightAssociationEnd are relations between classes Student, Graduate, Undergraduate, Course and relationship enroll, x, y.
3.3 Transformation Template

The rule set shown in Figure 2 is a collection of transformation rules. Here we provide the definition of transformation rule as followings:

Definition 3.2: A transformation rule \( r = P \rightarrow (T_{pre}, T_{post}) \) where \( P \) defines the pattern to select the element of source model and the template pair \( (T_{pre}, T_{post}) \) defines the mapping to target model. Respectively \( T_{pre} \) defines the mapping to target model before traversing children of selected element, and \( T_{post} \) defines the mapping to target model after traversing children of selected element. The rationale of this design is closely related to the transformation algorithm that we will talk about in the next sub section.

In our framework, the development of transformation is in a large part the process of constructing transformation rules. The rule set in the Figure 2 is an extensible component. Different set of templates can be used in different transformation tasks for various target platforms. That’s why we also call the template “cartridge” to reflect the exchangeability of templates.

3.4 Transformation Algorithm

Metamodel based transformation uses the elements of metamodel. Our adopting of Hierarchical Relational Metamodel(HRM) allows us to build an internal representation of ZOOM models in a structure similar to Abstract Syntax Tree(AST). Once metamodel is generated as an AST like structure, it is accessible by the transformation process through traversing the tree.

```plaintext
1 transformNode(node, ruleSet, outputModel)
2 rule < - findMatchingrule(node, ruleSet) //finding the matching rule for this node
3 targetText < - instantiate(rule.pre, node)
4 outputModel.append(targetText) //getting the output text by applying pre part of the rule
5 foreach c is a child of node
6 transformNode(c, ruleSet, outputModel)
7 endforeach //traverse all the children nodes and do transformation on each of them
8 targetText < - instantiate(rule.post, node)
9 outputModel.append(targetText) //getting the output text by applying post part of the rule
```

We use an algorithm of “pre-order” to traverse of the tree which means each node is visited before its children are visited and the root is visited first. The algorithm is exemplified by the pseudo code shown in the above pseudo-code.

As we can see in Definition 3.2, a transformation rule has two mapping part, \( T_{pre} \) and \( T_{post} \). They are represented as \( rule.pre \) and \( rule.post \) in the pseudo code. Shown in the above pseudo code: \( rule.pre \) is the mapping before traversing children of selected element, while \( rule.post \) is the mapping after traversing children of selected element. And as shown in line 5-7 in the pseudo code, each node in the metamodel will be...
visited once and all its children node will get visited. To trigger the transformation algorithm, the root node of source metamodel need to be passed, in the form of `transformNode(root, ruleSet, outputModel)`

### 3.5 Model Transformation Process

To start the MDE process we need to build a platform-independent model that comprises the essence of target software system. This is the only model that the developer will create completely “by hand.” The other models are mostly generated. The PIM for Community Sports League is depicted in Listing ??.

The complete transformation process is depicted in Figure 4. It is divided into 5 individual steps. Now let’s walk through the transformation process step by step.

#### 3.5.1 Parsing the source model

A source model is provided, in our case, student.zoom in Listing ??.

Parsing involves reading actual source code of the model, or the textual representation of model roster and splitting it into understandable language symbols. This is made possible by ZOOM’s formally defined syntax. A parser will parse the textual representation of model roster and generates an internal abstract syntax tree (AST) representation of roster, which is an object tree.

#### 3.5.2 Traversing the object tree

Once metamodel is generated as AST, which is accessible by the transformation process through traversing the tree. We use an algorithm of “pre-order” traversal of the tree which is introduced in section 4.3. It means each node is visited before its children are visited and the root is visited first. The traversing is exemplified by pseudo code shown in section 3.4. Since the matching metamodel node with rule(step 3) and generating target text(step 4) happen during the process of traversing. The pseudo code in section 3.4 actually shows all of these 3 steps. To trigger this transformation algorithm, the root node of source metamodel need to be passed. In our case, it is passing as `transformNode(m1, ruleSet, outputModel)`, since `m1` is the root node.

#### 3.5.3 Matching node in object tree with rule

The key step in this transformation process is applying rules to source models represented by their metamodels. At this point, transformation engine enters into the picture. As implied in transformation rule definition, `r = (P → (T_{pre}, T_{post}))`, to apply a rule on a certain model include both matching the pattern `P` and implement mapping `T_{pre}` and `T_{post}`. The pattern `P` is specified to make sure that right node is being located and used. In the pseudo code in section 3.4, `rule.pre(line 3)` and `rule.post(line 8)` function as a template for the transformation. When implement a template,
3.5.4 Generating target text

The generation of target text can be as simple as output the text included in the rule pre or emph post elements as shown in pseudo code in section 3.4 line 3 and line 8. However, more complicated scenario can be involved in this step. For example, in most of the cases, expressions are in different forms in source and target model. We provide an extensible mechanism to assist the transformation, or mapping. The result of these steps is Literate Target Code. It will be used as input in the final step, post processing.

3.5.5 Post processing

The Literate Target Code generated in Step 4 as shown in Figure 4 may or may not in a desirable order that fits to the target technical platform. Post processing is responsible to rearrange the Literate Target Code in a desirable style that fits to the target technical platform. Here we treated the Literate Target Code as pieces of segment that can be flexibly rearranged. A post process goes through all these pieces and place each of them in right places in final models or code. This frees the model transformation engine from the details of contextual requirements of target platform. This approach has a similar style as proposed in Knuth’s Literate Programming [14].

After all the above transformation steps, the results are models or source code of target platform. In our example, the results are a group of Java source code.

4 Related Work

A number of partial solutions to describe and implement model transformation are currently available. Some of these are applicable only in a limited domain, or provide very low-level abstractions for transformations [15].

4.1 AndroMDA

AndroMDA [16] is a code generation tool that takes a UML model as input and generates source code as output. It adopts a template-based transformation methodology similar to ours in a degree but differs significantly in handling of metamodel. Using a series of template files (which you can customize if you wish), AndroMDA can produce source code from a UML model in any programming language. Default templates exist to generate Java code (and in particular J2EE code). AndroMDA was designed to get the information necessary to generate code from MOF compliant models inside a MOF repository.

Both AndroMDA and our approach are template-based, metamodel-based model transformation frameworks that support code generation. Both have an extensible architecture consists of cartridges. These cartridges generate the code specific to a certain concrete technical platform. However, the fundamental difference between these two approaches are the metamodel that they base upon. AndroMDA uses MOF while we use HRM. Because of the complexity of the MOF compliant metamodel, when AndroMDA traverses its AST objects, it has to access them via a proprietary JMI interfaces, metamodel facades. Comparing to AndroMDA, our approach simplified the transformation template development by adopting a more concise metamodel.

4.2 QVT(Queries/Views/Transformations)

In the Model-Driven Architecture(MDA), QVT (Queries/Views/Transformations) is a standard for model transformation defined by the Object Management Group. Presently there are several products (commercial or open source) that claim compliance to the QVT standard. QVT defines a standard way to transform source models into target models. Duddy et al propose a transformation language which will meet the requirements of QVT RFP, and several others besides [17]. The language is declarative and patterns based. Transformation descriptions are explicitly reusable and modular. Rules that make up such descriptions may be aspect-driven, allowing for transformations to be written to address semantic concepts rather than structural features.

Since the abstract syntax of QVT conform to MOF 2.0 metamodel, one of the strength in our approach again is adopting a concise, tree-structure metamodel, HRM. Because of the simplified metamodel, when transformation engine traverses its AST objects, it can have direct access to the properties of the objects. This facilitates model transformer to develop transformation template in a easier and quicker way.
5 Conclusion

In this paper present a template based model transformation framework using Hierarchical Relational Meta-model (HRM). By adopting Z-based Object-Oriented Modeling notation (ZOOM) as the formal modeling notation, this model transformation framework provides benefits consist of: 1) readability and rigorousness of meta-model definitions; 2) simplicity of transformation definition; and 3) extensibility of transformation templates.

The current development of this project has made substantial progress and further research effort will be mainly focusing on two things: one is to extend the capacity of current framework; the other is to further prove the validity of this research by building more sophisticate cases. With these two major parts in place, we can further compare our approach with other model transformation mechanisms to verify the advantages of our framework, which is providing a simple, effective, and practical way to define model transformations.

References

A Comparison of Software Process Models

Lachana Inchaiwong¹, A.S.M. Sajeev², Xiaodi Huang³ and Sakgasit Ramingwong²

¹Department of Computer Engineering, Chiang Mai University, Chiang Mai, Thailand
²School of Science and Technology, University of New England, Armidale, NSW, Australia
³School of Business and Information Technology, Charles Strut University, Albury, NSW, Australia

Abstract - Software processes make a critical contribution to successful software projects. A great challenge is to choose an appropriate process for a project. However, since process specifications vary widely in their quality and level of detail, selecting the most appropriate process could be very confusing and misleading. Therefore, a systematic study of process models is needed. Software Process Analysis Method (SAM) is created to assist developers in comparing and analyzing software process models.

Keyword - software process model, process specification, process analysis method, process comparison, process selection.

1 Introduction

The significance and the diversity of software development lifecycles and methodologies have created a situation where a thorough study and analysis of various processes is needed to provide researchers and developers a better understanding and realization of process characteristics and their differences. Software process selection can ultimately lead the project to either success or failure.

Over the years, a large variety of software processes has been developed and applied by software engineers. Waterfall model, arguably the best-known software process, is now a forty-year-old concept. Due to uncountable scholarly feedback and criticism, this classic software process has resulted in a number of variations. In the same way, many other better-known processes such as Spiral model have more than one variations and applicability [1] [2].

Generally, a software process contains information of activities, objectives, conditions, time, and other related information. This information is usually demonstrated by using a model and associated descriptive text. Several software process modeling approaches have been proposed by researchers. Many of them are variations of existing software processes designed to enhance their abilities to cope with various derivative aspects [3] [4] [5] [6] [7] [8] whereas there are others which are entirely new concepts [9] [10]. Although the approaches are different, all attempts share the same purpose – to provide structure and discipline to software development activities.

Choosing a software process is not difficult. However, choosing the most appropriate process can be extremely challenging. The Waterfall model, even though has been subjected to a large amount of criticisms, is still in favor in several software organizations. Although the required software process characteristics is known, due to their great variety and variation, selecting the most appropriate software process is often difficult. A systematic analysis tool will greatly assist software engineers to choose an appropriate software process.

2 Research Problems

There are several problems in software process analysis. Firstly, the terminology used by different process developers can be confusing. It is not very surprising to see a term in one process being used for a totally different concept in another process. In the same way, it is also common to see a same concept being described using different terms by two different modelers. For example, user requirements are called “Feature” in Feature Driven Development while it is called “User Stories” in Extreme Programming [11] [12].

Another serious problem in software process analysis is that processes are defined at different levels of detail. A software process with five steps could be even more complex than a ten-step process. This is because each step in the first software process may consist of several sub-processes of which some may be very detailed.

Descriptions of software processes are often expressed in textual English and/or using diagrams. Often, there is insufficient explanation of the notations used in the diagrams. These make comparison of processes difficult. In order to properly analyze software processes, standardization is required. Once the notation is standardized, an efficient software process analysis method will be able provide insights into software processes.

3 Process Dictionary

As stated earlier, it is not uncommon for different software processes to use different terms to mean the same thing, or the same term to mean different concepts. Such terminological ambiguity can include vital key components in the software process such as activities, objects and other artifacts. This could generate confusion. For instance, for a...
software engineering, the term “Spike” may not come across as something to do with software processes unless they are familiar with Extreme Programming where it stands for a simple program developed to explore potential solution to tough design questions. A new process specification in future may use the term for something completely different such as a software bug or an unexpected obstruction.

Table 1. A Process Dictionary

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Granularity</th>
<th>Description</th>
<th>Synonym</th>
<th>Related</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acceptance test</td>
<td>Artifact</td>
<td>Atomic</td>
<td>A test written by the customer, (or QC on the customer's behalf) that tests the entire system to ensure that a specific piece of functionality is present and functions correctly.</td>
<td>Customer test, Test</td>
<td></td>
</tr>
<tr>
<td>Acceptance testing</td>
<td>Activity</td>
<td>Non-Atomic</td>
<td>Formal testing conducted to determine whether or not a system satisfies its acceptance criteria and to enable the customer to determine whether or not to accept the system.</td>
<td>Customer testing, Formal testing</td>
<td></td>
</tr>
<tr>
<td>Coding</td>
<td>Activity</td>
<td>Non-Atomic</td>
<td>(1) In software engineering, the process of expressing a computer program in a programming language. (2) The transforming of logic and data from design specifications (design descriptions) into a programming language.</td>
<td>Code, Implementation</td>
<td></td>
</tr>
<tr>
<td>Document</td>
<td>Activity</td>
<td>Atomic</td>
<td>(1) A medium, and the information recorded on it that generally has permanence and can be read by a person or a machine. Examples in software engineering include project plans, specifications, test plans, user manuals. (2) To create a document as in (1).</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A process dictionary is a key component in software process standardization. It contains collections of terms used in software process models and their connections to other related terms. The purpose of the process dictionary is not only to introduce a set of terms that can be universally used across all software processes but also to minimize the inessential repetition of terms. The process dictionary assists software engineers to precisely define process terms and map related terms defined in various software processes into common ones. Therefore, a systematic and balanced comparative analysis of software processes can be made. Table 1 is an example of a process dictionary.

The process dictionary, $D$, includes the definition of process terms and their relation to other terms. It initially contains terms taken from IEEE Glossary of Software Engineering Terminology [13] and IEEE/EIA 12207.0-1996 (Standard for Information Technology-Software life cycle processes) [14]. New terms can be added by the process analyst as appropriate.

The process dictionary is a collection of entries, $e$. Each entry involves a unique term referred as common term $t_e$. Each term is related to other terms with different degrees of similarity. In other words, a term is associated with a set of synonyms, $S_e$, and a set of related terms, $L_e$. In most cases, a term has only one description. However, a term can have more than one description if its meaning is different in different process models. A set of meanings of the term is referred by $Meaning(t_e)$. $T$ is a set of all terms.

Determining similarity and related terms is complicated and can be confusing, especially with a large collection of terms. Automatic determination can be performed by computers. However, in order to maximize the accuracy, human justification is still needed at the final stage. The following semantic is used for determining relationship and measurement of the term.

$$e = (t_e, Meaning(t_e), S_e, L_e), \text{ where}$$

$$\forall e_1, e_2 \in D : t_{e_1} \neq t_{e_2}$$

$$t_e \in T$$

$$S_e \subseteq T$$

$$L_e \subseteq T$$

Synonyms and related terms to term $t_e$ are determined by the degree of similarity between $Meaning(t_e)$, $Meaning(S_e)$ or $Meaning(L_e)$. Since $Meaning$ is a set, at a time, only one meaning from each set is considered. The higher the degree of similarity is, the higher the probability that two terms are related.

The term “Closeness” is used for referring to quantification of term-similarity. Theoretically, for two software process terms, $t_1$, $t_2 \in T$, $\text{closeness} C$, or $C(Meaning(t_1), Meaning(t_2))$, indicates the closeness in meanings between two terms. The value of $C(Meaning(t_1), Meaning(t_2))$ is between 0 and 1. Table 2 describes the meanings of closeness.

Table 2. Descriptions of Closeness

<table>
<thead>
<tr>
<th>$C$ (Closeness)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$t_1$ and $t_2$ have no relationship</td>
</tr>
<tr>
<td>1</td>
<td>$t_1$ and $t_2$ are identical</td>
</tr>
<tr>
<td>$0 &lt; C &lt; 1$</td>
<td>$t_1$ and $t_2$ are related but not identical</td>
</tr>
</tbody>
</table>

In order to automatically quantify the closeness of software process terms, an open source software called Reqsimile is adapted. Reqsimile is a Java application...
based on research led by Johan Natt och Dag [15]. The similarity measurement is based on Cosine similarity. Although ReqSimile is originally designed for tackling problems on requirement sets, it can be efficiently adapted to perform the process terminology comparison. In this research ReqSimile’s database is altered in order to suit the content of the previously described process dictionary. Upon selection of a term in a process model, ReqSimile can efficiently determine the closeness to all terms in the process dictionary.

The preparation of the analysis is based on the following rules:

- Each process term is verified for its existence in the process dictionary. If it exists and its meaning is identical to the meaning defined in the dictionary, no update is required.
- If the term exists in the dictionary but there are no suitable meanings, a new meaning is added to an existing term.
- If the term is not in the dictionary and cannot be substituted by any terms in the dictionary, it will be added to the dictionary as a new term.
- Then, the term’s connection to other terms in the dictionary is determined.

The algorithm for developing the process dictionary is illustrated in Figure 1.

```
for each process model, p  
for each term, ti in p  
let the meaning of ti in p be m  
if there exists an entry v in D such that  
if ti equals t  
if there is an element me in Meaning(t) such that C(m, me) equals 1  
break; // the entry is already in dictionary  
else if for all m in Meaning(t), C(m, me) equals 0  
// new meaning  
Meaning(t) = Meaning(t) ∪ {m}  
break;  
// go to next term in p  
else  
break;  
// not possible  
else  
let the meaning of ti in p be m  
if there is an element me in Meaning(t) such that C(m, me) equals 1  
{  
S = S ∪ {t}  
// add t as a synonym for entry e  
break;  
// go to next term in p  
if there is an element me in Meaning(t) such that 0 < C(m, me) < 1  
L = L ∪ {t}  
// t is a related term to t  
break;  
D = D ∪ {t}  
// add t as a new term to the dictionary  
continue;  
// to explore relations with other terms  }
```

Figure 1. The algorithm for developing the process dictionary.

### 3.1 Root terms

A concept called *root terms* is introduced to establish the baseline for comparison of process terminology. Root terms set the foundation for process terms in the process dictionary. All terms in the process dictionary are either directly or indirectly related to at least one of the root terms, either by having a close meaning or by being a part of the root terms. Thirteen root terms are derived from the development process defined in IEEE/IEA12207 international standard [17].

An additional root term, supporting processes, is added to represent supporting activities which are present in many software processes. For analysis purposes, these fourteen root terms are categorized into six groups, i.e., Planning, Specifying, Designing, Coding and Testing, Delivering, and Supporting. Table 3 displays all root terms used in this research.

<table>
<thead>
<tr>
<th>Group</th>
<th>Root terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Planning</td>
<td>1. Process implementation</td>
</tr>
<tr>
<td>Specifying</td>
<td>2. System requirements analysis</td>
</tr>
<tr>
<td></td>
<td>3. Software requirements analysis</td>
</tr>
<tr>
<td>Designing</td>
<td>4. System architectural design</td>
</tr>
<tr>
<td></td>
<td>5. Software architectural design</td>
</tr>
<tr>
<td></td>
<td>6. Software detailed design</td>
</tr>
<tr>
<td>Coding &amp; Testing</td>
<td>7. Software coding and testing</td>
</tr>
<tr>
<td></td>
<td>8. Software integration</td>
</tr>
<tr>
<td></td>
<td>9. Software qualification testing</td>
</tr>
<tr>
<td></td>
<td>10. System integration</td>
</tr>
<tr>
<td></td>
<td>11. System qualification testing</td>
</tr>
<tr>
<td>Delivering</td>
<td>12. Software installation</td>
</tr>
<tr>
<td></td>
<td>13. Software acceptance report</td>
</tr>
<tr>
<td></td>
<td>14. Supporting processes</td>
</tr>
</tbody>
</table>

The definition of root terms is as follows:

Let \( r \) be a root term and \( R \) be the set of all root terms

\[
R \subset T, r \in R
\]

\[
R = \{ \text{Process implementation, System requirements analysis, System architectural design, Software requirements analysis, Software architectural design, Software detailed design, Software coding, Software testing, Software integration, Software qualification testing, System integration, System qualification testing, Software installation, Software acceptance support, Supporting Processes}\}
\]

A root term is defined based on the following condition:

1. It is not close to any root terms, that is:

\[
\forall r_1, r_2 \in R \mid C(r_1, r_2) = 0
\]

2. Every term is directly or indirectly connected to at least one root term

\[
\forall t \in T \mid \exists \ r \in R \mid 0 < C(t, r) \leq 1
\]

### 4 Software Process Analysis Method (SAM)

Software process analysis method (SAM) is introduced to promote better understanding of software processes. This includes their similarities, differences and relations. The result gained from SAM can be efficiently used for assisting in process selection, process tailoring, process adoption, process re-design or other related areas.
SAM involves three main steps: (1) Elaboration, (2) Normalization, and (3) Abstraction. These essential steps are explained in following subsections.

4.1 Elaboration

Two approaches are commonly used for defining software processes, i.e., top-down and bottom up. In a top-down approach, the highest level is firstly described and then refined into smaller details. On the other hand, in a bottom-up approach, the individual detailed elements are defined first and then joined to form a higher level. In many cases, a combination of both approaches is used.

In SAM’s elaboration stage, the process definition is mapped to cover all details specified in the base references. A base reference is ideally the standard reference for the process definition. However, if a standard reference is not available, which is normally the case, the most available authoritative reference(s) is chosen. In this stage, *Order*, a step-wise software process decomposition notation, is used for elaborating the process specification [18]. The elaboration usually follows the top-down approach. The software process is elaborated until no further elaboration is possible. The final elaboration is determined either when all details in the base references are captured and/or when each element in the elaborated model reach atomic state. An atomic element is the smallest element which cannot be further divided.

4.2 Normalization

In general, a process term describes a process activity in a natural language phase. Each term has at least one meaning. In addition, the process term may have relationships to others terms in the process dictionary. For example, the terms “software requirement specification” and “write user stories” are very closely related. Obviously, using only a common logic might not be able to specify the relationship of these two terms. Fortunately, SAM can efficiently judge the relationship of both terms by systematically comparing their descriptions.

As stated previously, in order to be able to compare processes on equal footing, each process element needs to be defined using the same standard terminology. This can be done by using the process dictionary.

After the elaboration is finished, the normalization stage begins. In this stage, SAM, assisted by Reqsimile, automatically determines and replaces non-standard terms by their corresponding standard terms from the previously defined process dictionary. For example the term “user stories” defined in XP is replaced by “user requirements” and the term “sprint” used in Scrum is replaced by “iteration”.

The normalization stage ends when all elaborated specific process terms are replaced with dictionary terms. However, in some cases, the descriptions of the process terms can be extremely brief and vague. In such cases, human intervention is needed to improve the accuracy of the replacement.

4.3 Abstraction

Once every term is normalized in the normalization stage, all similar activities are combined to provide an abstract view of each process model. Through the abstraction step, SAM establishes the baseline for process comparison by relating normalized items to their root elements.

The normalized terms are further abstracted into root terms. This allows the process engineer to recognize the root of the activities. Thus, various process models can be compared.

In this stage, detailed abstract models of each software process are created. These abstract models are flow-chart like diagrams consisting of six nodes, i.e., Planning, Specifying, Designing, Coding & Testing, Delivering and Supporting, as defined in the previous section. Multiple occurrences of the same root term are compressed to one node. The frequency of occurrence is indicated by the thickness of the node. Similarly, the connections between activities are compressed and their frequencies are also indicated by the thickness of the arrow connecting each node. The filled circles indicate the starting and ending points of the process. Figure 2 is an example of the abstract model created by SAM’s abstraction stage.
The abstract model provides some valuable insight of the process nature. From the abstract model, it can be easily seen that some processes focus heavily on specifying while some other processes do not concentrate on this node at all. The case study in the next section provides a more concrete example of this stage.

Figure 3. A top-level view of the Waterfall model in Order notation

Figure 4. Waterfall’s Elaboration: Preliminary Program Design in Order notation

5 Case Study

In this case study, SAM is used for analyzing the classic Waterfall model. The base reference of the Waterfall model is [19]. This base reference describes the process in top-down perspective. Figure 3 illustrates the top level view of the Waterfall process in Order notation, as derived from the base reference.

5.1 Elaboration

This top level of the Waterfall model displays eight modules. Four of them can be further elaborated. Figure 4 shows an example of the second level elaboration from the preliminary program design module. In this case, the elaboration ends after the second turn since all activities reach atomic level.

5.2 Normalization

After the elaboration is completed, all final terms are normalized based on the process dictionary. Table 4 displays the process terms and normalized terms.

5.3 Abstraction

The normalized terms are further abstracted in this stage. Table 5 exhibits the abstracted terms of the Waterfall model.

With these abstracted terms, an abstract model of the Waterfall model is created as shown in Figure 5.

According to Figure 5, the main emphasis of Waterfall process is designing. Specifying and Coding & Testing are the second most important stages of this process. Interestingly, planning does not receive much attention as other development phases. Furthermore, the dotted supporting node indicates that the Waterfall model does not define any supporting processes at all.
Table 4. Normalized Terms of the Waterfall Model

<table>
<thead>
<tr>
<th>Waterfall process terms</th>
<th>Normalized terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>System requirements specification</td>
<td>Requirements specification</td>
</tr>
<tr>
<td>Software requirements specification</td>
<td>Software requirements specification</td>
</tr>
<tr>
<td>Document system overview</td>
<td>Document</td>
</tr>
<tr>
<td>Design, Specify and Allocate Data Processing Modes</td>
<td>Design</td>
</tr>
<tr>
<td>Define operating procedures</td>
<td>Design</td>
</tr>
<tr>
<td>Preliminary Design Review</td>
<td>Preliminary design review</td>
</tr>
<tr>
<td>Requirements Analysis</td>
<td>Requirement analysis</td>
</tr>
<tr>
<td>Software design and documentation</td>
<td>Design</td>
</tr>
<tr>
<td>Test planning</td>
<td>Develop plan</td>
</tr>
<tr>
<td>Test planning Documentation</td>
<td>Document</td>
</tr>
<tr>
<td>Critical Software Review</td>
<td>Critical design review</td>
</tr>
<tr>
<td>Coding</td>
<td>Coding</td>
</tr>
<tr>
<td>Visual Code Inspection</td>
<td>Inspection</td>
</tr>
<tr>
<td>Integration</td>
<td>Integration</td>
</tr>
<tr>
<td>Final Software Acceptance Review</td>
<td>Review</td>
</tr>
<tr>
<td>Software Installation and Documentation</td>
<td>Installation and checkout phase</td>
</tr>
<tr>
<td>Software System Diagnostic test</td>
<td>Testing</td>
</tr>
<tr>
<td>Functional Enhancement</td>
<td>Operation and maintenance phase</td>
</tr>
</tbody>
</table>

Table 5. Abstracted Terms of the Waterfall Model

<table>
<thead>
<tr>
<th>Normalized terms</th>
<th>Root terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements specification</td>
<td>System Requirements analysis</td>
</tr>
<tr>
<td>Software requirements specification</td>
<td>Software Requirements analysis</td>
</tr>
<tr>
<td>Document</td>
<td>System architectural design</td>
</tr>
<tr>
<td>Design</td>
<td>Software architectural design</td>
</tr>
<tr>
<td>Design</td>
<td>Software architectural design</td>
</tr>
<tr>
<td>Preliminary design review</td>
<td>Software architectural design</td>
</tr>
<tr>
<td>Requirement analysis</td>
<td>System requirements analysis, Software requirements analysis</td>
</tr>
<tr>
<td>Design</td>
<td>Software detailed design</td>
</tr>
<tr>
<td>Develop plan</td>
<td>Process implementation</td>
</tr>
<tr>
<td>Document</td>
<td>Process implementation</td>
</tr>
<tr>
<td>Critical design review</td>
<td>Software detailed design</td>
</tr>
<tr>
<td>Coding</td>
<td>Software coding</td>
</tr>
<tr>
<td>Inspection</td>
<td>Software coding</td>
</tr>
<tr>
<td>Integration</td>
<td>Software Integration, System Integration</td>
</tr>
<tr>
<td>Review</td>
<td>Software Acceptance Support</td>
</tr>
<tr>
<td>Installation and checkout phase</td>
<td>Software installation</td>
</tr>
<tr>
<td>Testing</td>
<td>Software Acceptance Support</td>
</tr>
<tr>
<td>Operation and maintenance phase</td>
<td>Software qualification testing, System qualification testing</td>
</tr>
</tbody>
</table>

Figure 5. An abstract model of the Waterfall model

6 Conclusion

Software Process Analysis Method (SAM) is a systematic framework for analyzing the nature of software processes. It comprises of three stages: elaboration, normalization and abstraction. The elaboration stage involves the decomposition of software processes until each activity reaches the atomic stage. Then, the normalization stage removes the confusion of specific terminologies by replacing them with standard dictionary terms. Finally, the abstraction stage categorizes the normalized terms into root terms and compresses them in order to produce a final abstract model. The abstract model reveals the nature of the software process. The information obtained from the abstract model can be used for further comparative analysis of software processes.

In the future, it is possible to minimize a major drawback of SAM in the normalization stage, that is, the occasional need for manual intervention to validate normalized terms. With a more elaborate process dictionary database and by including antonym relationship or artifact behaviors, the accuracy of the automatic normalization of terms is very likely to improve. Additionally, it is also possible to adapt SAM to the use of formal methods in order to support a wider participation of the software community.
7 References


Architecture Description for Model-Driven Development

A. Fatwanto and C. Boughton
1Department of Computer Science, The Australian National University, Canberra, ACT, Australia

Abstract - This paper presents a method to compose an architectural description that is suitable for model-driven software development. The method comprises viewpoints which address separately specification of platform-independent requirements from platform requirements and accommodate the specifications of functional and non-functional requirements. We used OOA/RD method as a basis for developing models within our method since it offers separation between platform-independent requirements with platform requirements. In terms of separation of concerns, our method offers clearer division compared with the specified viewpoints in MDA.

Keywords: Architectural description, MDA, separation of concerns.

1 Introduction

In general, the core concept of Model-Driven Development (MDD) is separating the Platform-Independent Requirements Specification (PIRS) from the Platform Requirements Specification (PRS). Hence, in our method, the PIRS and PRS are developed separately to address each concern differently. PIRS will be transformed into an Implementation Specification (IS) according to a set of rules. The rules are specific to a particular PRS. Therefore, a PIRS can be transformed into many IS using different sets of rules. In other word, PRS couples PIRS with IS. The general concept of MDD is depicted in figure 1.

Transformation rules are the manifestation (interpretation) of architecture specifications. In this case, architecture specifications represent PRS. PRS is part of Architectural Description (AD) and it encloses the specification of architecture. Hence an AD guides the creation of transformation rules where the rules are a manifestation of the PRS.

Particularly, AD provides some specific benefits for MDD such as: promoting separation of concern, design modeling guidance, and tools creation and/or selection guidance.

Despite its usefulness, there is still no well-specified method to compose an AD for MDD especially the one which preserves separation between requirements specifications and platform specifications. In accordance to the definition of MDD standard, i.e. MDA, Object Management Group (OMG) specified 3 viewpoints: computation-independent viewpoint (CIV), platform-independent viewpoint (PIV), and platform-specific viewpoint (PSV) but with no further information regarding the method to compose an AD [10]. Basically, it has three obstacles: it is violating the concept of separation of concerns, it has no clear views that address separately the functional and non-functional requirements concerns, and it has no specific view that addresses the platform domain.

Hence, the purpose of our study is to presents a method to compose an AD that suits to MDD methodology and keep maintaining the concept of separation of concerns, i.e. separating PIRS from PRS.

The remainder of this paper is structured as follows: section 2 discusses works related to this topic, section 3 explains the concept of our proposed method along with examples, and finally section 4 contains conclusion and description of future works.

2 Related Works

The importance of describing software architecture using multiple, concurrent views was first introduced in [7]. This concept is based on separating concerns from various stakeholders’ viewpoints. Further, it handles separately functional and non-functional requirements. [7] proposed 4 views: logical, process, physical, development, and complemented with +1 view which are scenario (as an instance of use case). Scenario describes how all the other 4 views work together. 4+1 method prescribes a number of notations and styles to model each view. This method was designed for generic application, and might be used for any types of development methodology. The idea was then adopted in Rational Unified Process (RUP) with minor
changes in its views name and the introduction to use UML as modeling notation [8].

Following Kruchten’s influential paper, there were many other works in the field of AD. Some were based on practices in a particular domain, while others try to cater for generic applications. Table 1 lists some related works in this area.

More recent work has been done by [11]. Aside from proposing 6 viewpoints and the method to construct them, they also introduced the concept of perspective which is established mainly to handle the quality properties (non-functional requirements). The idea behind perspective is similar to the concept of aspect in Aspect-Oriented Programming (AOP) except that it occurs at the architectural level.

<table>
<thead>
<tr>
<th>Method</th>
<th>Structure</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[7]</td>
<td>4 views + 1 scenario</td>
<td>generic/RUP</td>
</tr>
<tr>
<td>[13]</td>
<td>global analysis + 4 views</td>
<td>real-time &amp; embedded</td>
</tr>
<tr>
<td>[6]</td>
<td>5 views</td>
<td>system</td>
</tr>
<tr>
<td>[4]</td>
<td>14 viewpoints</td>
<td>information system</td>
</tr>
<tr>
<td>[10]</td>
<td>3 viewpoints</td>
<td>MDA</td>
</tr>
<tr>
<td>[2]</td>
<td>3 viewtypes + beyond</td>
<td>generic</td>
</tr>
<tr>
<td>[11]</td>
<td>6 viewpoints + 10 perspectives</td>
<td>information system</td>
</tr>
</tbody>
</table>

Table 1. Collection of works in AD

3 AD for MDD

As already known, models are the first-class artefact in MDD. Therefore, method to compose AD for MDD will be driven by architectural models. In the context of AD, architectural models are accommodated on the section of architectural views where each view may consist of one or more models. Hence, architectural views indirectly will be the first-class artefact in AD for MDD. However, any other elements of AD will also be dealt since views must conform to its associated viewpoint where stakeholders and concerns will influence the rationales behind the establishment of particular viewpoints.

A view is an encapsulation of a set of related concerns which a particular stakeholder might be interested. Viewpoint specifies convention for constructing and using a view [5]. In other word, a view can be seen as an instance of a viewpoint. Then the question is what kind of viewpoint should be considered when composing AD in the translative MDD context? Since translative MDD is a kind of software development methodology, then we must consider its specific characteristics. As explained earlier, the main characteristics of translative MDD is separating the PIRS from the PRS. Therefore, the decisions to establish a particular set of views will mainly be driven by that special characteristic.

In general, our proposed method is designed for translative MDD and it best suits for Concern-Oriented Model-Driven Development (COMDD) methodology [3]. The general concept of COMDD is depicted in figure 2.

3.1 Logical Viewpoint

The logical views describe the applicable features of a system. An applicable feature is represented as a concern at PIRS. A logical view depicts the structures of an applicable feature, its behaviors, and its activities. All models for logical views do not describe any information related to design and implementation. In the MDD context, these views are established to address the functional requirements and operationalizable quality attributes which are independent from platform.
Our classification on non-functional requirements is depicted in figure 3. Operationalizable quality attributes are the quality attributes that can be manifested (specified and derived) as operations or functions. For example, security can be manifested as an authentication function that manages the mechanisms to verify any party that try to access a protected entity. It can also be derived as an encryption function for data that are going to be sent through public channels. On the other hand, non-operationalizable quality attributes are the ones that can not be manifested as operations or functions. For instance, it is technically impossible to specify performance as operations or functions. What we can do is only to check whether this particular attribute has/has not been accommodated into the system specifications.

**Figure 3. Non-functional Requirements Classification**

**Stakeholders.** All stakeholders, for example: acquirers or end-users who are interested in or are affected by the system functional features, as well as developers who are interested in the functional structures.

**Concerns.** This viewpoint addresses the functional requirements and operationalizable quality attributes concerns.

**Model.** Under logical viewpoint, a number of views may be created where each view describes a specific concern. There are two types of models for logical views which can be developed to address the functional requirements and operationalizable quality attributes: class level models and concern level models. Class level models describe the static and dynamic structures of a system’s applicative features at class/object level where the description is in a detail manner. They comprises of functional models, behavioral models, and action models. Meanwhile, concern level models depict a system’s applicative features at concern level where the description is in a rather general form. They consist of interaction models and invocation models.

In the context of AD, concern level models are preferred since they provide descriptions of software architecture in more abstract (high level) fashion which are suitable for describing system-wide logical views. However, class level models are needed as a basis to develop concern level models. The following sections describe models for logical views.

### 3.1.1. Class Level Models

- **Functional Model**

A functional model describes the static structure of an applicative feature which specifies one concern of the system’s functional requirements or operationalizable quality attributes.

**Notation:** xtUML class diagrams [9] can be used to describe functional models. It captures the static structures of an applicative feature. An object is symbolized as an xtUML class. A relationship is symbolized as an xtUML association along with its name and multiplicity, a generalization/specialization association, or an association class. An interface is symbolized as an xtUML class stereotyped with <<interface>>.

**Figure 4. A Snippet of Functional Model for Online Store Application**

**Table 2. Elements of Functional Model**

<table>
<thead>
<tr>
<th>Element</th>
<th>Representation of</th>
<th>Notation (xtUML)</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>entity of the real world concept</td>
<td>class</td>
</tr>
<tr>
<td>relationship</td>
<td>relation among entities</td>
<td>association + text (name and multiplicity), generalization/specialization, association class</td>
</tr>
<tr>
<td>interface</td>
<td>(provided and/or required) service of a certain concern</td>
<td>stereotyped class</td>
</tr>
</tbody>
</table>

- **Behavioral Model**

A behavioral model describes the dynamic structure (behavior/lifecycle) of an active object, an object which has a state-dependent behavior.

**Notation:** xtUML statechart diagrams [9] can be used to describe behavioral models. It captures the behavior/lifecycle of an active object. A status is symbolized as an xtUML state. Initial and final status (particularly for instance having no successive progression) are symbolized as xtUML pseudo-state. There is no specific notation for final status of a
vanished instance. A transition is symbolized as an arrow. An event is symbolized as an xtUML event along with its identifier, meaning, and event data.

Another version of behavioral model is using State Transition Table (STT) [12]. This table lists all possible combinations of statuses and events of an active object. Every row denotes the possible status of the associated active object and every column denotes the possible event to the associated active object. The equivalent STT for the statechart diagram in figure 5 is depicted in figure 5.

<table>
<thead>
<tr>
<th>Event</th>
<th>new</th>
<th>add</th>
<th>checkout</th>
<th>cancel</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Create Order</td>
<td>Ignore</td>
<td>Add Order</td>
<td>Can’t Happen</td>
<td>Can’t Happen</td>
</tr>
<tr>
<td>Add Order</td>
<td>Can’t Happen</td>
<td>Add Order</td>
<td>Complete Order</td>
<td>Cancel Order</td>
</tr>
<tr>
<td>Complete Order</td>
<td>Can’t Happen</td>
<td>Can’t Happen</td>
<td>Ignore</td>
<td>Can’t Happen</td>
</tr>
<tr>
<td>Cancel Order</td>
<td>Can’t Happen</td>
<td>Can’t Happen</td>
<td>Can’t Happen</td>
<td>Ignore</td>
</tr>
</tbody>
</table>

**Figure 5. State Transition Table for Class “Order”**

- **Action Model**

An action model describes any process/action inside a particular status of a behavioral model.

**Notation:** Action Data Flow Diagrams (ADFDs) [12] can be used to describe action models. It captures the behavior/lifecycle of an active object. An action unit is symbolized as a circle. A dataflow is symbolized as an arrow along with the text stating the name of the dataflow. A control flow is symbolized as a dashed arrow along with the text stating its meaning. A data store is symbolized as a tape along with the text stating its name. Finally, an interface is symbolized as an xtUML class stereotyped with <<interface>>.

**Table 4. Elements of Action Model**

<table>
<thead>
<tr>
<th>Element</th>
<th>Representation of</th>
<th>Notation (xtUML)</th>
</tr>
</thead>
<tbody>
<tr>
<td>action unit</td>
<td>process</td>
<td>circle</td>
</tr>
<tr>
<td>dataflow</td>
<td>flow of data</td>
<td>arrow + text (name)</td>
</tr>
<tr>
<td>control flow</td>
<td>restriction toward the procedure of action</td>
<td>dashed arrow with text</td>
</tr>
<tr>
<td>data store</td>
<td>persistent data</td>
<td>Tape with text</td>
</tr>
<tr>
<td>interface</td>
<td>external entity</td>
<td>stereotyped class</td>
</tr>
</tbody>
</table>

**Figure 7. Action Model for “add order” State at Class “Order”**

### 3.1.2. Concern Level Models

- **Interaction Model**
An interaction model describes all asynchronous (signaling) communications along with their associated messages within a concern (applicative feature) which can occur either between internal active objects and/or internal active objects with external entities.

Notation: xtUML Collaboration Diagrams [9] can be used to describe interaction models. It captures the asynchronous communications patterns of applicative features. An active object is symbolized as an xtUML class. An asynchronous communication is symbolized as an arrow along with its identifier, meaning, and event data. An interface is symbolized as an xtUML class stereotyped with <<interface>>.

Invocation Model

An invocation model describes all synchronous (operation invocation) communications along with their associated messages within a concern (applicative feature) which can occur from either internal or external active objects to either internal or external, active or passive objects instance data.

Notation: xtUML Collaboration Diagrams [9] can be used to describe invocation models. It captures the synchronous communications and accesses signals patterns of applicative features. An object is symbolized as an xtUML class. A synchronous communication is symbolized as an arrow along with its identifier. An access signal is symbolized similar as a

3.2 Architectural Viewpoint

Architecture is the organization of a system design. It defines strategies (rules and mechanisms) to manage data, function, and control for the implementation of all software elements of the entire system. Architectural views describe platform and design specifications of a system. Any information related to the applicative features does not specified in these views. In the translative MDD context, these views are established to address the architectural concern where it must be independent from applicative features.

Stakeholders. Developers, especially: architect and designer.

Concerns. Process viewpoint addresses the non-functional requirements concerns (especially non-operationalizable quality attributes).

Model. There are two types of models for architectural views: subsystem level models and system level models. A subsystem level model depicts the design structures of for a particular part of a system. It consists of component models. Meanwhile, system level models describe the design structures
for the whole system. They comprise of platform models and design models.

### 3.2.1. Subsystem Level Models

- **Component Model**

  This model describes the structures of components (software module) for a particular part of a system (such as the application server). It shows how the components are composed into a set of packages, its structure, the dependency between those components, and what types of supporting components such as utility, library, and driver (if any) are needed to execute a system for a particular platform.

  **Notation:** xtUML Package Diagrams [9] can be used to describe component models. It captures the structure of software module that is used in a particular part of the system. A layer is symbolized as xtUML package. Software modules are symbolized by xtUML class. Relation is symbolized by xtUML association without any annotation.

![Figure 11. Component Model for Application Server](image)

<table>
<thead>
<tr>
<th>Table 8. Elements of Module Composition Model</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Element</strong></td>
</tr>
<tr>
<td>layer</td>
</tr>
<tr>
<td>component</td>
</tr>
<tr>
<td>relation</td>
</tr>
</tbody>
</table>

### 3.2.2. System Level Models

- **Platform Model**

  A platform model describes the structures of specific platform technology being applied for the entire system. It also explains any information related to the technology being used. In order to preserve the principle of separation of concerns, this model avoids any descriptions about system’s applicative features. Hence, any allocations of applicative features to the underlying platform are not described in this model. **Notation:** xtUML Class Diagrams [9] can be used to describe platform models. It captures the platform that is used for the entire system. A machine/processor/node/technology-specific function is symbolized as an xtUML class. A communication medium is symbolized as an association along with its multiplicity and meaning. A port/interface is symbolized as an xtUML class stereotyped with <<interface>>.

![Figure 10. Platform Model of Web-based System Using Layered Architecture](image)

<table>
<thead>
<tr>
<th>Table 7. Elements of Platform Model</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Element</strong></td>
</tr>
<tr>
<td>platform</td>
</tr>
<tr>
<td>medium</td>
</tr>
<tr>
<td>interface</td>
</tr>
</tbody>
</table>

- **Design Model**

  This model describes the structures of components (software module) for a particular part of a system (such as the application server). It shows how the components are composed into a set of packages, its structure, the dependency between those components, and what types of supporting components such as utility, library, and driver (if any) are needed to execute a system for a particular platform.

  **Notation:** xtUML Class Diagrams [9] can be used to describe the static design structures of a system. It captures the design structures for the entire elements of a particular system. Element is symbolized as xtUML class. Relations are used to implement the structural rules and will be symbolized by xtUML association along with its name and multiplicity, generalization/specialization association, or an association class. Meanwhile, xtUML State Diagram [9] can be used to describe the dynamic design models (architectural operations/mechanisms). They are described using the same notation as for the behavioral models (see table 3 for the details). Subsequently, the processes/actions of a status can be described using ADFD [12]. They are described using the same notation as for action model (see table 4 for details).

| Table 9. Elements of Design Model |
Another way to describe design models is using specific design language. This notation is created for a specific type of architecture style and pattern. For example, we can use specific design language to describe static design model as shown in figure 14. This architecture implements “multi-task, asynchronous, event-based, object-oriented, direct instance data implementation” pattern.

The processes/actions for each state can be further specified using action specification notation. The following code specifies the processes/actions for state “process” of class action task”.

```
  .create object instance newItem of OrderedProduct;
  .select any item from instances of Product where selected.itemID==rcvd_evt.itemID;
  .relate item to self across Rx using newItem;
  .newItem.quantity=rcvd_evt.quantity;
  .newItem.unitPrice=item.unitPrice;
```

![Figure 14](static-design-model.png)

**Figure 14.** Static Design Model of a Multi-task Architecture Described Using Specific Design Notation

## 4 Summary and Future Work

In this paper we propose a method to compose AD to be used for translatable MDD approach that designed to comply with the IEEE Standard 1471-2000. Our method provides a way to construct architectural model that both separating PIRS from PRS. Our set of viewpoints offers clearer division to address different concerns compared with the specified viewpoints in MDA.

We plan to apply our method in a real project to develop an election system using MDD methodology. We try to find out its effectiveness and drawback in term of separating concern.

## 5 References


Standardization of Software Cost Estimation Process for Outsourcing in Pakistan: Statistical Methods Used for Checking Accuracy of Model Prediction

A. Syeda Umema Hani\textsuperscript{1}, B. Dr. Prof. Muntaz-ul-Imam\textsuperscript{2}, and C. Dr. Gulam Qadir Memon\textsuperscript{3}

\textsuperscript{1,2} Computer Engineering, Sir Syed University of Engineering and Technology, Karachi, Sindh, Pakistan
\textsuperscript{3} Computing & Mathematical Sciences Department, Dadabhoy Institute of Higher Education, Karachi, Sindh, Pakistan

Abstract - This paper is in continuation to the paper [1] that has discussed the case study in very detail; now this paper is highlighting the statistical procedures to be used for accessing the model accuracy level. This research study will be employed as guidance for software houses registered with Pakistan Software Export Body. The project under demonstration is now a complete ERP system a real life project that has been assigned by PSEB to a newly established company as a part of its “Industrial Automation Program”, whose single module was discussed in earlier version of this paper. It highlights two aspects: first, list of estimation readings following the procedures as discussed in previous version of this paper; and second, review and calculations of statistical validation procedures with the need for using which procedure in what circumstances. In lasts the interpretation of result for making the right selection. The estimation model under observation is again COCOMO that satisfies the criteria for model evaluation and supports all possible effort multipliers for justifying software development effort cost under any organizational circumstances and also supports their calibration.

Keywords: Software Cost, Effort Estimation, Estimation, Model Validation, COCOMO, Estimation Accuracy.

1 Introduction and Background

Software outsourcing solutions to international market requires legalization of imposing standardization in software development effort estimation process by SEB which strongly involves the implication of proper model through interaction with research groups as advised in previous paper so that they guide which model needs to be adopted.

A fine-tuning is always required to suit model in specific organization environment. Lots of research work has been done for the identification of empirical models that are either portable across multiple organizations or needed to be calibrated before using them in different environment. effort, quality, and cycle time [6], [7], [8], [9]. The determination of model portability is done by calculating its estimation accuracy in a form PRED accuracy in (\%) = probability of occurrence of that accuracy.

Automated toolkits like COSEEKMO is the fast solution to do so but we need to know how to go through this procedure and what are statistical steps required into it. If the selected model is not accurate enough to support your project environment, or in other words If model is made by using a kind of projects data that were made in less or more productive environment then yours, then that empirical model definitely gives large variance towards under and over estimates [5]. The last stage comes for its calibration by using toolkits like COCONUT [11].

Model accuracy scale checking discussed in last paper was in case when we were performing estimation of single project while conducting a single case study. COCOMO II claims to be a model which is portable across multiple organizations and gives accuracy about PRED (20%)=70% [4].

Methodology adopted here follows:
A) Initially characteristic requirements are used which serves as basis for assigning weights to effort multipliers (EM) and Scaling factors.
B) Then Function Point gets calculated and converted in LOC for targeted technology.
C) After it a list of project data is compiled by applying COCOMO II Post Architecture Model with adapted software development on N project modules treated as N individual projects.
D) Then a table is compiled showing all statistical procedures required for the verification of model accuracy i.e. PRED (20%)=70%.
E) In last results are discussed to advise how to interpret estimates.

2 ERP System: Cost Effecting Requirement Constraints

Following are cost effecting requirements [1].
a. Technical Staff having at least 4 years prior experience with client-server application and are skilled in the use of Visual Basic /ASP/SQL languages.

b. Equipment and Tools are sufficiently available in development environment.

c. In-house technical training sessions.

d. Participates in different technical conferences.

e. The Project Manager also involved organization in practicing maturity levels of Software Process using CMMI’s Software Project Management Process Area as a part of research studies performed.

f. Development technology used is VB.

g. System Requirement Specification submission is must before confirmation of the project contract and also we have to develop complete separate User Manual for each module.

The development of whole ERP was done in RAD style. In which we have used reusable (module)/adaptable code based development; well-understood requirements get converted quickly in to a Working system with in 60 to 90 days period.

![Figure 1: ERP modules](image)

**3 COCOMO II Post Arch. Run**

Function points get calculated by add up each of the individual counts ILFs, EIFs, EIs, EOs, and EQs to get a total unadjusted function point count for each module of ERP separately.

**Table 1: Total Degree of Influence and Value Adjustment Factor**

<table>
<thead>
<tr>
<th>S No.</th>
<th>Factors</th>
<th>Rating Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Data Communication</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Distributed data processing</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>Performance</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>Heavily used configuration</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>Transaction rate</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>Online data entry</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>End user efficiency</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>Online update</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>Complex processing</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>Reusability</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>Installation case</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>Operational case</td>
<td>1</td>
</tr>
</tbody>
</table>

**Table 2: Scaling factor Computation**

<table>
<thead>
<tr>
<th>Scale Factors (Sfj)</th>
<th>Ratings</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREC</td>
<td>Very High</td>
<td>1.24</td>
</tr>
<tr>
<td>FLEX</td>
<td>Very Low</td>
<td>5.07</td>
</tr>
<tr>
<td>RESL</td>
<td>Nominal</td>
<td>4.24</td>
</tr>
<tr>
<td>TEAM</td>
<td>High</td>
<td>2.19</td>
</tr>
<tr>
<td>PMAT</td>
<td>Nominal</td>
<td>4.68</td>
</tr>
</tbody>
</table>

\[ \sum_{j=1}^{5} = 17.42 \]

**Table 3: Effort Multipliers Calculation**

<table>
<thead>
<tr>
<th>Cost Driver</th>
<th>Description</th>
<th>Ranking</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product</td>
<td>RELY Required software reliability</td>
<td>Nominal</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>DATA Database size</td>
<td>Nominal</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CPLX Product complexity</td>
<td>Nominal</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>RUSE Required reusability</td>
<td>Nominal</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>DOCU Documentation</td>
<td>High</td>
<td>1.11</td>
</tr>
<tr>
<td>Platform</td>
<td>TIME Execution time constraint</td>
<td>Nominal</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>STOR Main storage constraint</td>
<td>Nominal</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>PVOL Platform volatility</td>
<td>Nominal</td>
<td>1</td>
</tr>
<tr>
<td>Personnel</td>
<td>ACAP Analyst capability</td>
<td>Very High</td>
<td>0.71</td>
</tr>
<tr>
<td></td>
<td>PCAP Programmer capability</td>
<td>Very High</td>
<td>0.76</td>
</tr>
<tr>
<td></td>
<td>PCON Personnel continuity</td>
<td>Very High</td>
<td>0.81</td>
</tr>
<tr>
<td></td>
<td>APEXP Applications experience</td>
<td>Very High</td>
<td>0.81</td>
</tr>
<tr>
<td></td>
<td>PLEXP Platform experience</td>
<td>Very High</td>
<td>0.85</td>
</tr>
<tr>
<td></td>
<td>LTEX Language and tool experience</td>
<td>High</td>
<td>0.91</td>
</tr>
<tr>
<td>Project</td>
<td>TOOL Software Tools</td>
<td>Nominal</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>SITE Multisite development</td>
<td>Nominal</td>
<td>1</td>
</tr>
</tbody>
</table>

\[ \text{Total Product of EM} = 0.4662 \]

According to the conclusion derived from last case study, COCOMO II’s adopted software development model results were more accurate, although they showed more than half reduction in profit earned, which was the reality of that case study [1].

For Effort calculation the software used here is COCOMO II. 2000.0 by the University of Southern California Center for Software Engineering, which is based on UCS-COCOMO II model. This software is a stand-alone software system intended for a single user and suitable for academic level demonstrations. In professional environment use of Co-Star is preferred [2].

For Adaptation Estimation Scenario tables given bellow depicts scaling factor, effort multiplier, estimation formulae and software run respectively.
4 Interpretation of Actual Effort

Actual effort i.e. mma for ith project is calculated as:

\[ \text{mmai} = \frac{\text{Total Project Cost of labor for ith module excluding misc. charges}}{\text{Average salary of Technical manpower involved in SDLC}} \]

5 Statistical Procedures for Model accuracy

The accurate prediction of software development costs is a critical issue to make the good management decisions and accurately determining how much effort and time a project required for both project managers as well as system analysts and developers. Without reasonably accurate cost estimation capability, project managers can not determine how much time and manpower cost the project should take and that means the software portion of the project is out of control from its beginning; system analysts can not make realistic hardware-software tradeoff analyses during the system design phase; software project personnel can not tell managers and customers that their proposed budget and schedule are unrealistic. This may lead to optimistic over promising on software development and the inevitable overruns and performance compromises as a consequence. Considerable studies are now directed at constructing, evaluating and selecting better software cost estimation models and tools [13][14].

A General Framework has also been proposed which may serve as the “Framework of Analysis for Estimation [102]. In this paper a focus is on the degree to which the model’s estimated effort matches the actual effort. Following are the stepwise options for accuracy Statistics:

1. Discuss detailed real life cost estimation implementation studies to collect n Real Costs.
2. Tabulate the results and then calculating Mean and Standard deviation of all samples of “n” Real Cost values.

A) Calculate E, RE, MRE to Actual:

\[ \text{Absolute Error} = E_i = mme_i - mma_i \]

Problem: size of error varies with the size of project like for 10 mm project error = 9 mm but for 100mm project? [14]

\[ \text{Relative Error} = \frac{mme_i - mma_i}{mma_i} \]

Problem: cannot analyze model’s average performance over the entire set of project. Error (1. mme < mma Under estimates 2. mme > mma Over estimates i.e. addition of system features not required).

\[ \text{Magnitude of relative error to actual MRE_i} = |\text{RE}_i| \]

Solution: Two type of errors cannot cancel out each other when an average of multiple errors is taken.

\[ \text{Bias if model is developed with it in less productive environment then the evaluation site may generate errors that are biased towards Overestimates.} \]
The aggregation of MRE over multiple observations, say N, can be achieved through the Mean Magnitude of relative Error = MMRE % = \(100/N \left[ \sum (MRE_i) \right]\), as a reference, a MMRE < = 0.25 is considered as acceptable for effort prediction models [9] [10]. D sized data set, with Tr sized Train data where T<= D and Ts test set = D – |Train|, here N is basically the Ts.

Solution: mean measure of central tendency and takes into account the numerical value of every single observation in the data distribution. It represents the balance point, or center of gravity.

Problem: The MMRE is sensitive to individual predictions with excessively large MREs i.e. extreme values. Its chief drawback in skewed distributions and when there are one or two outliers among the observations.

Let \(x_1, x_2, \ldots, x_n\) denotes our sample i.e. MREi arranged in increasing order of magnitude then Median Magnitude of relative Error =

\[
\text{MdMRE} = \begin{cases} 
(MRE_{\left[n+1\right]/2}) & \text{for } n \text{ odd} \\
\frac{(MRE_{\left[n\right]/2}) + (MRE_{\left[(n+2)/2\right]})}{2} & \text{for } n \text{ even}
\end{cases}
\]

Solution: Median is probably much more meaningful measure of central tendency for the majority of observations as it is not influenced very much by extreme values.

PRED (N): A complementary criterion that is commonly used is the prediction at some % level \(L\), \(PRED (L\%) = \frac{K}{N} \times 100\), where \(K\) is the number of observations where MRE is less than or equal to \(L\).

In addition to mean and median accuracy figures, practitioners are concerned with the risk of making very erroneous estimates that deviate substantially from the expected inaccuracy. This is evaluated with following test metrics: The Standard Deviation of MRE SDMRE

\[
\text{Mean Abs} = \bar{X} = \frac{\sum |X_i|}{n}
\]

\[
\text{Standard Deviation} = \sqrt{\frac{\sum (X_i - \bar{X})^2}{n-1}}
\]

\[
\text{SD.MER} = 0.0497
\]

\[
\text{MAX MER} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]

\[
\text{MAX MMR} = 0.25
\]
- MERi = (mmai - mmei)/mmei,
- MMER % = 100/N \[\sum (MERi)] [12] shows MMER had better results than MMRE.
- for n odd:
  MdMER = (MER[n+1]/2)th observation value
- for n even:
  MdMER = [(MER[n]/2) + (MER[(n/2)+1])] /2
- PRED (L%) = K/N * 100, where K is the number of observations where MER is less than or equal to L.
- MER, SDMER, MAXMER

MMRE, MdMRE, and PRED are used here to assess cost estimation model. Table 4 shows all calculations with necessary parameters required to make decision for selecting the estimation model.

6 Commentary on End Results

In this paper the estimation model accuracy checking procedure and parameters were demonstrated, in order to give awareness to the PSEB and registered software houses that are involved in Software Development for out sourcing especially.

COCOMO model when tested on our 5 different modules of same ERP treated as five different projects, it verified the prediction accuracy level PRED (20%) = 70%. MdMER% = 12.14%, MMER% = 17.602%, MAX.MER% = 25%.

Above parameters depict that if we use COCOMO II for medium sized business software then it will give results with in variance 12% min and max 25% i.e. chances of Over or Under estimates 80 to 100% of times and in remaining 20% times may be we will get extreme cases of variance. So we need to quote the price with three cost levels i.e. Optimistic, Pessimistic and Most Likely values keeping in mind the plus minus of 20 to 25 % in this case. Greater the value of MMER/MdMER/MAX.MER, lesser will be the accuracy of Estimation model.

If the model accuracy is not at satisfactory level then the last procedure will come i.e. model calibration for multi-organisation environment in order to support the outsourcing.

7 Future Work

Cross-validation approach gives more realistic accuracy measures that involve dividing the whole data set into multiple train and test sets and calculating the accuracy for each test set, and then aggregating the accuracy across all the test sets. Calculating accuracy in this manner indicates the accuracy of using an external multi-organizational data set for building a cost estimation model, and then testing it on an organization’s projects [8]. Later discussions will also cover the impact of Jack Knife approach in combination with the leave one-outcross validation process and the Hold Out experiments as discussed in COCONUT studies.

Model Calibration steps will be covered in upcoming versions of this research study that will cover the Recalibration process through Linear Regression, Non Linear Regression and Hypothesis of effect and T-Value.

8 References


[13] Lionel C. Briand, Khaled El Emam Katrina D. Maxwell Dagmar Surmann, Isabella Wieczorek, , An Assessment and

A Software Cost Estimation Meta-model for Systematic Reuse Approaches

S. Ben Abdallah Ben Lamine, L. Labeled Jilani, and H. Hajjami Ben Ghezala
Laboratoire Riadi-Gdl, Ecole Nationale des Sciences de l’Informatique
Campus Universitaire la Manouba, La Manouba, 2010, Tunisia

Abstract – PLE, CBSE and COTS-BD are systematic reuse approaches, where reuse is planned at the beginning of a project and is a large scale activity. Therefore, they need important initial investments. Cost estimation environments (models associated with supporting tools) can help the stakeholders to discover the strength of each large scale reuse approach, to better understanding them, to appreciate their promised benefits and choose between them for specific needs and contexts. In this paper, a software cost estimation meta-model is proposed for such approaches, as a result of an investigation showing the similarities between them. The meta-model adopts a four-cycle reuse investment. Variabilities between the approaches necessitate an instantiation of the meta-model for each of them. To quantify estimations, many techniques can be used, depending on the organization’s needs, habits, available data, etc.

Keywords: PLE, CBSE, COTS, cost estimation model.

1 Introduction

Large scale reuse activities promise higher productivity gains, better software quality and time to market. Stakeholders willing to adopt a systematic reuse approach such as CBSE (Component Based Software Engineering), PLE (Product Line Engineering) or CBD (Commercials-Off-The-Shelf Based Development) may wonder which reuse approach to adopt, which investment will be better for the organization context, what is the return on investment, what will happen in the future, develop reusable components or not, buy COTS components or not, etc. Several works propose estimation models or techniques for systematic reuse approaches. But cost estimation is known to be difficult even with traditional software development. It is then obvious that is becomes more complicated with complex reuse approaches, where many new cost drivers should appear. Moreover, the calculations’ precision of such cost models can’t be guaranteed such as in traditional development, particularly because real historical data for systematic reuse is not easily available. Thus, it is reasonable not to wait for the cost models to give you exactly “what will it cost” or “what will be the return on investment” but just to have approximate values and results can help to appreciate the cost drivers’ impact on the projects’ success or failure.

In this paper, we propose a software cost estimation meta-model for systematic reuse approaches. In fact, after a deep investigation on the life cycles of each of the three systematic reuse approaches, the commonalities deduced between PLE, CBSE and CBD let us think about a meta-model for cost estimation that describes the cost components dealt with when using, indifferently, one of these approaches.

Variabilities of the pre-cited systematic reuse approaches make the instantiation of the previous meta-model necessary at the aim to call attention to the specific characteristics of the approaches and their relative impact on costs (and benefits) of the use of an approach or the other (to commit from an approach to an other, from ad-hoc reuse to a systematic reuse approach, etc.). The obtained three instantiated models are encapsulated in a suite called SoCoEMo (Software Cost Estimation Models) containing SoCoEMo-PLE, SoCoEMo-CBSE and SoCoEMo-COTS respectively for PLE, CBSE and CBD approaches. The refinement of this suite is detailed with quantitative formulae. Hence, we have a three leveled modeling approach, with the meta-model as level one, the specific SoCoEMo suite as level two and the refined SoCoEMo suite as level three. With those latter and the environment supporting them, we wish to provide to a systematic reuse approach adopter, some indicators helping to have an idea about its economic impacts (value added), let him decide about strategic choices, calibrate parameters to reach some planned results, etc. The high initial investment can discourage managers, but with an estimation tool, they can better appreciate the real gains inherent to these approaches where reuse is better planned than with ad-hoc reuse.

The meta-model adopts a four-cycle investment approach: the component engineering investment cycle, the domain engineering investment cycle, the application engineering investment cycle and the organizational investment cycle. For each cycle, cost and benefit estimations are done, not just to know about the whole organization return on investment, but also about economic indicators for each cycle.
In the next section, we present the identification of four cycles for systematic reuse approaches denoting similarities in their life cycles. Section three is dedicated to the presentation of the meta-model and section four to its economic equations. We will present in section five only some examples to illustrate the instantiation and refinement phases for the domain engineering cycle. Related works are summarized in section six. We end with conclusions in section seven.

2 Identification of four cycles for systematic reuse approaches

As in [1], we notice that there are four parties in the software systematic reuse process (see figure 1): the organizational management cycle, which has a stake in seeing the reuse program reap benefits for the organization; 2) the domain engineering team, which has a stake in seeing its domain engineering products reused; 3) the application engineering teams, which has a stake in producing applications with low cost, high quality, and short time-to-market; and 4) the component developers, who have a stake in seeing their components reused widely.

In COTS Based Development, the application engineering team can be omitted if we are in the case of COTS-Solution Systems; one substantial product (suite) tailored to provide significant system functionality, 100% COTS (common for administrative systems). In the case of COTS-Intensive Systems, where multiple COTS products, from multiple suppliers, are integrated to collectively provide system functionality; a mix of COTS and custom code (common for operational, safety-critical systems), the component engineering cycle is omitted because all the reusable components are of type COTS (bought components). Hence, the COTS Based Development approach is considered as an instance of a CBSE reuse approach where all the reused components are of type COTS. These COTS components can be reused in several applications of a given organization and so, stored in a reuse library.

In the CBSE approach, the domain engineering team can be: 1) absent or 2) present without a deep domain analysis task or 3) in a better case, present with deep domain analysis which will lead to a significant reduction of customer built components (unique code) in a component based development task. This latter situation can lead to a PLE reuse approach where a set of software-intensive systems can be adopted. Those share a common, managed set of features that satisfy the specific needs of a particular market segment or mission and are developed from a common set of core assets in a prescribed way [2].

Compared to CBSE, PLE is also based on components reuse, but the reusable assets are more complicated because we reuse the reference architecture, software components where commonalities and variabilities are essential and other assets such as documentation.

We will adopt this view of reuse process in our proposed economic meta-model, which can be used to evaluate the ROI of component based software development, COTS based development and product line development. So, our economical model can be used for one of such approaches and/or for using both of them and allowing the decision makers to adopt the one that better satisfies their needs.

3 A Proposed meta-model

In this section, we present the proposed meta-model, denoted SoCoEMo, for Software Cost Estimation Model.

3.1 A Three leveled modeling

The cost estimation modeling logic proposed is a three leveled one (cf. figure 2), owing to the studies done on systematic reuse approaches and the estimation cost models proposed to them in literature.

In fact, similarities between systematic reuse approaches lead us to propose, as level one, a cost estimation meta-model that captures these similarities without focusing on each approach’s characteristics. The meta-model delimits cost components that distinguish between systematic reuse approaches.

Nevertheless, variabilities and specificities of each reuse approach must have specific impact on the cost of adopting the one or the other; consequently, it is necessary to have a specific cost estimation model by approach. This leads us to propose a second level of models which comprises an instance of the meta-model for each systematic reuse approach (PLE, CBSE and CBD). The obtained instances are respectively SoCoEMo-PLE, SoCoEMo-CBSE and SoCoEMo-COTS. At this second level, the specific models of the SoCoEMo Suite do not give any detail of the calculation formulae, they only emphasize cost components that distinguish between systematic reuse approaches.
In the third level, a refinement of the cost components proposed in the second level models is proposed and detailed calculus formulae are given to allow quantitative estimation.

### 3.2 Modeling steps

Modeling steps include four phases:

- Adoption of a four cycle investment reuse organization: the meta-model and the SoCoEMo suite adopt four investment cycles, like the integrated cost estimation model for reuse [3]. These cycles are: the component engineering investment cycle, the domain engineering investment cycle, the application engineering investment cycle and finally the organizational investment cycle. Each investment cycle has to make investment decisions (to invest or not) in a reuse activity, in a way it guarantees not only the benefit of the organization, but also the benefit of the current cycle. The reuse organization of the SoCoEMo meta-model and suite supposes a clear separation between project teams (application production) and domain engineering teams (component and architecture production). The latter ones sell components internally to the project teams.

- Adoption of a cost-benefit calculus based approach for each cycle: whatever the reuse approach is, for each cycle, estimations are done for three cost and benefit factors: IC (initial investment cost), C(y) (periodic cost for years y, where \(SD+1 \leq y \leq Y\)), and B(y) (periodic benefit for years y, where \(SD \leq y \leq Y\)), where Y is the investment period (in years), d is time value of money (discount rate, abstract quantity, between 0.1 and 0.2), and SD is the start date of the investment.

- Determination of cost (or benefit) components for each investment cycle: whatever the reuse approach is, cost components of each cost or benefit factor (IC, C and B) are determined. In the meta-model level those cost components are determined so that they are generic (for any of the systematic reuse approaches). In the instantiated level (two), specific cost components are highlighted for each reuse approach (PLE, CBSE and CBD).

- Quantification of economic functions for each investment cycle. Using cost and benefit factors, economic functions can be calculated, like ROI, NPV, etc. They are indicators for the investment decisions.

### 3.3 Inputs and outputs of the meta-model

Figure 3 outlines the meta-model’s inputs/outputs.

- **Inputs**
  - Level 1: Meta-model
  - Level 2: Specific models
  - Level 3: Specific refined models

- **Outputs**
  - Second intermediate outputs
  - Organizational cycle’s cost components
  - Domain cycle’s cost components
  - Component cycle’s cost components

Inputs of the models proposed are the different cycles’ cost components. Depending on the formulae used in the refined level (level three) those cost components may be fed by more precise and low level inputs (lines of code, function points, object points, etc). Thus each cost component can be estimated by the available means in the organization (a data base of completed projects, experts in the field of the project, other estimation technique or method). The former cost components are used to estimate the cost and benefit factors (IC, C(y), and B(y)) for each cycle and then for the organization. These different cycles feed cost and benefit...
information to each other according to a studied cost cascade. The meta-model organizes the different cost components -inherent to a software development according to one of the systematic reuse approaches- between the reuse investment cycles in an organization. Thus, these cost and benefit factors are the outputs of the meta-model (it is true for the models since they are instantiations).

The latter outputs are used to calculate some economic functions (ROI, NPV, PB, etc.). The obtained values of the economic functions are then the second intermediate outputs of the meta-model. These economic functions are very important because they will give an idea on the worthiness (or not) of an investment.

As mentioned in the introduction, our final goal of implementing such a meta-model and instantiated suite, is to help stakeholders to take some “decisions”. Perhaps, the use of the model has to be supported by other activities? (Decision making is not easy!). That is why, the ultimate importance of the cost estimation modeling given here is not to provide real or precise estimations “what will it cost?” but to help managers to answer questions such as “is it worthy to invest in such reuse approach?”, “what will be the return on investment in a period of x years or months if we buy COTS components?”, “and what if we develop reusable components internally in the organization?”. In fact, “precision” is relative, because cost estimation is “difficult”. The environment implementing the SoCoEMo suite (a user friendly suite of tools) must primarily give a “piloting board” to the decision maker where he can see curves, histograms, compare, calibrate some parameters, change, make a strategic choice, etc. It must be a support for the stakeholder to make a decision “in the right” way.

4 Meta-model’s equations

4.1 Component engineering cycle

Investment cost. Whatever the systematic reuse approach is, the investment cost of the component engineering cycle (IC₇) comprises:

(1) ER: Estimation of development cost of the reusable component for Reuse.
(2) LI: Certification and Library Insertion cost.

Hence, the equation for IC₇ is:

\[ IC₇ = ER + LI \] (1)

Periodic cost. Whatever the systematic reuse approach is, the periodic cost of the component engineering cycle (C₇) comprises:

(1) OC(y): Operating Cost of the library of reusable components.
(2) MN(y): MaiNtenance Cost of the reusable component.

Hence, the equation for C₇ is:

\[ C₇(y) = OC(y) + MN(y) \] (2)

Periodic benefit. Whatever the systematic reuse approach is, the periodic benefit of the component engineering cycle (B₇) comprises:

(1) B₇WB(y): Benefit of selling reusable components as White Box.
(2) B₇BB(y): Benefit of selling reusable components as Black Box.

Hence, the equation for B₇ is:

\[ B₇(y) = B₇WB(y) + B₇BB(y) \] (3)

4.2 Domain engineering cycle

Investment cost. Whatever the systematic reuse approach is, the investment cost of the domain engineering cycle (IC₈) comprises:

(1) DA: Domain Analysis cost.
(2) CompC: Component Cycle cost; costs inherited from component engineering.
(3) COTS: COTS cost; costs resulting from the eventual use of COTS components.

Hence, the equation for IC₈ is:

\[ IC₈ = DA + CompC + COTS \] (4)

Periodic cost. Whatever the systematic reuse approach is, the periodic cost of the domain engineering cycle (C₈) comprises:

(1) AEC: Architecture Evolution Cost.
(2) CompC: Component Cycle cost; costs inherited from the component cycle.
(3) COTS: COTS cost; Cost of buying a new COTS or a new release of an old COTS.

Hence, the equation for C₈ is:

\[ C₈(y) = AEC(y) + CompC(y) + COTS(y) \] (5)

Periodic benefit. Whatever the systematic reuse approach is, the periodic benefit of the domain engineering cycle (B₈) comprises:

(1) BComp: Benefits inherited from component cycle.
(2) BCOTS: Benefits of using COTS components.

Hence, the equation for B₈ is:
\[ B_\alpha(y) = B_{Comp}(y) + BCOTS(y) \]  

(6)

4.3 Application engineering cycle

**Investment cost.** Whatever the systematic reuse approach is, the investment cost of the application engineering cycle \((IC_\alpha)\) comprises:

(1) \(Cic\): cost of internal components (from component cycle).
(2) \(Cuc\): cost of unique components.
(3) \(Ccots\): cost of COTS components.

Hence, the equation for \(IC_\alpha\) is:

\[ IC_\alpha = Cic + Cuc + Ccots \]  

(7)

**Periodic cost.** Whatever the systematic reuse approach is, the periodic cost of the application engineering cycle \((C_\alpha)\) comprises:

(1) \(Evolci\): Evolution impact of internal reusable components on the application cycle; maintaining reusable components in the component cycle has an effect on applications using those components.
(2) \(Evolcots\): Evolution impact of COTS components on the application cycle; maintaining COTS components (which is done by its vendor) has a cost which is captured by the cost of a new release cost in the domain engineering cycle. This COTS evolution has an impact on the application engineering cycle. This phenomenon is known as volatility in literature.
(3) \(MNuc\): cost of maintaining unique components.

Hence, the equation for \(C_\alpha\) is:

\[ C_\alpha(y) = Evolci(y) + Evolcots(y) + MNuc(y) \]  

(8)

**Periodic benefit.** Whatever the systematic reuse approach is, the periodic benefit of the application engineering cycle \((B_\alpha)\) comprises:

(1) \(Bci\): benefits of using internal reusable components.
(2) \(Bcots\): benefits of using COTS components.

Hence, the equation for \(B_\alpha\) is:

\[ B_\alpha(y) = B_{ci}(y) + B_{cots}(y) \]  

(9)

4.4 Organizational Cycle

**Investment cost.** Whatever the systematic reuse approach is, the investment cost of the organizational cycle \((IC_\rho)\) comprises:

(1) \(INF\): cost of reuse infrastructure (personnel training, etc.).
(2) \(DomC\): costs inherited from domain engineering cycle(s) initiated in year \(SD\).
(3) \(AppC\): costs inherited from application engineering cycle(s) initiated in year \(SD\).

Hence, the equation for \(IC_\rho\) is:

\[ IC_\rho = INF + DomC(SD) + AppC(SD) \]  

(10)

**Periodic cost.** Whatever the systematic reuse approach is, the periodic cost of the organizational cycle \((C_\rho)\) comprises:

(1) \(DomC\): periodic costs inherited from domain engineering cycle(s),
(2) \(AppC\): periodic costs inherited from application engineering cycle(s).

Hence, the equation for \(C_\rho\) is:

\[ C_\rho(y) = DomC(y) + AppC(y) \]  

(11)

**Periodic benefit.** Whatever the systematic reuse approach is, the periodic benefit of the organizational cycle \((B_\rho)\) comprises:

(1) \(Bdom\): periodic benefits inherited from domain engineering cycle(s),
(2) \(Bapp\): periodic benefits inherited from application engineering cycle(s).

Hence, the equation for \(B_\rho\) is:

\[ B_\rho(y) = B_{dom}(y) + B_{app}(y) \]  

(12)

5 Meta-model instantiation and refinement

As outlined in figure 2, after instantiations of the meta-model of level one, we obtain a suite of models, SoCoEMo, containing three cost estimation models specific to three systematic reuse approaches, SoCoEMo-PLE, SoCoEMo-CBSE and SoCoEMo-COTS, respectively for PLE, CBSE and CBD.

However, for the sake of brevity, we will present here just an example of instantiation and refinement (levels 2 and 3) for the periodic costs of the domain engineering cycle.

5.1 Instantiation example: level 2

The instantiation of the periodic cost’s equation of the domain engineering cycle (cf. equation (5)), for SoCoEMo-PLE, SoCoEMo-CBSE and SoCoEMo-COTS models are given respectively by:
the well known and widely used COCOMO suite.

obtained model. That is why, in the refinement phase of our
take metric formulae from it) is, the better it is for the new
course, the more validated and widely used the model (to
estimation models to formulate its proper equations. Of
model that allows the use of metrics taken from other cost

For example, the refinement equation that we propose
for IdSel, that captures the COTS component identification
and selection cost, is the equation of the assessment sub-
model of the COCOTS model [5]. If no metrics are given in
literature for a cost component like AEC, the organization
can use the expert judgment estimation method.

6 Related works

When comparing cost models for systematic reuse
approaches [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16], we
found them very different concerning their level of detail of
formulae (from very sketchy to very detailed and
comprehensive) and their view to cost estimation (from a
very broad point of view to a very specific and pointed
(metrics oriented) point of view).

In some approaches, the modeling logic is based on a
global and high level point of view of the concerned reuse
approach (life cycle steps, market strategic view,
specificities of the reuse approach they deal with, etc.).
Other models take interest in low level cost factors that are
in relation with the developer, its experience, the number of
lines of codes or function points he develops (software size
metrics), the development environment, etc. They use
classical estimation techniques [17]. Some models are of
very low level, because in addition to the characteristics
offered by the former ones, they take interest to particular
types of applications (Web applications, security
applications, etc.), to particular development techniques (4
GL, etc.). Finally, hybrid Models are a mixture of high level
and low or very low models. In fact, these models have two
points of view, different but complementary, to cost
estimation; the first is a systemic view, which is a generic
view on the reuse approach adopted, strategic choices, life
cycle, market tendency, etc; the second point of view
considers details close to development, they give more
importance to measurements, data of completed projects,
etc.

The meta-model proposed in this paper gives a general
modeling of cost estimation for software developed using
one of the systematic reuse approaches. In addition, it has a
systemic view of reuse, since it adopts a four cycle reuse
organization (component engineering cycle, domain
engineering cycle, application engineering cycle and
organizational cycle); it proposes then different points of
view on the reuse practice in an organization. The
SoCoEMo suite (level two), comprising three specific cost
estimation models, is still like the meta-model; it does not
give any detailed formulae. Concerning the models of level
tree, they incorporate detailed calculus formulae that can
be of low or very very low level depending on the needs and
context of the users.

7 Conclusions

The final objective of this work was to propose a
meta-model for software cost estimation dedicated to
systematic reuse approaches (PLE, CBSE and CBD). In fact, our deep study of the latter approaches as well as the cost models specific to them, led us to propose a meta-model that gives a general view on their deduced commonalities, but also variabilities. Those specific variabilities are emphasized in a second and third level of modeling, where specific characteristics of each systematic reuse approach are highlighted. In fact, in level two, no equation details are given; the aim is to identify specific cost components for reuse approaches. It is in level three where some propositions of cost details are given through the use of adequate equations proposed in validated and known models in literature. An organization aiming to adopt a large-scale reuse approach to develop software can make use of these three levels of models, according to its specific needs and context, to have an increasing level of “precise” estimation from level one to level three of our modeling approach. Our future work focuses on the validation of the meta-model.

8 References


Abstract - A safety-critical system is one whose incorrect function may have very serious consequences such as loss of human life, severe injuries, large-scale environmental damage, or considerable economical penalties. Hence software for safety-critical systems must deal with the hazards identified by safety analysis in order to make the system safe, risk-free and fail-safe. Software safety is a composite of many factors. Existing software quality models like McCall’s, Boehm’s and ISO 9126 are inadequate in addressing the software safety issues of real time safety-critical embedded systems. At present there does not exist any standard framework that comprehensively addresses the factors, criteria and metrics (FCM) approach of the quality models in respect of software safety. This paper proposes a new model for software safety based on the McCall’s software quality model that specifically identifies the criteria corresponding to software safety in safety critical applications. This model is then applied to a prototype safety-critical system viz. a software–based Road Traffic Control System (RTCS) commonly used in city traffic, to validate its utility.

Keywords: safety-critical system, software safety, software quality

1 Introduction

A safety-critical system is one that has the potential to cause accidents. Software is hazardous if it can cause a hazard i.e. cause other components to become hazardous or if it is used to control a hazard. Software is deemed safe if it is impossible or at least highly unlikely that the software could ever produce an output that would cause a catastrophic event for the system that the software controls. If it is used to control a hazard i.e. cause other components to become hazardous or if it is used to control a hazard. Software is deemed safe if it is impossible or at least highly unlikely that the software could ever produce an output that would cause a catastrophic event for the system that the software controls. It is impossible or at least highly unlikely that the software could ever produce an output that would cause a catastrophic event for the system that the software controls. Hence, a hazard is a potentially dangerous situation. Safety constraint. A hazard characterizes a system state that for safety reasons should not occur. If this is negated and some safety margins are included we get a safety constraint, i.e. a description of a property that the system should possess in order to be safe. The NASA Technical Standard on Software Safety [7] gives the following definitions. Safety-Critical. Those software operations that, if not performed, performed out-of-sequence, or performed incorrectly could result in improper control functions (or lack of control functions required for proper system operation) that could directly or indirectly cause or allow a hazardous condition to exist. Safety-Critical Software. Software that: (1) exercises direct command and control over the condition or state of hardware components; and, if not performed, performed out-of-sequence, or performed...
incorrectly could result in improper control functions (or lack of control functions required for proper system operation), which could cause a hazard or allow a hazardous condition to exist. (2) monitors the state of hardware components; and, if not performed, performed out-of-sequence, or performed incorrectly could provide data that results in erroneous decisions by human operators or companion systems that could cause a hazard or allow a hazardous condition to exist. (3) exercises direct command and control over the condition or state of hardware components; and, if performed inadvertently, out-of-sequence, or if not performed, could, in conjunction with other human, hardware, or environmental failure, cause a hazard or allow a hazardous condition to exist.

This paper is organized as follows: section 2 reviews software quality models, section 3 presents the proposed software safety model, section 4 describes the application of the framework to a laboratory prototype safety critical system and the final section gives a summary of the work done and conclusion.

2 Software quality models

There have been two notable models of software quality attributes viz. McCall’s and Boehm’s. There are others but these two illustrate the general purpose quality models. Both McCall and Boehm have described quality using a decompositional approach [8,9]. McCall's model of software quality (The GE Model, 1977) incorporates 11 criteria encompassing product operation, product revision, and product transition. Boehm's model (1978) is based on a wider range of characteristics and incorporates 19 criteria.[10]. The criteria in these models are not independent; they interact with each other and often cause conflict, especially when software providers try to incorporate them into the software development process. ISO 9126 incorporates six quality goals, each goal having a large number of attributes [11].

2.1 McCall software quality model

This framework is useful for its integrated approach to quality. In the framework, software quality attributes are classified into a hierarchy of three levels as shown in Figure 1. At the top level are the so-called “quality factors” from a customer or user perspective: correctness, reliability, efficiency, integrity, usability, maintainability, testability, flexibility, portability, reusability, and interoperability. At the second level, are the “quality criteria,” which represent technical concepts. At the third level, are the “quality metrics,” which measure the attributes of software products. The last two levels are from engineering perspectives. McCall suggests the following application steps:

- Deduce quality factors based on the characteristics of the system;
- Trade-off and prioritize the quality factors based on the needs of the customers/users;
- Deduce related quality criteria and metrics using the framework; and
- Base specification, design, coding, and testing on the deduced factors, criteria, and metrics

The original eleven quality factors in McCall Software Quality Model are: Usability, Integrity, Efficiency, Correctness, Reliability, Maintainability, Testability, Flexibility, Reusability, Portability, Interoperability.

2.2 Modified McCall’s framework applied to software safety

Raghu Singh has proposed a modified framework to address software safety [12]. The four factors relating to software safety in his model which are part of the original McCall model are: Correctness, Efficiency, Reliability, Testability.

<table>
<thead>
<tr>
<th>Table 1. Factors and Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FACTORS</strong></td>
</tr>
<tr>
<td>Correctness</td>
</tr>
<tr>
<td>Efficiency</td>
</tr>
<tr>
<td>Reliability</td>
</tr>
<tr>
<td>Responsiveness</td>
</tr>
<tr>
<td>Testability</td>
</tr>
</tbody>
</table>
To these four quality factors, a new factor – responsiveness was introduced to account for the real time performance. For each factor the corresponding criteria (attributes from the developer point of view) are derived as shown in Table 1. It is argued that determination and application of specification, design, coding, and testing methods in a project should be based on the metrics derived from the criteria in order to "ensure" software safety.

All these quality models – McCall’s, Boehm’s and ISO 9126 and the modified model by Raghu Singh do not directly address the specific issues of software safety but emphasize the general quality attributes. They have the following limitations. First, many of the factors suggested by these models are not directly related to the specific issue of hazards contributed by the malfunction modes of software. Second, they assume that the concepts of reliability and safety are equivalent whereas a system can be reliable and still be not safe. Making a system more reliable is not sufficient if it has unsafe functions. This translates to having a system that reliably functions to cause unsafe conditions. Finally, these models seem to focus on non-safety critical systems where the emphasis is more on efficiency and other quality attributes and less on the safety issues of hazards and mishaps that can endanger human life and property. To overcome these limitations, a new model is proposed that captures the major issues specifically related to software safety.

3 Proposed model for software safety

The proposed model for software safety based on the factor, criteria and metric approach is shown in Figure 2.

The quality factor software safety may be decomposed into six quality criteria – System Hazard Analysis, Completeness of Requirements, Identification of Safety Critical Requirements, Design based on Safety Constraints, Run-time issues management, and Safety Critical testing. Each criteria may be further decomposed into a set of lower level quality metrics, which are directly measurable. Each proposed criteria of software safety is briefly explained as follows.

3.1 System hazard analysis

While modeling software safety is the focus of this paper it is important to note that no software works in isolation. The entire system must be designed to be safe. The system contains the software, hardware, the users, and the environment. All must be given consideration when developing software. All parts of the system must be safe. Functional and operational safety starts at the system level. Safety cannot be assured if efforts are focused only on software. The software can be totally free of 'bugs' and employ numerous safety features, yet the equipment can be unsafe because of how the software and all the other parts interact in the system.

Preliminary system safety analyses (e.g., Preliminary Hazard Analysis (PHA), conducted during the system requirements phase when the role of software is being defined, begin to identify the hazards associated with a particular design concept and/or operation. These preliminary analyses and subsequent system and software safety analyses identify when software is a potential cause of a hazard or will be used to support the control of a hazard. This software shall be classified as safety-critical and shall be subjected to software safety analysis.

The system safety analyses are the first place to identify software safety requirements necessary to support the development of the software requirements specification. These requirements shall be provided to the developer for inclusion into the software requirements document. Some examples of software safety requirements include limits (e.g., redlines, boundary values), sequence of events, timing constraints, interrelationship of limits, voting logic, hazardous hardware failure recognition, failure tolerance, caution and warning interfaces, hazardous commands, etc.

The system safety analyses continue throughout the project life cycle. The software safety analysis process needs to continue to review the results of the systems analyses to assure that changes and findings at the system level are incorporated into the software as necessary. In addition, the software safety analyses provide input to the system safety analyses. The software safety analyses are a special portion of the overall system safety analyses and are not conducted in isolation.
In essence there are four safety-relevant parts of a system development process: 1) identifying hazards and associated safety requirements 2) designing the system to meet its safety requirements 3) analysing the system to show that it meets its safety requirements 4) demonstrating the safety of the system by producing a safety case [13].

3.2 Completeness of requirements

Completeness can be defined as the property that requirements are sufficient to distinguish the desired behavior of the program from that of any other undesired program that might be designed. It should not be surprising then that most errors found in operational software can be traced to requirements flaws, particularly incompleteness [14]. Completeness is a quality often associated with requirements but rarely defined. In addition, nearly all the serious accidents in which software has been involved in the past 20 years can be traced to requirements flaws, not coding errors. The software may reflect incomplete or wrong assumptions about the operation of the system components being controlled by the software or about the operation of the computer itself. The problems may also stem from unhandled controlled-system states and environmental conditions. Thus simply trying to get the software "correct" in terms of accurately implementing the requirements will not make it safer in most cases. Basically the problems stem from the software doing what the software engineer thought it should do when that is not what the original design engineer wanted. Integrated product teams and other project management schemes to help with this communication are being used, but the problem has not been solved [12].

3.3 Identification of software-related safety-critical requirements

A safety critical software requirement may be understood as a software requirement identified as essential to the safe system operation or use [15]. Specifically, a safety critical software requirement performs one or more of the following functions:

- Controls or directly influences the functioning of safety critical hardware
- Controls or directly influences hazardous systems
- Monitors the state of the system for purposes of ensuring its safety
- Senses hazards and/or displays information concerning the protection of the system
- Handles or responds to fault detection priorities
- Disables or enables interrupt processing software
- Generates output that displays the status of safety critical hardware
- Computes safety critical data

The above listed functions are based on the functions presented in STANAG 4404 [16]. Based on the interaction among safety-critical, safety-related, and non-safety-critical components each module of the software-based system is classified into one of four criticality levels:

C3—Safety-critical: a module where a single deviation from the specification may cause the system to fail dangerously

C2—Safety-related: a module where a single deviation from the specification cannot cause the system to fail dangerously, but in combination with the failure of a second module could cause a dangerous fault

C1—Interference-free: a module that is not safety critical or safety related, but has interfaces with such modules

C0—Not Safety-related: a module that has no interfaces to safety-related or safety-critical modules

At the simplest level, any module that is directly used in implementing a safety requirement would be C3, and any safety integrity requirements would be C2.

3.4 Design based on safety-constraints

The first step in the safety-constraint centered design approach is the specification of safety constraints [17]. In hardware systems, redundancy and diversity are the most common ways to reduce hazards. Hardware detection and control includes mechanisms such as fail-safe designs, self-tests, exception handling, warnings to operators or users, and reconfigurations. For software intensive safety-critical systems, software design must enforce safety constraints. Reviewers should be able to trace from requirements to code and vice versa. In addition to the specific safety constraints developed for the system being designed, the design should incorporate basic safety design principles. Safety, like any quality, must be built into the system design. Software represents or is the system design [18]. The most effective way to ensure that a system will operate safely is to build safety in from the start, which means that system operation must not lead to a violation of the constraints on safe operation.

System accidents result from interactions among components that lead to a violation of these constraints -- in other words, from a lack of appropriate enforcement of constraints on the interactions. Because software often acts as a controller in complex systems, it embodies or enforces the constraints by controlling the components and their interactions. Software, then, can contribute to an accident by not enforcing the appropriate constraints on behavior or by commanding behavior that violates the constraints. The requirement for software to be safe is not that it never “fails” but that it does not cause or contribute to a violation...
of any of the system constraints on safe behavior. This observation leads to the suggested approach to handling software in safety-critical systems, i.e., first identify the constraints on safe system behavior and then design the software to enforce those constraints.

3.5 Run-time issues management

An operating-system kernel and application programming interface often perform the most important role in a safety-critical system. Exception handling, deadlocks, process and stack management, scheduling and flow control, and memory protection all have repercussions on the safety function and can be key elements of meeting safety-integrity requirements. Figure 3 shows the decomposition of the run-time issues criteria into five sub-criteria or lower-level criteria which provide a basis for measurements.

3.6 Safety critical testing

Safety critical software functions provide the source of requirements to be tested. Testing shall be performed to verify correct incorporation of software safety requirements. Testing must show that hazards have been eliminated or controlled to an acceptable level of risk. Additional hazardous states identified during testing shall undergo complete analysis prior to software delivery or use. Software safety testing of Safety-Critical Computer Software Components (SCCSC) shall be included in the integration and acceptance tests. Acceptance testing shall verify correct operation of the SCCSCs in conjunction with system hardware and operators [19]. It shall verify correct operation during stress conditions and in the presence of system faults. It is important to tailor the safety-critical testing effort to emphasize the parts of the software that need additional analysis and testing. The greatest effort must be placed on the hazards posing the highest risk. We consider it adequate to divide the software into two risk groups for test purposes. Group one includes hazards that are catastrophic or critical. Group two includes hazards that are marginal or negligible per the definitions in MIL-STD-882C. Software in the first group deserves special safety analysis and testing since the hazards pose a higher level of risk. The normal level of software analysis and testing performed for operational software is adequate for group two.

4 Application of safety model to road traffic control system (RTCS)

The proposed safety model based on the factor-criteria-metrics quality model is applied to RTCS to validate its utility. RTCS is a prototype safety-critical application of limited complexity. A modern traffic signal system consists of many components, all of which must operate reliably in order for the system as a whole to operate smoothly and efficiently. Figure 4 shows the functional block diagram of RTCS [20]. It consists of the following main components: micro controller-based Traffic Controller, User Interface/Display, Driver Circuitry, Sensors, and Signal Heads.

4.1 Components of RTCS

Traffic Signal Controller: The traffic signal controller is the means of changing the signal indications and consists of computer controls that operate selection and timing of traffic movements in accordance to pre-programmed sequences and vehicle detection. Traffic signal software is stored in the memory of the microcontroller. User Interface/Display: This is used to operate the selections of the microcontroller. Driver Circuitry: Communicates the controller information to the signal heads and activates them. Signal Heads: A signal head consists of one or more signal faces that can include solid red, yellow or green lights. S1, S2, S3, S4 are the signal heads and are represented by small circles in figure 4. Sensors: These are used to detect the working status of the signal lights at each of the traffic signals S1, S2, S3, and S4. Information from
each of the signals is passed to the traffic controller through the sensors. The city 4-way traffic intersection is shown as a bi-directional X at the upper left-hand corner.

4.2 Normal operation of RTCS

When RTCS is first switched on, it does a preliminary check of the normal working status of all the subsystems involved – the driver circuitry, the user interface, the sensors and the signal heads. If all the components are found to be in normal working condition, it executes the code related to normal operation of the traffic signal. Based on the pre-programmed requirements and user selections, RTCS continuously cycles through the signals based on a fixed-time mode of operation. If RTCS detects any abnormal situation or state during its normal mode of operation, perhaps due to an unexpected lightning strike or rainstorm that disrupts the circuitry of the signal heads, it executes the code relating to emergency situation wherein the working signal heads simultaneously flash red signal. This is an indicator to the public that the traffic signal is not in working condition and that they need to take necessary precaution in crossing the intersection.

All the six criteria of the safety model were applied to RTCS. First, the system-level hazard analysis was done to identify hazardous failure conditions at the system level. Second, completeness of requirements criteria was applied to check any missing or ambiguous specifications. Third, all the safety-critical and non-safety critical requirements were identified. Fourth, a design that enforced the safety constraints was chosen for RTCS and the implementation was done in C language using a standard compiler. Fifth, run-time performance was monitored for problems relating to exceptions, deadlocks, memory related issues etc. Lastly, safety critical testing of RTCS was done by separating the code into two risk groups. Group one includes hazards that are catastrophic or critical. Group two includes hazards that are marginal or negligible. The preliminary results in applying the safety model in developing the safety-critical RTCS are satisfactory. The next phase of this work involves applying this safety framework to a prototype railroad crossing control system.

5 Conclusion

This paper discussed the criteria relevant to software safety. An approach for modeling software safety was proposed. A set of quality criteria that form the basis of software safety was presented. The proposed model is applied to a laboratory prototype model of a 4-road traffic control system that includes safety-critical operations and observed satisfactory results. Work is in progress to apply this framework to a software safety based railroad crossing control system. Using the experimental results of the proposed framework with road traffic control system and a railroad crossing control system, work can be extended to address issues of development cost and development time in implementing this framework to achieve software safety metrics. Rigorous work is needed to meet the complete requirements of software safety aspects that leads to standardization of framework with safety metrics.

6 References


SESSION

PRODUCT MANAGEMENT

Chair(s)

TBA
Feasibility of User-Driven Development in a Corporate Environment

Tryggvi Björgvinsson, Helgi Thorbergsson
Department of Computer Science, University of Iceland, Reykjavik, Iceland

Abstract - A new software development methodology, called User-Driven Development, for custom or bespoke software development is proposed in this paper. The goal of this new development methodology is to incorporate users of software as the decision maker and main driving force in the software development process, without changing in-house methods of software developers. An important sub-goal of the methodology is not to introduce any new methods for software development but to re-use existing methods and techniques.

Efforts are mostly focused on identifying possible issues which might arise and discuss how existing solutions can be used to ease integration of the methodology into existing software development environments. The paper also proposes an abstract model of a possible hierarchy which can be used in user and developer firms when incorporating User-Driven Development.

The research examines, from a theoretical point of view, the feasibility of the methodology in a corporate environment. The main research question this paper tries to answer is whether this is a viable methodology, and what is needed of users and developers, respectively. After introducing User-Driven Development the paper looks how it affects some important aspects of software development.

Keywords: User involvement, Software development methodologies, User-Driven Development

1 Introduction

Various development methods exist which try to increase user involvement in the process. Participatory design (or cooperative design) is a software development method for software design[15] which falls under user-centred design[20] and puts focus on user participation, the agile manifesto which underlines the importance of a close relationship between users (business) and developers[1], and the non-software oriented community-driven development of The World Bank[19] are just a few examples of the recognitions of user participation in development processes. User involvement is also further emphasised in Keil and Carmen’s paper on customer-developer links[14], where they show that the more direct links customers and developers share, the more likely it is that the project will be successful.

Eric von Hippel gives unwavering arguments, in his book Democratizing Innovation, for how a high number of users, what he calls lead users, are more capable of innovating than manufacturers[25]. In fact he shows, by quoting a study by himself and Riggs[22], that users are better at developing new products but manufacturers are better at improving existing products. Von Hippel explains that the capability of users, rather than manufacturers, to innovate is correlated with how sticky information is, i.e. the amount of effort required to transfer the information.

Cockburn and Highsmith claim that having user experts on board in development gives developers rapid feedback on any implications and prevents that a misunderstanding accumulates as the work progresses[5]. Ives and Olson call it “common wisdom” that user involvement leads to a higher chance of success[13] and Kujala adds to the benefits by showing that user involvement also leads to increased user satisfaction[17]. Furthermore, the ISO 13407 standard on human-centred design processes for interactive systems, recommends user involvement for better understanding of the user and the task[12]. All in all, it is inevitable that the user must be involved, to a high degree, in the software development process.

User-Driven Development is a fresh methodology which is aims to make the user the most important developer, i.e. the driving force and leader of the project. The methodology can be seen as an organisational supplement to common developing methods and is meant to incorporate as many existing methods and techniques for user involvement as possible, but still keep everything under control. The fundamental ideas of User-Driven Development is explained in this paper and possible problematic issues concerning software development discussed. The main goal of the research is to identify existing methods and techniques which can possibly be used to circumvent any issues and would aid User-Driven Development.
2 User-Driven Development

2.1 Definition

Before continuing it is necessary to define User-Driven Development which serves as a foundation for the theoretical feasibility research of this paper. Since User-Driven Development lets the user control the software development it is best to define the methodology from the perspective of the user's role and how the users actions affect software developers.

**Definition.** User-Driven Development is an organisational methodology, for custom or bespoke software development, where the user is made responsible for: (1) Achieving the project’s goals, (2) motivating developers, (3) being available in all areas of the development, (4) making correct and justified decisions, (5) resolving conflicts, and (6) managing stakeholders.

To put it in a simpler way, User-Driven Development makes the user the overall project leader. As has been mentioned above the development method is aimed at development of custom or bespoke software, which von Hippel shows that many users want and are willing to pay for[25]. Because the methodology is only focused on custom or bespoke software development it is relatively easy to define who users are, and who developers are. Grudin’s definition can be used, where users are the people directly engaged with the system or end users and developers are active members of a development project [9]. Grudin’s definition for developers is a narrow one, and excludes those in management and support roles, thus serving User-Driven Development perfectly since it tries to move the software development management from developers to the users.

It is important to understand that it is perhaps not effective that all users serve as project leaders but a representative of the users (or a small group of representatives) should take on the role of project leader and motivator. Another aspect which is essential to User-Driven Development and important to understand is that developers do not sell the user a custom or bespoke software solution, instead the provide developmental services. These services include, for example, software coding, software testing, security implementation, database design, user interface design. It is then up to the user to identify the services needed, choose a provider for each service (there can be many competing providers for each service), approve the work of the service providers, and tie everything together into a complete product which fulfills the user’s needs. This arrangement is shown in figure 1, where one service provider out of many potential providers gets a piece accepted by the user into the project.

![Figure 1. Arrangement of service providers and the user in User-Driven Development](image)

2.2 Successful Realisation

A well known and established software development model which one can categorise under User-Driven Development is open source software development. It is commonly accepted that developers of open source software are always users of the software[8]. The project leader is generally the founder of the project[7], that is the initial user who wanted a custom or bespoke software solution to fulfil some specific need[21].

In open source software development, users collaboratively develop the software by sharing ideas, patches, and opinions publicly. This method often results in a solution which suits the needs of many different users, something that proprietary software developers would need extensive research to be able to achieve. Many open source software projects are widely recognised to be developed rapidly, but still maintain high quality.

Many open source software projects can also be categorised as free software projects, although exceptions do exist. The two approaches are very similar in legal terms, but differ in the underlying philosophy: Free software is an ideological approach to give users essential freedoms when using the software (that is free software refers to freedom not price), while open source software is a pragmatic approach, i.e. it recommends a method to develop software (usually free software) which results in rapidly developed, high quality software. The freedoms users of free software have are four: The freedom to use the software, study and adapt the software, modify the software, and distribute the software and the modifications[24]. Free software developed using the open source software development method is usually called free and open source software and abbreviated FOSS.

It is straightforward that the four freedoms of free soft-
ware make User-Driven Development possible by giving the legal right over the work to the user. Additionally free software helps to avoid a situation where a single service provider will be able to monopolise the service market and the user as well. Therefore it is highly recommended that the software being developed is free software.

3 Corporate Environment

The corporate environment consists of four sub-environments: The social, political-legal, economic, and technological environment. Managers must constantly try to cope with an ever changing environment where major sources of change include globalisation and product quality[16].

It is necessary to understand how User-Driven Development works in these various environments and how the methodology might help coping with new changes. The following two sections consider the impact on the corporate environment from the perspective of a user organisation, and a service provider or development organisation. The major sources of change are not discussed in detail since they focus mostly on the change in solutions rather than methodology. However if User-Driven Development will have a similar effect on the software as the open source software development method, the two sources of change mentioned above, i.e. globalisation and product quality, might lead to corporations picking up User-Driven Development. This effect must first be backed up with more research and experiments, before acting on the theory.

3.1 User Side

The biggest impact of User-Driven Development, considering the social environment of user corporations, is that it should help them manage diversity and avoid discrimination effectively. Instead of forcing upon users a solution which suits the majority, the methodology encourages active user-involvement in all steps of the development process. This approach is similar to participatory design which has as an objective to get users to participate in the development of a product which helps them performing their job[15].

It might become problematic in some countries that those who “work for hire” can retain their copyright and thus threaten with lawsuits and so forth. If the product is free software any legal problems of getting different service providers to work together will be reduced. Licensing under well known free and open source software licenses will avoid a few legal issues which might come up. External politics will usually not be affected much by a User-Driven Development project, however internal politics of user firms might need a change to cope with a new minuscule democracy which probably emerges in software project decisions, replacing a segment of the hierarchical structure by building a network of equals.

The economical environment will very likely be highly affected by User-Driven Development. More effort will be required by the user, but instead hopefully it will lead to a more suitable solution for the users, delivered on schedule and cheaper. What might become exceptionally tricky will be the management of the different services and paying different contributing providers. The economic environment must be carefully planned when taking up a User-Driven Development method.

Technologically the user gains a better product which many users will find more suitable to their needs. Furthermore, given the correct employment of service providers, it is quite likely that the product will also be technically more capable, e.g. hiring a security implementation provider will probably boost up the security of the system manifold.

The user also benefits from using service providers, rather than solution providers. An important side effect to the democratic service provision, i.e. many service providers pool different services to the same project, is that it forces service providers to explain their work so that others can build upon it. This means that the knowledge of an authoring developer remains available to everybody if that developer leaves the company. In addition, the particular developer might be able to continue working on the same software after switching his or her employers.

3.2 Developer Side

The biggest change on the developer side will most likely be the change from being a solutions developer to being a service provider. Consulting companies will not not be as affected as regular developers which develop complete solutions for customers. Of the sub-environments of the corporate environment it is therefore probably the economic environment which will change the most. Selling a particular service will be the focal point which will lead to better cost analysis, i.e. estimating cost of a specific service is easier than estimating it for a complete project. Service providers will very likely be more willing to take some risks which will be beneficial to the project in whole.

The technological environment and the social environment will change hand in hand in User-Driven Development. The technological focus of developers will probably move from complete solutions to a specific field and specialty. As a result, software development might split up into more specialised areas. Individual developers will have a better ability to choose a field which they are very interested in and have a better chance to work with something they find passionate, benefitting the project as a whole. This
is, of course something which will probably happen gradually and only if users decide to use User-Driven Development when needing custom or bespoke software, which in turn relies on that User-Driven Development is beneficial for them.

Effects on the political-legal environment will probably be more legal awareness, especially if free software is used as a foundation. Service providers will be on the lookout for copyright violations in the form of license violations, which will benefit the users as well as themselves, i.e. there will not be a need to prevent copyright violations through technical implementations. Internal politics of development departments will not change much, but service providers must watch out that other providers do not try to externally use politics for their benefit by, for example, trying to ensure that users should only do business with them and go against the democratic ideas of User-Driven Development.

4 Abstract Model

The model in figure 2 is an example of a hierarchy structure for a User-Driven Development method which tries to affect existing development processes of service providers as little as possible and be as unobtrusive with regard to normal business procedures of the users. Another example is the open source software development method which can be categorised under User-Driven Development and would not be modelled this way.

Users who need a custom or bespoke software solution will put up an internal discussion forum, the private user archive, where affected users within the user firm can submit ideas for requirements, give their opinions on or build upon existing ideas, and vote for or prioritise polished requirements. This way, every interested and affected end-user can have a saying in the establishment of the requirements. However, to prevent users from going ballistic with the requirements specification a user manager is needed to control the process and limits due to time constraints or budgets, i.e. the manager acts as a filter. This manager can be an external consultant working as a service provider. The manager forwards accepted requirements to a external service portal, the public developer archive, and disapproved requirements should always be supported by strong reasoning for the disapproval, possibly set by a company policy, to maintain the user democracy.

The public developer archive will probably be tricky to set up so that possible service providers will be able to participate in the development. However, the infrastructure to achieve this already exists in open source projects. For example, Freshmeat\(^1\) maintains a large index for software and reports updates on projects and web portals such as Sourceforge\(^2\) provide tools for management of collaborative development. Therefore, this can be solved by setting up an electronic market square similar to the Icelandic government\(^3\) or a less formal approach using mailing lists or social networks. The most appropriate solutions must be chosen to suit both users and developers, but a general solution is preferred to avoid confusion for service providers.

---

\(^1\)http://www.freshmeat.net
\(^2\)http://www.sourceforge.net

Figure 2. Model of a possible User-Driven Development hierarchy
more, each project or user firm should have its own public developer archive to prevent any misunderstanding, conflicts or confusion.

The public developer archive serves as a central point for the development of the project. Service providers submit their work on each requirement, criticise other providers’ work (diplomatically and with strong reasoning), and ask questions or post comments on requirements. Every service provider is free to submit work on as many requirements they see fit but should be aware that any other provider is able to criticise their work. In the end it is the user manager which decides what gets approved into the project, based on arguments and through acceptance tests. This should encourage developers to work rapidly but still efficiently which should result in high quality custom or bespoke software, developed rapidly and free of any lock-ins to a specific developer.

5 Development Issues

This section goes through the most well known activities of software development and from the perspective of User-Driven Development tries to find well known, existing methods which can be applied to each activity. These methods are not the only ones which can be used, but can be seen as a possible solution which would comply with User-Driven Development, thus, in the end indicating whether User-Driven Development is feasible and can be achieved easily.

Since flow of information is very important to User-Driven Development, information flow is mentioned, specifically. A small model which can be used when defining how information flows in a specific project is explained to show how users and developers can possibly decide upon the correct information flow or methods.

5.1 Software Development Activities

5.1.1 Requirements

Moving to free and open source software is probably recommended in the long run for user firms that expect high benefits from customising software. These users are what Eric von Hippel calls lead users, i.e. users who are ahead of the market and inclined to innovate[25]. User-Driven Development should make it possible for lead users to innovate without implementing, that is to specify their innovation ideas as requirements and get service providers to implement it for them. Using free and open source software means that specifying requirements should be relatively easy since it would only mean explaining modifications to existing, easily accessible, software developed by others, that gives users the freedom to adapt and modify it.

It is of course possible that free software, which can be built upon, does not exist or that existing software is not mature enough for stable use. However it should still be worth it to demand that the product be free software because User-Driven Development flows more naturally around free software projects. Perhaps a few user firms could pool together in order to make the free software mature enough for actual use. The mere existence of a non-stable version of a free software product indicates that there is a need from other users for a that particular software product.

Defining requirements without affecting users too much and making it available for an unlimited amount of service providers calls for an unusual requirements process. The public developer archive, introduced in a previous section, is a perfect candidate for hosting electronic versions of requirements specifications. This would make it easy for users to post their requirements (through the user manager) and for any interested service providers to access the requirements quickly and easily.

Volere snow cards[23] are a very effective way to define requirements by simply filling in fields such as description, rationale, fit criterion and so forth. An electronic version of Volere snow cards might be good for requirements specifications for a User-Driven Development method such as the one defined in the previous section. The fields would need some adaptions, some should be removed and new fields introduced, but the general method of snow cards suit the purpose quite well.

5.1.2 Design and Implementation

Prototyping is probably the most suitable way to design and implement custom or bespoke software via User-Driven Development. Releasing prototypes is a quick way to understand whether developers have understood the requirements correctly and get feedback on each service provider’s work. Prototypes do diffuse the processes for design and implementation of software somewhat, so it is best to view prototypes as a good solution for both.

Evolutionary and incremental prototyping[10, 11] are two slightly different approaches to prototyping which work for User-Driven Development. While evolutionary prototyping is better suited when an initial design has not been decided upon and requirements are not quite clear, incremental prototyping is good for adding new features or modifications to software where an initial design exists.

When designing new software where a basis for further development is not available, i.e. no free software to build upon exists, rapid evolutionary prototyping for a single core component in the system can quickly build a working free software prototype which can be used as a basis. If a basis exists or has been developed through evolutionary prototyping, further development could be performed through incre-
mental prototyping, i.e. implementing new requirements to the basis.

For service providers it will probably be best to use agile software development methods to cope with the rapid development of the system, which results from using prototyping for design and implementation. However, any particular method is not preferred, only that it suits the service provider and does not become burdensome so that the provider can be a competitive force in service provisions.

User-Driven Development fosters competition in design and implementation since any service provider can submit their work on any requirement defined by the users. That does however not guarantee that the first submitting service provider automatically gets accepted. Other service providers have an opportunity to criticise each work with well founded arguments or add functionality to the work. So a first to market with substantial quality is probably the best strategy to choose for service providers.

5.1.3 Testing

The free and open source software community has an effective static analysis testing procedure, peer reviews[18], which can and should be used extensively as the main quality assurance technique. The service providers are responsible themselves to constantly be on the lookout for low quality submits from one another. Competing service providers, i.e. developers of the same piece of the project, are responsible for finding bugs and bad quality in submissions of a service provider. It cannot be stressed enough that any criticism should always be backed up by strong reasoning to avoid an unfair and confusing competition. Cooperating service providers, i.e. developers working on different parts of the system, are responsible for checking whether bad quality of a submission by another provider affects their work and point it out through constructive criticism.

Normal and accepted testing procedures such as unit, integration, and system testing[4] should be optional for the user. That is, the user should be able to decide whether testing services are purchased. Of course testing is always recommended, but perhaps users would like to take a chance or the software might not be critical so testing may not have a high priority. Of the three, integration testing is perhaps the most important testing method for further development of the product. Both the current integration and future integration possibilities should be tested. One can assume that unit testing is done by the service providers if peer reviewing is performed on any submission, otherwise the submitting service providers would risk damaging their reputation as a respectable developer. System testing is something which can be expected that the user does by using the system. It is very likely that the user will be more understanding of problems when the development is run and managed by him or her and system testing is an optional service.

Acceptance testing of prototypes or preprototypes[6] is important in User-Driven Development. The users must be very clear on whether a submitted work is to their liking. Accepting work into the project is a statement declaring which direction the project is going. Before accepting submitted work the user must try it out in the user environment (perhaps on dedicated test machines set up by a testing service provider) and all comments from reviews be read and understood. The user always has the last saying and should not be controlled by any service provider. However, the user must be fair and be willing to look at issues from other perspectives. Eric S. Raymond expresses this in the Cathedral and the Bazaar by claiming that it is important to be able to recognise good ideas from others and that the most striking solutions come from realising that ones concept of the problem was wrong[21]. Strong reasoning, explained in a language the user understands, is therefore vital when trying to get users to choose ones work above others.

Deployment of the product is not a distinct and necessary step. The user decides when an acceptable solution has been developed. As said before, the user has the last saying through the results of acceptance testing. When the complete product with all requirements has been accepted by the user, it can be seen as a successful deployment of the system.

5.2 Information Flow

It is very important to look at how information flows between users and developers in User-Driven Development. Lack of a good information flow might cause numerous problems and even stall the software development. It is therefore necessary for users and developers to reach an agreement for how and what information is shared between the two entities.

Users must be aware that they need to form their requirements and other information they give to service providers in a structured and understandable way. Service providers must also be aware of whether users can understand what is being said and refrain from using technological jargon when communicating with the user. This cannot be achieved in every flow of information, e.g. it is unnecessary to explain interfaces to software libraries in a non-technical way, but clear commenting should be something which every stakeholder strives for.

A possible model, shown in figure 3, can be used to visualise the information flow for each of the development activities above and how it flows between different information spaces has been introduced by Björgvinsson and Thorbergsson[2]. The model shows in a relatively understandable way what information flows and where.
Figure 3. Information flow model

6 Conclusions

From the findings and discussions of this paper it is safe to conclude that User-Driven Development is a feasible option for software development. Existing techniques and methods can be used in the various areas of software development so the only hindrances might come from the changes to the corporate environment.

However, as can be seen from the research and discussions in this paper User-Driven Development still needs a lot more research. This paper has tried to identify the areas where more research is needed in order to make User-Driven Development a viable option for users and developers. It is important to define a good business model for both users and service providers. It is also necessary to dig deeper into the methods used for the different development aspects. It might be beneficial to look at psychological implications, especially communications, identify and explain fundamental services, and identify beneficial users. The legal environment must be examined and finally the effectiveness of the methodology verified with an experiment.

References

An Approach to Software Artefact Specification for Supporting Product Line Systems

Waraporn Jirapanthong
Faculty of Information Technology
Dhurakij Pundit University
110/1-4 Prachachuen Road,
Laksi, Bangkok 10210, Thailand
waraporn@it.dpu.ac.th
+66(0)9547300 ext. 207

Abstract - This research is aimed to resolve the difficulties of software specification, particularly software product line systems. The Meta model is introduced as the reference model for software product-line artefact specification. During the software product line engineering, requirements and design artefacts are produced. Moreover, we have demonstrated the approach to software product line specification by creating different scenarios of product line development. Particularly, five tasks were created to demonstrate different situations of software product line development, involving different types of documents and different stakeholders. The experiments of artefact creation have been evaluated by considering two criteria: precision and recall measures; and satisfaction of users.

Keywords: Software Product Line, Software Artefacts, Software Specification, Product Family.

1 Introduction

In recent years we have been experiencing the proliferation of a large number of software systems that share a common set of features and have also their own distinct characteristics. Examples of such systems are found in the telecommunication domain in which products including personal digital assistants (PDAs), mobile phones, and pagers have many common characteristics. Other examples are found in the automobiles, electronics, medical imaging, and elevator control domains. These systems are known in the literature as product line systems (Ardis and Weiss 1997, Bass et al. 2003, CAFE 2003, Clements and Northrop 2002, Clements and Northrop 2004, Staudenmayer and Perry 1996, Weiss and Lai 1999) and are characterized as being software systems that share a common set of features and are developed based on the reuse of core assets and addition of new functionalities. According to the software product line development, the main activities are analysis, design, and implementation of similar and different aspects of the systems:

1) Analysis – this activity is aimed to explore and justify the requirements of product line systems which represent the common and variable aspects of the systems. In particular, the artefacts being generated from the analysis process of software product line systems are namely reference requirements.

2) Design – this activity is aimed to elaborate the requirements from the analysis process and to design the software systems for the product line. More specifically, the design presents the commonality and variability in design aspects. The artefacts being generated from the design process of software product line systems are namely software product line architecture.

3) Implementation – this activity is aimed to implement the requirements and design artefacts produced from previous processes as components and to assemble a software system which includes common and variable aspects of a product line. In particular, the artefacts being generated from the implementation process are a set of reusable software components and software systems of product line.

In principle, the reference requirements, software product line architecture, and reusable software components are gradually generated during the process of software product line development. They are later reused for developing a software product member in an effective and efficient way. It is meant to support the risk reduction during the software development. However, according to the majority of approaches being used in organisations, there are some issues found:

1) the software product line approaches proposed recently are not flexible, practical, and appropriate enough to the conventional approaches being used in the organisations;

2) different organisations have various behavioral cultures and traditions depending on the strategies and missions of the organisations. In particular, many organisations found difficulties to adopt the software product line approaches to fit into their strategies and missions.
Consequently, there are still errors and mistakes during the development of software systems that requires the reuse of software components. Also, invalid use of software product line approaches decreases the benefits of having the product line systems. We advocate the fact that the software product line systems should be established in an organisation, since the systems support the reuse of software components which leads less error-prone and less time consuming. We expect to reduce the difficulties of development activities, in particular, analysis and design.

This paper introduces the meta model of software specification for supporting the development of software product line systems in organizations. Firstly, we have investigated which artefacts are playing the main roles in the process of product line system development and which artefacts are applied in organizations; Secondly, we proposed a model for specifying of software product line artefacts. The model is proposed by taking into consideration: (a) the semantics of document types; (b) the activities during the software development process; and (c) the available techniques and tools being used in organisations. Moreover, we have justified the model for generating the artefacts in the domain of product line systems through five different scenarios. Each scenario allows the evaluation of generating the documents that occurs during the process of product line system development. The remaining of this paper is structured as follows. In section 2, we describe the meta model for specifying of artefacts. In section 3, we present and discuss the evaluation and analysis our model. In section 4, we conclude and discuss the approach.

2 Software Artefact Specification for Product Line Systems

Our work concentrates on the specification of artefacts generated during the phases of domain analysis and domain design. Particularly, the approach includes two main essentials: (i) the types of documents represented software artefacts created during the phase of analysis; and (ii) the types of documents represented software artefacts created during the phase of design. Additionally, we believe that a feature-based object-oriented engineering approach is required when developing product line systems. A feature-based approach is important to supporting domain analysis and design, enhances communication between customers and developers in terms of product features, and assists with the development of software product line architecture. On the other hand, an object-oriented approach is necessary to assisting with the development of the various product members. As the following section, we elaborate the idea of applying feature-based object-oriented engineering approach. We describe each type of software artefacts. We also give examples by referring the system domain of mobilephone systems.

2.1 Requirements Artefacts

The requirements artefacts created during the analysis phase is represented by feature model and use case. In the following, we described the details.

Use Case

Use case is a textual specification language that captures a contract between the stakeholders of a system about its behavior (Cockburn 1997). In our work, we express the requirements of product line systems by extending the use case definition given by Cockburn (Cockburn 2000). A use case is composed of:

1. **Use Case** – the element consists of three attributes, which are information of the use case: (a) Use_Case_ID – this attribute is identified as a use case; (b) System – this attribute specifies which domain of product line is; and (c) Product_Member – this attribute specifies for which product member the use case is specified.

2. **Existential** – this element is used to represent the existential of a use case. It consists of an attribute Commonality_Variability – this attribute can be (i) mandatory, which indicates a use case must be satisfied by product members; (ii) alternative, which indicates a use case may or may not be satisfied by product members; and (iii) optional, which indicates a use case may or may not be satisfied by a product member. Moreover, in the case that the attribute Commonality_Variability is specified as “alternative”, the element Existential can consist of sub-element Variant_Point. The element Variant_Point specifies a particular point of the use case’s variability. The element Variant_Point can consist of a sub-element either Variant or Parameter. The element Variant specifies a set of alternatives for the particular variant point, as the element Parameter specifies the domain of the Variant_Point. Note that in the case that the attribute Commonality_Variability is either mandatory or optional, the element Variant_Point may not exist.

3. **Title** – the element Title is the title of use case.

4. **Description** – the element Description is specified for a brief textual description.

5. **Level** – the element describes the level of functionality that it describes within a system.

6. **Preconditions** – the element describes the conditions that must be satisfied before its execution.

7. **Postconditions** – the elements describes the conditions that must be satisfied after its execution.

8. **Primary_Actors** – the element specifies primary users of the use case.

9. **Secondary_Actors** – the element specifies secondary users of the use case.

10. **Flow_of_events** – the element specifies a list of the events that trigger the use case and the specification of the normal events that occur within it. The element Flow_of_events consists of the sub-element Event,
which specifies a particular event being preceded in the use case.

(11) **Exceptional events** – the element describes the events that do not always occur when the use case is executed.

(12) **Superordinate use case** – the element specifies a use case for which the use case is elaborated.

(13) **Subordinate use cases** – the element specifies a use case to which the use case is specified.

Figure 1 illustrates an example of a use case *Sending a Message* from a mobile phone for product member PM1 of the mobile phone case study. The use case is identified with UseCaseID (“UC1”), System (“MobilePhone”), and Product_Member (“PM1”). The use case contains Existential element which its attribute Commonality_Variability is specified as “Alternative”. The sub-element Variant_Point (“v1”) is declared along with the sub-element Variant which includes a set of possible values for v1. It also contains elements i.e. Title, Description, Level, Preconditions, Postconditions, Primary_actor, Secondary_actors, Flow_of_events, Exceptional_events, Superordinate_use_case, and Subordinate_use_case that describe the context of the use case.

<table>
<thead>
<tr>
<th>Use_Case</th>
<th>UseCaseID=&quot;UC1&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>&quot;MobilePhone&quot;</td>
</tr>
<tr>
<td>Product_Member</td>
<td>&quot;PM1&quot;</td>
</tr>
</tbody>
</table>

**Existential**

| Commonality_Variability="Alternative"
|-----------------|

**Variant_Point**

<table>
<thead>
<tr>
<th>v1</th>
</tr>
</thead>
</table>

**Variants**

<table>
<thead>
<tr>
<th>v1</th>
</tr>
</thead>
</table>

| [keying-in a phone number of a receiver, selecting a phone number from a list of contacts] |

**Title** Sending a Message

**Description** The phone is able to send a text message. The user can specify an address of a receiver by selecting from a list of contacts.

**Level** User Goal

**Preconditions** The user has already selected function of sending a text message from the main menu.

**Postconditions** The phone has sent the message.

**Primary_actor** The user

**Secondary_actors** -

**Flow_of_events**

| Event 1 | The system shows an editor for writing a message. |
| Event 2 | The user inputs a phone number by [v1]. |
| Event 3 | The system displays the phone number to which the message is being sent. |
| Event 4 | The user enters the message and confirms sending the message. |
| Event 5 | The system sends the message and displays an acknowledge on the screen. |

**Exceptional_events** -

<table>
<thead>
<tr>
<th>Superordinate_use_case -</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subordinate_use_case -</td>
</tr>
</tbody>
</table>

Figure 1: an example of a use case

**Feature Model**

We proposed to apply the feature model presented in FORM (Kang et al. 1998) which is based on the feature model proposed by (Kang et al. 1990). The authors enhanced the feature model with a textual specification for each feature. Our feature model describes the requirements artefacts of a product line system and illustrates the features available in the line.

A feature is represented by a name and can be (i) mandatory, when it must exist in the applications in the domain; (ii) optional, when it is not necessary to be presented in the applications in the domain; or (iii) alternative, when it can be selected for an application from a set of features that are related to the same parent feature in the hierarchy.

The features can be classified into four groups namely (i) application capabilities, signifying features that represent functional aspects of the applications (e.g. calling, connectivity, personal preference, and tool features); (ii) operating environments, signifying features that represent attributes of the environment in which product members are used and operated (e.g. network, input and output methods, and operating system features); (iii) domain technologies, signifying features that represent specific implementation and technological aspects of the applications in the domain (e.g. WAP and XHTML browser types; specific Java application support like mobile media and wireless messaging application programming interface; SMTP, POP3, and IMAP4 network protocol features); and (iv) implementation techniques, signifying features that represent more general implementation and technological aspects of the applications, but not necessary specific for the domain (e.g. PGP and DES encryption methods; AMR, MIDI, and MP3 sound formats; and 3GPP and MPEG video format features).

Feature can also be related by different types of relationships. Examples of these relationships are (i) composed_of, (ii) generalisation/specialization, and (iii) implemented_by relationship types.

Figure 2 presents an example of a textual specification for feature *Text Messages*. As shown in Figure 2, the textual specification represents (i) a name, (ii) a description, (iii) issues and decisions representing trade-offs, rationale, or justifications for including the feature in an application, (iv) a type such as application capabilities, operating environments, domain technologies, and implementation technologies, (v) commonality indicating if a feature is mandatory, optional, and alternative, (vi) relationship with other features such as composed-of, implemented-by, generalisation/specialization, (vii) composition rule representing mutual dependency and mutual exclusion relationships to indicate consistency and

---

completeness of a feature, if any, and (viii) allocated-to-subsystem indicating the name of a subsystem that contains the feature, if any.

<table>
<thead>
<tr>
<th>Feature-name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text Messages</td>
<td>The phone can edit, send, and receive a short text message</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Issues and decision</th>
<th>Text message over mobile phone is a way of communication</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Application capability</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Commonality</th>
<th>Mandatory</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Composed-of</th>
<th>Sending Text Messages, Receiving Text Messages, Editing Text Messages</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Composition-rule</th>
<th>Messaging</th>
</tr>
</thead>
</table>

| Allocation-to-subsystem | |
|-------------------------| |

**Figure 2: an example of a textual specification for feature Text Messages**

### 2.2 Design Artefacts

In our approach, we adopt UML class diagram, statechart diagram, and sequence diagram to present the software product line architecture. In the following, we described the details.

#### Class Diagram

We extend the class diagram presented in (Clauss 2001) by adding some elements. The diagram consists of elements as described following:

1. **Class Diagram** – the element consists of three attributes, which are information of the class diagram: (a) **Class_Diagram_ID** – this attribute is identified as a class diagram; (b) **System** – this attribute specifies which domain of product line is; and (c) **Product_Member** – this attribute specifies for which product member the class diagram is specified.

2. **Existential** – this element is used to represent the existential of a class diagram. It consists of an attribute **Commonality_Variability** – this attribute can be (i) **mandatory**, which indicates a class diagram must be satisfied and altered for particular product members; and (ii) **alternative**, which indicates a class diagram must be satisfied and altered for particular product members; and (iii) **optional**, which indicates a state chart diagram may or may not be satisfied by a product member.

3. **State** – the element **State** specifies the system’s particular status. The states of the diagram can be representing some aspects of the variability. We define three types of a state for expressing variability in product line: (i) **variationPoint**, which represents a state that initiates a variation point of product line; (ii) **variant**, which represents an alternative states of a particular variation point; and (iii) **optional**, which represents an optional state.

4. **Transition** – the element **Transition** describes a driving method to transform a state to another state. To capture and represent variability of a product member, a transition can be specified as one of three transition types: (i) **variantTransition**, which describes one of possible driving methods to transform a state to another state; (ii) **parameterTransition**, which describes a transition requiring a parameter to drive the method; and (iii) **optionalTransition**, which describes a possible driving method to transform a state to another state.

#### State Chart Diagram

In the Meta model, we propose the extension of UML state chart diagram by adding some elements. The diagram consists of elements as described following:

1. **State Chart Diagram** – the element consists of three attributes, which are information of the state chart diagram: (a) **State_Chart_Diagram_ID** – this attribute is identified as a state chart diagram; (b) **System** – this attribute specifies which domain of product line is; and (c) **Product_Member** – this attribute specifies for which product member the state chart diagram is specified.

2. **Existential** – this element is used to represent the existential of a state chart diagram. It consists of an attribute **Commonality_Variability** – this attribute can be (i) **mandatory**, which indicates a state chart diagram must be satisfied by product members; (ii) **alternative**, which indicates a state chart diagram must be satisfied and altered for particular product members; and (iii) **optional**, which indicates a state chart diagram may or may not be satisfied by a product member.

3. **State** – the element **State** specifies the system’s particular status. The states of the diagram can be representing some aspects of the variability. We define three types of a state for expressing variability in product line: (i) **variationPoint**, which represents a state that initiates a variation point of product line; (ii) **variant**, which represents an alternative states of a particular variation point; and (iii) **optional**, which represents an optional state.

4. **Transition** – the element **Transition** describes a driving method to transform a state to another state. To capture and represent variability of a product member, a transition can be specified as one of three transition types: (i) **variantTransition**, which describes one of possible driving methods to transform a state to another state; (ii) **parameterTransition**, which describes a transition requiring a parameter to drive the method; and (iii) **optionalTransition**, which describes a possible driving method to transform a state to another state.

#### Sequence Diagram

We also propose the extension of UML sequence diagram by adding some elements. The diagram consists of elements as described following:
(1) **Sequence Diagram** – the element consists of three attributes, which are information of the class diagram: (a) *Sequence_Diagram_ID* – this attribute is identified as a sequence diagram; (b) *System* – this attribute specifies which domain of product line is; and (c) *Product_Member* – this attribute specifies for which product member the sequence diagram is specified.

(2) **Existential** – this element is used to represent the existential of a sequence diagram. It consists of an attribute *Commonality_Variability* – this attribute can be (i) *mandatory*, which indicates a sequence diagram must be satisfied by product members; (ii) *alternative*, which indicates a sequence diagram must be satisfied and altered for particular product members; and (iii) *optional*, which indicates a sequence diagram may or may not be satisfied by a product member.

(3) **Sequence** – the element *Sequence* specifies an interaction between an object and actor, or between objects. We propose three types of sequences for expressing variability in a sequence diagram: (i) *variationPoint*, which represents a sequence that initiate a variation point of later sequences; (ii) *variant*, which represents an alternative sequence of a particular variation point; and (iii) *optional*, which represents an optional sequence.

(4) **Message** – the element *Message* basically represent a called operation from an object interacting to another object. A message can be representing the variability of a product member. Specifically, we propose three types of messages for expressing the variability: (i) *variantMessage*, which is one of possible messages being sent from a *variantPointSequence* to another sequence; (ii) *parameterMessage*, which is a message requiring a parameter to drive the method and (iii) *optionalMessage*, which is an optional message that may or may not be sent on a sequence.

### 3 Evaluation and Analysis

In order to evaluate and demonstrate our approach, we have implemented a prototype tool. We envisage the use of our tool as a general platform for creating the software artefacts for product line’s analysis and design. The tool has been implemented in Java. Additionally, the objective of the evaluation is to evaluate whether the model helps an organization in making software product line artefacts more precise and consistent. The testing was inducted by concerned the following factors: (a) *selection of participants* – The testing scenarios used in our evaluation were based on two main factors. The first factor was concerned with the different ways in which organizations can develop product line systems. As proposed in (Krueger 2001), organisations can develop product line systems in three different ways: (i) when an organisation decides to analyze, design, and implement a line of products prior to the creation of individual product members (proactive approach); (ii) when an organisation enlarges the product line systems in an incremental way based on the demand for new product members or new requirements for existing products (reactive approach); and (iii) when an organisation creates a product line based on existing product members by identifying and using common and variable aspects of these products (extractive approach).

According to our survey, we found that these approaches are not mutually exclusive and can be used in combination. For instance, it is possible to have product line systems initially created in an extractive way to be incrementally enlarged over time by using a reactive approach. Specifically, we randomly selected up to 50 organisations of different business areas such as software production, financial, trading, logistics, airlines, insurance and so on by analysis the infrastructure of the organizations. If there organisations are not fitting in to the criteria, that organisation was not included in our study. Specifically, the criteria which we applied to justify the organisations for our testing are: (i) maturity, (ii) size, and (iii) number of software products.

The second factor was concerned with the stakeholders involved in the product line system development process. Various stakeholders may be involved in this process ranging from market researchers, to product managers, requirement engineers, product-line engineers, software analysts, and software developers. These stakeholders contribute in different ways to the product line system development process, have distinct perspectives of the system, and have distinct interests in different aspects of the product line systems. For example, a market researcher may be interested in the requirements and features of a new product member to be developed, while a software developer may be interested in the design and implementation aspects of this new product member. Therefore, the stakeholders would be interested in different types of documents that may assist them in their various tasks during system development.

#### 3.1 Test Cases

In order to take into consideration the various ways of developing product line systems, the heterogeneity of stakeholders, and document types. The five scenarios used in our testing include:

(a) the creation of a new product member from existing product line – for this scenario, it is necessary to compare various documents of a product line, documents of existing product members and new product member.

(b) the creation of product line from already existing products – for this scenario, it is necessary to compare various documents of existing product members.

(c) changes to a product member in a product line;

(d) changes to the core assets of a product line; and

(e) impact of changes to the core assets of a product line to a product member.

For each of these scenarios we have identified the stakeholders involved in the process and the types of documents according to the meta model that are related to the
scenarios. We asked our participants to perform some of above tasks twice: (i) by applying the prototype tool prepared by the author; and (ii) by manually performing. Manual practice may subject to applying any existing software of the organizations. The results of each task are software artefacts which are developed according to the meta model. The types and number of software artefacts are various in each task. The author has prepared the software artefacts for some tasks as required.

3.2 Measurement of Test

In this evaluation, we have conducted sets of testing related to five different scenarios of product line system development. The tests are justified by concerning two aspects. Firstly, we have used the following standard definition of recall and precision given in (Faloutsos and Oard. 1995). The authors described that precision measure represents the soundness of documents to be retrieved due to an inquiry and recall measure represents the proportion of the relevant documents. We then adopt the measurement techniques to capture the commonality and variability of a product line system. As the following, the precision and recall are calculated by:

\[
\text{Precision} = \frac{|A \cap S|}{|A|} \quad \text{Recall} = \frac{|A \cap S|}{|S|}
\]

where
- \(A\) is the set of artefacts which are available in a system;
- \(S\) is the set of artefacts which are specified by participants; and
- \(|X|\) denotes the cardinality of the set \(X\), in which represents the artefacts are specified validly.

Note that an artefact which is considered in a test is a fine-grained element of document. Secondly, we measured the time to complete a task when users were proceeding each one of test cases with normal procedure and available tools, and the time to complete the same task with the proposed model and prototype tool. We also asked the participants to fill in our questionnaire containing the questions with a five-point scale to measure aspect of use. The score of each aspect of use—easy to decide to next step, easy to understand the requirements and design, easy to literate and locate information and overall satisfied with analysis and design based on a five-point scale that score 1 = Strongly Disagree, 2 = Disagree, 3 = Neutral, 4 = Agree and 5 Strongly Agree.

3.3 Evaluation Results and Analysis

In the tests, we have organized groups of participants from five organisations which properties are fitted into our criteria (as discussed previously). However, since it is an agreement between the author and participants. We do not explicit the source and profile of participating organisations in this paper due to the confidential issues. Each group is assigned to perform some of defined tasks (as describe previously).

<table>
<thead>
<tr>
<th>Table 1: Precision and Recall Rates (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>Task 1</td>
</tr>
<tr>
<td>Task 2</td>
</tr>
<tr>
<td>Task 3</td>
</tr>
<tr>
<td>Task 4</td>
</tr>
<tr>
<td>Task 5</td>
</tr>
<tr>
<td>Average Precision/Recall of all tasks</td>
</tr>
</tbody>
</table>

Table 1 shows the results of our testing for each task in terms of recall and precision rates. The results shown in the table provide positive evidence about our approach to apply the meta model to specify software product line artefacts at a high level of recall and precision. We applied the histograms to compare the precision and recall in the testing. Figure 3 shows that the precision figures in all the cases and the recall figures in all the tests are not so significant. On average, the performance of our approach in terms of precision and recall measurements in tests seems to be consistent.

Additionally, the time spent during the generation of the software product line artefacts in the tests varies depending on the size of the artefacts and the number of requirements and design artefacts. Moreover, the experience and expertise of stakeholders who involve the specification process also
contributes to an increase of the processing time. Moreover, after completely all tasks, the subject was observed regarding attitudes toward various aspects of software specification without and with the meta model. The results are summarized in Figure 4. Figure 4 depicts how participants evaluated the applying of the meta model for specifying software product line artefacts i.e. requirements and design artefacts through our questionnaire. As seen in the figure, the participants agreed average of 3 scores ease of deciding next step in specification with the conventional software engineering methodologies, while they agreed 4.3 scores with our approach. Similarly, ease of understanding the rational of systems, the participants agreed on average 2.3 scores with the conventional software engineering methodologies and 4.5 scores with our approach. Ease of locating the information, the participants agreed 3.1 scores and 4.4 scores for the conventional documents and our proposed documents respectively. On the average, participants feel more satisfied with specification of software product line systems when applying our meta model than conventional methods and techniques.

![Figure 4: Comparison of Qualitative User Evaluation on conventional software engineering approaches and our approach for specification of software product line systems](image)

### 4 Conclusion and Discussion

Basically, problems of the establishment and maintenance of product line systems in organisations are: (i) the difficulty to get support from organisations; (ii) the uncontrolled growth of variety; (iii) the difficulty in communication i.e. sufficient resources e.g. staff or tool to facilitate the communication; differences in organisational cultures; distinct organisational structures; and stakeholders’ attitudes and aspirations; (iv) the difficulty of defining commonality and variability; (v) the difficulty of documenting management; (vi) the confliction and dependency between artefacts in product line systems; (vii) the difficulty to specialise variability; and (viii) issues of evolution of product line systems.

Additionally, this research has shown that some degree of systematic process in creating software artefacts which can be partly facilitated by the prototype tool. The creation of documents captures the semantics that are represented through the structure of each document type. As shown in this research, the results of creation are measured by using precision and recall rates. The average precision measured as 83.0% and average recall measured as 93.0%. The results shown in the research are giving very positive to the approach. We have demonstrated the possible situations of the use of meta model during the development of product line systems. Particularly, the research has found that the degree of reusing core assets of product line systems affects the cost of the development of the systems. The cost of the product line system development depends on the proportion of reuse of the core assets for the development of product members. However, the poor reuse would cause higher cost to the product family system development. The specification of software product line artefacts influences the development by reducing the cost i.e. effort and time. Moreover, we have shown that different stakeholders, who have different experiences in the product line system development process, have different perspectives regarding to software artefacts. Several artefacts are used to represent stakeholders’ requirements and design. Different types of software artefacts represent different levels of software product-line systems i.e. coarse-grained and fine-grained levels. This can facilitate to understanding of document specification.

### 5 References


Accuracy of Software Cost Estimation

Hassan Pournaghshband
Southern Polytechnic State University
School of Computing and Software Engineering

Shahriar Movafaghi
Southern New Hampshire University
Information Technology Department

Abstract

Effective software cost estimation is one of the most challenging and important activities in software development. Proper project planning and control is not possible without a sound and reliable estimate. Since accurate cost estimates are critical to customers as well as software developers, researchers and practitioners have been attracted to work on this issue from the early days of software development. In this paper, we elaborate on software cost estimation accuracy and propose a cost estimation process that can be used by the project manager (or anyone else in charge of the estimation task) to estimate the project cost with a good degree of accuracy.

1. Introduction

Software cost estimation can be regarded as the process of predicting the efforts needed to develop the software systems and usually emphasizes estimation of the size of the software system to be developed. A good software cost estimate can help the customer to make a wise and informative decision on whether to go with the project or abort it. A bad cost estimate, on the other hand, can contribute to inaccurate prediction of software development costs, which in turn could lead to more serious problems. Underestimating the costs can lead to understaffing the project resulting in staff burnout, under-scoping the quality assurance effort (running the risk of low quality deliverables), and setting too short a schedule resulting in loss of credibility, as deadlines are missed. For those who figure on avoiding this situation by generously padding the estimate, overestimating the cost can be just about as bad for the organization. This is a very serious issue especially because overruns resulting from inaccurate estimates are believed to occur frequently. In fact, the study reported in [1] found that nearly two-thirds of all major projects substantially overrun their estimates. If you give a project more resources than it really needs without sufficient scope controls it will probably use them, and as a result the project is then likely to cost more than it should, and would take longer to deliver than necessary. It could also delay the use of resources on the next project. Whenever we make an estimate, we want to know how close our numbers are to reality. The bottom line is that we won’t know about this until we complete the project, and therefore, we have to live with some uncertainty. Naturally, we want our estimates to be as accurate as possible, given the data we have at the time making estimates. What is meant by an “accurate” estimate? In this sense, accuracy is an indication of how close something is to reality. This is different from precision which is an indication of how finely something is measured. There are many reasons why accurate cost estimation is vital [2.] Accurate cost estimation not only helps in prioritizing development projects with respect to an overall business plan, it is also used in determination of resources to commit to the project. In this paper, we elaborate on software cost estimation accuracy and propose a cost estimation process that can help the project manager (or anyone else in charge of the estimation task) regarding this crucial charge. The rest of the paper is organized as follows. In
Section 2, we will discuss some of the common software cost estimation issues and problems, and show how one can deal with them. In Section 3, we will present an estimation approach and discuss the process that can be used to “accurately” (to some degree) estimate the cost of software. Conclusions and recommendations for future research are given in Section.

2. Issues and Problems

Estimating software cost is usually a difficult task. In fact, practically we don’t have any single model that can estimate the cost of software with a very high degree of accuracy. A software cost estimate lacking necessary attributes can easily contribute to inaccurate prediction of software development costs, which could be disastrous for the organization. Authors in [1], have proposed nine management guidelines for better cost estimation. These guidelines are 1) assign the initial estimating task to the final developers, 2) delay finalizing the initial estimate until end of through study, 3) anticipate and control user changes, 4) monitor the progress of the proposed project, 5) evaluate proposed project progress by using independent auditors, 6) use the estimate to evaluate project personnel, 7) computing management should carefully study and approve the cost estimate, 8) rely on documented facts, standards, and simple arithmetic formulas rather than guessing, intuition, personal memory, and complex formulas, and 9) don’t rely on cost estimating software for an accurate estimate [1]. While these guidelines seem to be convincing it would make more sense to modify the last one from “don’t rely on cost estimating software for an accurate estimate” to “rely on cost estimating software only to some degree of accuracy.”

Many researches have been done regarding good software cost estimates and necessary attributes that such a cost estimate should possess. According to [4], the following attributes should be part of any good software cost estimate: 1) should be supported by both the project manager and also the development team, 2) should be based on a well-defined cost model, 3) should be based on a database of relevant (similar) project, 4) should be carefully defined so that its key risk areas are well understood, and 5) should be accepted by all stakeholders as realizable. We believe that in addition to considering these attributes for a good cost estimate, we should also include some of the key concerned parties in the process. This is in fact what our approach is about. In Section 3, we present an effective software cost estimation process that can be employed by the project manager for cost estimation considering major factors affecting the accuracy of the estimate.

3- Our Approach

In this section we present a non-algorithmic approach for accurately estimating the software cost using knowledge and expertise of key interested parties. The outcome of this approach (i.e., comparing the actual cost with individual’s estimates) can be also used for future estimation of other projects. The better the individual’s estimate, the higher the weighting rate assigned to them for future estimations.

The process – Figure-1 shows the seven steps of the process that the project manager can use for estimating the software cost. These seven steps are explained below.

Step-1. In this step, the project manager prepares a list of key people that have the potential to give a good cost estimate. The list could include the project manager, developers, customers, external experts, etc.

Step-2. Next, each estimator is assigned a weighting rate (say 1 to 10, 10 the highest) according to a set of factors such as application domains, experience, interest in the project, educational background, etc.

Step-3. Here, a short questionnaire such as the one shown below is prepared for the estimators to be complete after they are done with their estimates.

<table>
<thead>
<tr>
<th>Question</th>
<th>Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimate</td>
<td>Low</td>
</tr>
</tbody>
</table>

...
Step-4. In this step, each estimator makes an estimate using his/her knowledge and expertise.
Step-5. In this step, each estimator completes the questionnaire.
Step-6. Next, the project manager collects all estimates along with the completed questionnaires.
Step-7. Finally, the project manager makes the final estimate as follows:
He/she first creates a table using information collected from estimators.

<table>
<thead>
<tr>
<th>Estimator</th>
<th>Estimate</th>
<th>Score</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>e1</td>
<td>s1</td>
<td>r1</td>
</tr>
<tr>
<td>A2</td>
<td>e2</td>
<td>s2</td>
<td>r2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>An</td>
<td>en</td>
<td>sn</td>
<td>rn</td>
</tr>
</tbody>
</table>

In this table \( A_i \) represents estimator \( i \), \( e_i \) is the estimate made by \( A_i \), \( r_i \) is the weighting rate for \( A_i \) (from step-2), and \( s_i \) is the total score given by estimator \( A_i \) to questions in the questionnaire.

Next, the project manager can use the information given in this table to compute the final estimate using the following formula.

\[
E = \frac{e_1s_1r_1 + e_2s_2r_2 + \cdots + e.ns_nr_n}{s_1r_1 + s_2r_2 + \cdots + s_nr_n}
\]

Researchers and practitioners all agree that experience of the software cost estimator plays a significant role in the degree of accuracy of the cost. Authors in [6] discuss factors affecting effort estimation errors in software development projects. One of the four dimensions considered in their paper, in fact, is the estimator’s experience. In our approach, as it is shown above, the process of estimating the software cost puts a significant weight on estimator’s experience. Our approach differs from other approaches also because the process requires the experience and knowledge of a group of people rather than an individual. In addition,
the process not only works for estimating the cost of software at hand, it also greatly helps with cost estimation of upcoming projects

4. Conclusions

In this paper, we elaborated on software cost estimation accuracy and proposed a cost estimation process that can be used by the project manager (or any one else in charge of the estimation task) to estimate the project cost with a good degree of accuracy. The outcome of this approach (i.e., comparing the actual cost with individual’s estimates) can be also used for future estimation of other projects. The better the individual’s estimate, the higher the weighting rate assigned to them.

References

Looking for Smells: Visualizing Java Code with Dotplots

Alvin Jefferson and Michael Wainer
Department of Computer Science, Southern Illinois University Carbondale
Carbondale, IL 62901 USA

Abstract - As code is modified to accommodate changing requirements it often begins to outgrow initial assumptions and becomes awkward, inefficient and difficult to understand and maintain. Agile software development methods expect change, and as code degrades (begins to smell), they refactor it to continuously adjust the design to the emerging requirements. When, where and how to refactor code presents many challenges, here we explore the possibility of a tool which uses dotplots to help visualize where code smells may be occurring as well as to offer insights into how the smells might be removed.

Keywords: software development, software visualization, refactoring, code smells, tools

1 Introduction

Source code is always changing. As the development process moves forward new features are added, old features are removed, and existing code is modified. All the while the programmer is left to cope with these changes and ensure that the code remains reliable and performs the necessary tasks efficiently. Problems arise since as the code grows it becomes harder to maintain. To combat this problem many tools have been developed to find problem areas and ensure that the code development continues smoothly. This paper will describe one such tool that attempts to find these problem areas using dotplots to visualize the code.

Dotplots are a visualization technique used to graphically compare two sequences of symbols, words, or lines. A two-dimensional graph is created where matches are plotted as dots. The dots in the graph also represent the amount of duplication that occurs between the two sequences that are being compared. Duplication is the first and easiest code smell that can be found when examining Java code with the use of dotplots. Code Smells are possible indications of poor design or problematic code. The presence of duplication may also lead to the discovery of other Code Smells in the Java code.

Smells do not always mean that a problem exists, however smelly code should be reexamined to ensure that the code does not need to be refactored. To refactor means “to restructure software by applying a series of refactorings without changing its observable behavior” [1]. For example if there are multiple copies of a segment of code a helpful refactoring would be to extract the duplicate code to a method that can be called wherever it is needed. These changes make the code more manageable and maintainable. They also make it easier to spot problems in the future.

2 Related Work

There are a few tools and methods that have been the inspiration for our plug-in. The first such tool is called “dotplot”. Designed by Kenneth Ward Church and Jonathan Isaac Helfman, “dotplot” is a tool for browsing millions of lines of text and code using dotplots [2]. This tool examines the idea of viewing source code using dotplots. Church and Helfman used the dotplots to view large amounts of code on one screen to see the big picture and discover structures that might not be easily seen with other tools.

Another tool is Duploc. Duploc is a program written in the Smalltalk programming language that uses dotplots to detect duplication in source code [3]. It supports many languages including C, Java, Smalltalk, Python, and others. Duploc is interesting because it also explores the use of dotplots for visualizing source code. However, its only focus is on finding duplicate code and it does not attempt to explore any other Code Smells.

Panopticode [4] is a source code visualization tool. It creates a grid map with large boxes representing classes in a package and the smaller colored boxes within representing each method of the class. The method boxes are filled in with different colors to represent the nature of the problem within. For example one option uses the Cyclomatic Complexity metric to fill in the boxes. Regions with low complexity are shaded green while high complexity areas are shaded black. The tool uses two metrics to form the grid map. One metric determines the size of the boxes while the other metric determines the color. Many different metrics can be used for these tasks. Aside from visualizing with images, the results are also output in text form. Panopticode is useful since it shows an overall view of a whole package and uses metrics illustrate possible problems.

PMD [5] is also useful during the development process. This plug-in analyzes source code using a large set of rules and metrics to discover problem areas and possible code smells. PMD also allows users to modify and add their own rules. The plug-in format allows for quick analysis in the same environment where development is taking place and instant feedback after changes are made.
There is also increased support of agile programming and refactoring in IDEs like Eclipse [6]. The Eclipse Java development environment has some built-in refactoring tools to aid the user. With more emphasis on agile development the search for Code Smells becomes more important. That is why it is necessary to develop tools that can speed up the process of finding problem areas in source code.

### 3 Dotplots of Java Code

To generate dotplots from Java source files we first read in each file as text. Each file is taken as a string and then divided into words by splitting this string using white space as a delimiter. At this point each file is represented as a collection of words. The dotplot is now formed by comparing these two collections of words (one on the horizontal and the other on the vertical axis) and graphing each match. The dotplot is shown as this two-dimensional graph of dots and visual analysis can now begin. Much revealing information can often be found by plotting a file against itself. Areas forming similar patterns along diagonals are the areas that contain the most duplication and seemingly offer the most likely places for code smells to occur. Some example Code Smells include:

- **Duplicate Code** – The same code structure appearing in more than one place.
- **Long Method or Class** – Long methods and classes are usually hard to understand and take on too much responsibility.
- **Shotgun Surgery** – Changing the code requires many small changes in many different places.
- **Switch Statements** – Using a switch or if statement in the place of polymorphism.
- **Feature Envy** – If a method frequently calls methods of another class (often getter methods) more often than methods of its own class.
- **Inappropriate Intimacy** – Classes that spend too much time calling each other.

These descriptions from [1] give an idea of what some of the smells are. In the following examples we will see how well the plug-in can find some of these smells.

We will begin with an example from [7], the `Game` Class. The images in Figure 1 are the `winner` method from the `Game.java` file and its corresponding dotplot. It is clear from looking at the code that duplication is present and the dotplot also shows this through the presence of many diagonals and some unique rectangular patterns. The strategy is to first, look at the dotplot and upon finding this interesting pattern, go to the file to find its cause. In this case we find the `winner` method containing areas of duplication and an inefficient design. The `winner` method should be refactored. Figure 2 represents one possible refactoring and the corresponding dotplot. The removal of the duplication has resulted in a dotplot that is less cluttered.

This method represents a straight-forward example. More research needs to be done to better understand the relationship between the dotplot and the source file.

```java
public char winner() {
    if (board.charAt(0) != ' -' 
        && board.charAt(0) == board.charAt(1) 
        && board.charAt(1) == board.charAt(2))
        return board.charAt(0); 
    if (board.charAt(3) != ' -' 
        && board.charAt(3) == board.charAt(4) 
        && board.charAt(4) == board.charAt(5))
        return board.charAt(3); 
    if (board.charAt(6) != ' -' 
        && board.charAt(6) == board.charAt(7) 
        && board.charAt(7) == board.charAt(8))
        return board.charAt(6); 
    return ' -';
}
```

Figure 1: `winner` method from `Game.java` and dotplot
private boolean isSquareEmpty(int move) {
    return board.charAt(move) == ' ';  
}

public char winner() {
    for (int move = 0; move < board.length(); move++){
        if (!isSquareEmpty(move) 
            && board.charAt(move) == board.charAt(move+1) 
            && board.charAt(move+1) == board.charAt(move+2))
            return board.charAt(move);
    }  
    return ' ';  
}

Figure 2: Refactored winner method from Game.java and dotplot

Figure 3: Dotplot of makeCustomerButtons method from Table.java

Another example using the Table.java class from [7] shows the potential of this plug-in. The Table class is said to have many different code smells including Long Method, Long Class, Duplicate Code, Feature Envy, and Inappropriate Intimacy [7]. It may be very difficult to find Feature Envy or Inappropriate Intimacy using dotplots. Both of these smells rely on one class calling another, therefore focusing on repeated method calls may be the key to locating them. However these calls don’t have to be in close proximity to each other so the technique of looking in areas with high concentrations of dots may not work for these smells.

When looking at the dotplot for the Table class we see a couple of areas with duplicate code that may need refactoring. Figure 3 is a piece of the dotplot for the Table class, more specifically it is the dotplot for the makeCustomerButtons method. The dotplot shows many clear diagonals meaning that this method is a good candidate for the Duplicate Code smell. When we examine the source code, we find that the makeCustomerButtons method also contains anonymous inner classes. To refactor this class we extract these inner classes to their own classes. During the refactoring process the Feature Envy
and Inappropriate Intimacy becomes clearer. In this way we can use the Duplicate Code smell to find other more complex code smells. Wake [7] has characterized duplication as a root smell and alludes to the notion that some other smells are special cases of duplication.

Every file is different and provides its own unique dotplot. Also areas with high concentrations of dots do not always represent places where the code needs to be refactored. However, other methods of working with the dotplots may help in understanding the code better. Preliminary results are promising. It can be shown that categorizing matches with dot colors and the use of key words makes visualization of the Long Method Smell readily apparent. This also suggests that using colors may enhance the search for other code smells in the dotplots. Examination of multiple dotplots has also illustrated the presence of some common patterns. These patterns appear as variations of the main diagonal and represent short bursts of repeated code. Figure 4 shows some examples of nested if/else statements that could qualify for the Switch Statements code smell. This is illustrated by the symmetrical and even cone or pyramid structures coming off of the main diagonal. The repeating block nature of the switch statement and the nested if/else statement creates these patterns that may be easily distinguished in the dotplot and could lead to areas in need of refactoring.

Figure 4: Common dot patterns that may indicate the Duplicate Code Smell
4 Visualizing During Development

Our plug-in is designed to work with the Eclipse IDE. Eclipse was chosen because it is a popular Java development environment with refactoring support and also a good plug-in development platform and community. The plug-in allows dotplots to be created during the development process and viewed alongside the source files that are being compared. There are also features that connect the dotplot to the source files to aid the user in finding areas of interest in the source code. The plug-in lets the user click on a dot in the dotplot and receive information about the dot. This information consists of the words being compared at that dot and their line number locations in the source files. Figure 4 illustrates these features with a screenshot of the plug-in. The screenshot shows the Eclipse SDK with the Table.java file open in the top window and the plug-in in the bottom window. Figure 5 also shows the two options windows. The “File Output Menu” lets the user create a dotplot from two files and outputs the dotplot to an image file. The “Change Matching Options” box changes the matching criteria from exact match to partial match. Using exact match means that each dot in the dotplot represents two words that are exactly equal to each other character by character. Partial matches allow only part of the words to match exactly. In addition to this information the lines are also highlighted in each of the source files for even more convenient analysis. When an area in the dotplot that may contain Code Smells is found the user can first click a couple of dots to find out what words are being matched in this area. Then the source file can be examined around the line that is highlighted to see if any refactoring is needed. After the code is refactored another dotplot can be created to see how things have changed.

5 Summary

The idea behind our plug-in is simple, to find out if duplication can be used to find code smells in Java source code. To accomplish this dotplots are used to visualize the source code and its duplication. Visualizing the source code is the important feature here. The visualization using the dotplots allows for quicker scanning of the files so that the programmer does not need to go through his/her program line by line. The current level of this project only allows for comparing a file to itself or another file. This means that looking at larger projects that contain a large number of files can be tedious. However, this plug-in might be useful for students or beginning programmers. At this early stage in development it could possibly be used as a training tool for discovering code smells. With continued research we may see dotplots as another useful tool in the development process.

6 Future Work

This plug-in is still in the early stages of development and there are many features that can be added. One interesting feature would be the inclusion of multiple parsing options. Currently files are just read and split on their white spaces. However, parsing the files in another way, possibly with the use of abstract syntax trees, may yield different dotplots that could lead to alternative insights. Also different matching criteria could be explored. Instead of only using exact or partial matches to get a dot, other criteria could be used to get new dotplots. Another area of exploration is to think of larger projects and packages. At the moment we are just comparing one file to another file, but it could be interesting to find a way to compare over more than just two files or compare a project to another project. As we continue to move forward we will also learn more about how the dot patterns relate to the different Code Smells and may be able to improve our tool to better visualize these smells.

7 References


Ensuring the Maturity of Processes During Transition to SOA

T. Peplow
iMeta Technologies Ltd
Phi House
Enterprise Road
Southampton Science Park
Southampton, SO16 7NS, UK

Dr B. J. Dupée & Prof. M. R. Ross
School of Computing & Communications,
Faculty of Technology,
Southampton Solent University,
East Park Terrace,
Southampton, SO14 0YN, UK

Abstract
Software development organizations have, under competitive pressure, government dictat and financial necessity, spent considerable effort in preparing for and achieving mature processes.

After several years of process improvement (reaching CMMI level 3 in all areas and level 4 in some) the organization requires a new architectural model that encapsulates the changing needs of its IT systems. It has been decided that a Service Oriented Architecture (SOA) will be deployed to leverage both existing and new system assets and enable the exposure of business capabilities over the internet through the use of web services. However, the organization wishes to maintain its current level of process maturity.

This paper analyses the potential impact of changing the system architecture and outlines an action plan for ensuring that the current level of process maturity is maintained.

Keywords: Service Oriented Architectures, Process Improvement, CMM

1. Introduction

Software architecture defines the system characteristics that are realised in each development of the organization’s products. The implementation of common design decisions throughout the software product creates the conceptual integrity so fundamental for success [4]. Therefore, the simplicity and completeness of how the architecture meets the system requirements contributes to the quality of the products delivered.

Service Oriented Architecture (SOA) is a contemporary model with the goal of bringing together IT assets and investments to drive greater value into business through integration [11]. The services encapsulate business capabilities (for example “Give Mortgage Quote”) that are consumed via messages that define the interface (or contract) of the service [10]. The organization has become aware of the potential benefits of changing their architectural model to SOA. However, as the architecture is so fundamental to the quality of the products delivered it is essential that any changes are properly controlled.

The implementation of a new architecture will influence the team’s approach to software development resulting in changes to the software process. For example, adopting SOA requires additional activities that deal with the analysis and design of the service contracts [8]. The architecture may create opportunities for improved working practices. For instance, it may enable more effective team collaboration and simplify previously complex tasks such as system integration.

These opportunities created by the new architecture (i.e. consuming existing third party services) introduce additional issues that the process must consider, e.g. security or acquisitions [12]. It is also important that the new architecture is sustained within the organizational context by adapting to new business requirements and maturing through previous project experiences [3].

An organization that has achieved level 3 of CMMI has, at least, identified and defined its organizational processes [1] — it has quality certification largely based on those defined and standardized processes — but will not necessarily have in place mechanisms for wholesale process modification and performance measurement. However “strengths, weaknesses and improvement opportunities for the organization’s processes are identified periodically” [1]. It is therefore not an insignificant issue as to the impact of transition on ensuring process maturity.

The transition to an SOA should be viewed as a process improvement activity and be supported by the existing institutionalized capabilities that are in place within CMMI level 3 organizations [6]. This ensures that SOA is the correct architectural choice and it is implemented correctly with the impact on any key process areas being addressed. Therefore, CMMI adds value to the process of architectural change, minimising the risk of decreased product quality and maximising the return on investment. The fact that the
organization is CMMI level 3 should viewed as a benefit to change, not as a constraint.

The remainder of this paper suggests a method for implementing a transition to SOA within the organization while highlighting the process areas that could be most affected including recommendations as to how to proceed. The recommended approach uses the CMMI framework that already exists within the organization as part of continual process improvement.

2. The Action Plan

All of the organizational areas have reached the defined maturity level (level 3) embodied by a set of rigorously defined organizational standard processes that are well characterized and understood, which are described in standards, procedures and tools, with each project using a tailored instance of this organizational process [5].

The generic practices deal with establishing and maintaining the defined processes and the collection of improvement information derived from planning and performing the processes. These result in the proactive management of the process [1, 5]. Maturity level 4 extends level 3 creating a quantitatively managed process [1]. Therefore, any changes resulting from the implementation of a SOA must meet these criteria presenting a challenge considering that neither measures nor project experiences exist to support organizational best practice.

2.1 Organizational Process Focus

The ideal starting point for implementing this change is the organizational process focus (OPF) process area (see Figure 1), which deals with the identification and implementation of an organization’s process improvement activities [1]. The organization has already identified that SOA is a required improvement, and the assumption is that it was determined as an improvement opportunity from OPF.

The key practices within the second goal of OPF (plan and implement process improvement activities) address many of the generic points relating to the implementation of process improvement [5], which enables this paper to focus on the implementation of SOA.

The OPF suggests that once action plans have been established they are implemented through the use of pilots and action teams [1] to test selected process improvements [7]. However, what would add considerably more value is if the pilot project were part of an incremental rollout of service oriented principles. Therefore, these pilot projects facilitate the collection of best practices to be embedded within the organizational processes and the identification of measures to contribute to the organizational measurement repository.

The use of a SOA consultant during these pilot projects will help contribute existing experience ensuring industry-wide best practices are deployed and common problems are avoided. The expert knowledge provided by a consultant can also be used to help create project plans and estimates for early SOA projects ensuring that project controls are not lost and can help define the specific tailoring guidelines. These early projects can then be used as benchmarks to measure further process improvement. The experiences of the team, coupled with the knowledge of the consultant, can be used to create organization specific training material ensuring that all members of the team are given adequate training in the application of the new architecture and are informed of all process changes.

However, consideration also needs to be made about the quality of the architecture implemented and how this can be facilitated through the early pilot projects. The concept of SOA is based on the premise that the data within an organization is subject to the least change, followed by the capabilities of that organization [10]. Services are defined that realise the business capabilities consuming a common data model, with the value coming from the durability of services [11].

Consequently, the quality of the definition of a canonical data schema and service portfolio that best represent the business is pivotal to the quality of that SOA. Therefore, the pilot project should include a phase that develops these items. The process should have a mechanism for keeping these up to date in line with organizational change.

2.2 Integrated Project Management

There are three SOA delivery strategies to consider. The first is top down analysis of service requirements starting with detailed business analysis defining an enterprise wide ontology which is aligned with business process models leading on to the analysis, design and implementation of services. The second is a bottom up approach, where services are analysed and designed in relative isolation to meet
application-centric requirements quickly. The third is an agile approach that iteratively implements top-down analysis in line with the delivery of pertinent business requirements as services [8].

Since it is important to verify the implementation of a SOA, it is recommended that an agile approach is taken. This will also employ all aspects of SOA making available the most data and process information. It will also help create services of sufficient maturity to be considered process enabled where applications delegate all process control to the SOA creating the highest return on investment through software agility [10]. This facilitates higher levels of service maturity supporting future sophisticated third party integration extending value chains across organizational boundaries [2]. It will need considering within Supplier Agreement Management (SAM) with security and service availability being considered in the context of the Risk Management (RSKM) process area (see Figure 2). If the organization finds that a large number of projects require integration to third party products or services then the Acquisition Extension should be considered because its goals focus on managing the acquisition of products and services from third parties [1].

Service contracts are central to the development of a SOA [10] and impact heavily on the requirements development (RD) process area (see Figure 5) surrounding the analysis and definition of message oriented interfaces. The verification of these interfaces is also important, therefore it requires consideration within the verification (VER) and Product Integration (PI) process areas (see Figure 5).

The Web Service Definition Language (WSDL) created is the definition of the interfaces, with messages that consume data items defined in schema, XSD, creating the operations that services expose [8]. The WSDL is a fundamental artefact of SOA that must be under adequate configuration control and be available to all teams, promoting service reuse, in a maintained service repository. The issue of maintaining these WSDL definitions needs careful management and it is vital that the consumers of interfaces are known to ensure all parties are notified if the interface is changed. This is particularly important if the interface is exposed to third parties. Configuration management may also become difficult as the same service contract may be used by several projects running concurrently.

The decision analysis and resolution (DAR) process area (see Figure 3) should be used to assess and manage interface changes and make decisions regarding interface versioning. Therefore, it is required that measures of the impact of changing interfaces to support these decisions are recorded and kept. It is also recommended that the service contract should drive the project (i.e. plans and estimates are created for the delivery of service contracts) [10]. This therefore impacts the project management process areas as projects are now planned and managed as iterations of service contacts. The coupling between service contracts, process assets and project management creates a process that can easily be measured, monitored and controlled.

Service contracts also facilitate the concurrent development of services [10]. Consequently, there needs to be consideration regarding the management of concurrent teams. It is recommended that an SOA Board is created with the aim to use service contracts to coordinate concurrent development [10] — this will also support effective configuration management. Concurrent teams can be further supported within the CMMI framework by implementing the Integrated Product and Process Development (IPPD) extension, with the addition of two goals to Integrated Project Management (IPM) (see Figure 4), helping manage teams in a project context (for instance distributing requirements) and Integrated Teaming (IT) intended to create and sustain integrated teams [1]. IPPD will help ensure that people are brought together to gain maximum benefit from collaboration through effective planning, communication and train-
2.3 Software Engineering

The new concepts presented within SOA will need to be considered in the Technical Solution (TS) process area (see Figure 5). In particular, thought should be given to new design methods that should be applied to services, e.g. functional decomposition techniques to identify services and their boundaries and data modelling to capture interface specifications [8]. As the level of service maturity increases the organization will be in position to use Enterprise Application Integration (EAI) [2] to integrate purchased systems into the application landscape [11]. This, therefore, also needs careful consideration within the Technical Solution process area.

The Product Integration (PI) process area (see Figure 5) will also be influenced as loosely coupled services and tightly coupled components will be integrated. Both the TS and PI process areas will be affected if the organization begins orchestrating services as part of configurable model driven business processes (using a business process engineering tool, for example, Microsoft BizTalk) — there will be further design requirements, for instance long running transactions and compensating actions [10], with changes to the way services are integrated and tested. It is recommended that, as the organization matures in its use of SOA, that the Organizational Process Focus (OPF) process area is used to ensure changes are properly analysed and implemented.

3. Conclusion

The organization has taken the decision to adopt a Service Oriented Architecture and have acknowledged that work is required to ensure their current level of maturity is maintained. The paper has discussed the importance of the architecture on software quality and identified that, if the implementation of a SOA is to be successful, the CMMI framework must be used to support the change as well as adapting to the new requirements created through SOA.

The action plan suggests that Organizational Process Focus (OPF) pilots are executed to test the validity of the architecture as well as analysing the requirements of the new software process and facilitating the definition and capture of initial process metrics. These pilot projects should be led by a SOA consultant who provides guidance and industry best practice while helping with organizational acceptance and training. The action plan also suggests a delivery strategy that ensures the quality, durability and maturity of the SOA while considering the process, measurement, supplier agreement and risk management required by CMMI, highlighting that future integration requirements may benefit from the implementation of the acquisition extension.

The importance of the service contract is discussed in context with changes within the requirements development process area regarding the definition of message oriented interfaces while considering the configuration management requirements and the support available from additional metrics to be collected and used within the decision support and resolution process area. The plan advises that the service contract is used to drive projects with the support of a SOA Board as well as the use of the Integrated Product and Process Development (IPPD) extension to assist with the development of concurrent teams.

Considerations regarding changes to software design are indicated within the Technical Solution process area and changes to the way products are integrated in the Product Integration process area. The future applications of SOA are introduced proposing that these are implemented through OPF when service maturity increases and the initial changes...
are institutionalized into the organizational processes, thus delivering a controlled staged rollout of SOA into the organization.

In conclusion, changing the architecture does not impact considerably on the process of delivering software as the activities remain the same. However, there are benefits in tailoring the software process specifically for a SOA. This paper indicates that a SOA can be implemented into the organization, retaining the current level of process maturity and product quality, if a controlled approach, such as the one suggested, is undertaken. The risks of implementing the architecture incorrectly are mitigated through the use of the existing processes, with CMMI providing many supporting frameworks. The initial investment, ensuring that the architecture is deployed using a process which supports it, will maintain the high quality of the products delivered, given that it is the process that governs the quality of the developed software [9].

References

Internal Marketing to Elicit Conceptual Needs for Internal Software

Mark Elkins  Prof. Margaret Ross  Geoff Staples  Dr. Brian Dupée
School of Computing & Communications,
Faculty of Technology
Southampton Solent University,
Southampton, SO14 0YN, UK

Abstract
Within software requirements elicitation the most crucial element is arguably identifying the conceptual need/problem to justify a software project. Employees are important stakeholders/users who have significant potential to identify valuable internal software projects. The problem considered by this paper is: What value can internal marketing have within organizations to identify conceptual needs for internally used software? The Chartered Institute of Marketing define internal marketing as “The process of eliciting support for a company and its activities among its own employees, in order to encourage them to promote its goals”. Discussion based on a literature survey suggests that key components from an internal marketing approach can be usefully mapped to requirements elicitation/engineering. This is backed up by initial findings from a pilot survey. Existing requirements engineering practice is considered in relation to this approach.

Keywords: Requirements Engineering, Internal Marketing

1. Introduction

The starting point for requirements engineering is identifying the conceptual need/problem upon which a software project is to be based. This is arguably the most crucial [1, 2] element in the life cycle of a project. Identification of initial need is the foundation that all software projects are based and hence the importance this has on the quality of software produced [2, 3]. Problems at this stage might result in software projects that fail [4, 5, 6], lose money and/or services [7], and that do not reflect the real needs of a customer [8, 6]. Indeed a “...lack of knowledge among technologists about what business users want” was mentioned in a “...review of 1,000 projects by the UK Office of Government Commerce (OGC)...”, which “...found that technology was one of the least likely reasons for a project to fail” [1].

Employees within an organization are important users and stakeholders who have the potential to identify valuable internal software projects. The problem considered by this paper is: What value can internal marketing have within organizations to identify conceptual needs for internally used software? The definition of such software for the purpose of this paper being an internally identified conceptual need for any software that will be used within the organization regardless of whether the software is bespoke, open source, or Commercial-Off-The-Shelf (COTS) and/or developed and/or customised internally and/or externally.

Internal marketing [9, 10] is defined by the Chartered Institute of Marketing as “The process of eliciting support for a company and its activities among its own employees, in order to encourage them to promote its goals” [11]. Although there is no universally accepted definition of internal marketing [9, 12], a literature review of the topic identifies certain key attributes/components that are thought to belong to it. All these attributes, shown in Table 1 below, can probably be directly mapped into an requirements engineering context. For instance the role of knowledge management in “elicitation and elicitation technique selection” [13]. Internal communication is an important example because without it problems and therefore requirements can remain unknown. “One of the most important goals of elicitation is to find out what problem needs to be solved, and hence identify system boundaries” [14]. Communication at the start between stakeholders is thought to result in more satisfaction with systems and a demand for fewer corrections [15] that supports other evidence that early user involvement is beneficial for software project success and requirements quality [16].

Marketing is concerned with the identification and satisfaction of customer needs [17, 18, 19]. Requirements engineering is also concerned with this [13]. Indeed there is some literature that considers how marketing might be used in an requirements engineering context [20, 21]. Various approaches are used in marketing to identify a need for a product and/or service. Market research is one method [18, 19]. Another is to let the market know that a product
or service exists by using the promotional mix of advertising, personal selling, sales promotion and public relations [17], which may trigger and then satisfy a need. Product, promotion (promotional mix), the price of a product or service, and the place where it is sold are the traditional components of the marketing mix [17]. This has been extended [22] to include people who for example deliver a service, process being the organization of business processes, and physical evidence such as product packaging. There are those [23, 24] who argue that due to technological and other changes the traditional marketing mix is out dated. Internal marketing [9, 10, 12] is considered part of that change. Its basic premise is that successful marketing requires the effort of all employees and not just those in the marketing department.

Those unfamiliar with the theory of internal marketing may define the term in a narrow way to mean literally internal marketing of X by Y within their organization. Indeed there is literature within an Information Technology (IT)/Information Systems (IS) context that has considered internal marketing where X equates to software and Y to the IT or IS department [25, 26]. Much of this could be considered as essentially applying the label of internal marketing to the internal use of the promotional mix. Other literature offers specific illustrations of internal marketing by the IT/IS department as making internal use of the 7p’s of the marketing mix [27, 28]. A main problem that might occur when using this narrower definition of internal marketing is that considerable conflict within an organization might happen where each different department seeks to promote/market its own agenda. This may or may not be at the expense of other departments within the organization or have benefits for potential and existing external customers.

Other definitions of internal marketing focus on a more “holistic” approach such as “The objectives of internal marketing are... to get the commitment of the employees to the strategies and tactics of the firm and to create an environment where they feel motivated for a customer-orientated and marketing-like performance” [29]. Within this broader approach the whole organization is encouraged to use internal marketing in a team effort to provide customer satisfaction [30]. Emphasis is put on matters such as training, motivation, and the general well-being of employees in the belief that if employees are looked after they will look after customers [30]. This applies not only to front-line customer facing staff, but also to the rest of the organization whose efforts go into producing the goods and services offered to customers.

“Requirements elicitation is concerned with where software requirements come from and how the software engineer can collect them. It is the first stage in building an understanding of the problem the software is required to solve” [31]. It could be argued that the “...commitment of the employees to the strategies and tactics of the firm...” [29] within an “holistic” approach towards internal marketing will create a climate where employees are more likely to assist in this. For instance a good working environment may lead to better internal relationships leading to an open exchange of ideas offering ways to understand and solve a problem/conceptual need in more detail.

The rest of this paper now considers in more detail a variety of ways that internal marketing might be used within organizations in a requirements elicitation context to identify conceptual needs for internally used software.

2. Discussion

Table 1 below lists a number of internal marketing attributes identified through a review of the literature. Some of these such as internal communication, internal relationships, and open exchange have already been mentioned in the introduction. Another of these innovation is a key driver to competitive advantage and business survival in the marketplace [32]. Therefore an organization has a vested interest in stimulating and making use of the creative ideas of its employees. It has been argued that the requirements process can be assisted by encouraging creativity to maximise the number of ideas that can be developed into requirements for innovative products [33]. This could equally apply to generating concepts for innovative internal software projects.

<table>
<thead>
<tr>
<th>Table 1. Internal marketing attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Innovation</td>
</tr>
<tr>
<td>Internal communication</td>
</tr>
<tr>
<td>Knowledge management</td>
</tr>
<tr>
<td>Linking human resources strategy to customer needs</td>
</tr>
<tr>
<td>Open exchange of ideas</td>
</tr>
<tr>
<td>Employee retention</td>
</tr>
<tr>
<td>Internal relationships</td>
</tr>
<tr>
<td>External relationships</td>
</tr>
<tr>
<td>Staff awareness of marketing campaigns</td>
</tr>
<tr>
<td>Processes</td>
</tr>
<tr>
<td>Service standards</td>
</tr>
<tr>
<td>Service measures</td>
</tr>
<tr>
<td>Using corporate strategy to achieve competitive advantage</td>
</tr>
<tr>
<td>Product/Service branding</td>
</tr>
<tr>
<td>Customer focus</td>
</tr>
<tr>
<td>Employee development</td>
</tr>
<tr>
<td>Corporate vision</td>
</tr>
<tr>
<td>Staff awareness of product/service portfolio</td>
</tr>
</tbody>
</table>

It is probably possible to map all the attributes shown in Table 1 into an requirements elicitation context, because
they are all have relevance to the business needs of an organisation.

There are several possible permutations of how requirements elicitation might make use of internal marketing. For example an entire organization could adopt the holistic approach identified in the introduction above, an IT department alone could adopt it, or requirements engineers/analysts or developers might make use of internal marketing.

2.1. Internal Marketing - Holistic Approach

It could be argued that the adoption of an holistic approach to internal marketing by an entire organization would align of all the parts of that organization with each other. A default expectation is that this would have the effect of aligning IT with the rest of the organization. This in turn may make it easier to identify conceptual needs for internally used software. For example within a requirements elicitation context good internal communications and relationships between employees may mean that they convey the business needs of the organization to the IT department on an ongoing basis and vice versa. Such information forms the basis for the creation of use case, scenarios, user stories, and other methods used in requirements elicitation and other parts of the requirements engineering process.

Emphasis “...to work as a team to provide customer satisfaction” [30] may help to overcome some of the issues caused by social and political factors prevalent within organizations. For example software that “...has proved unsatisfactory because it has stressed the requirements of one group of stakeholders at the expense... of others” [31].

Team effort generated by the holistic approach may also help to prioritize conceptual needs for software. Such prioritization enables “...trade-offs in the face of finite resources...” [31]. One study concluded that “Requirements prioritized by stakeholders drive successful RE teams” [34].

In theory the greater the maturity level of internal marketing the greater the number of multi-skilled employees within an organization due to the interaction of the attributes in Table 1 causing a corresponding spread of knowledge across the organization. This might lead to a move away from demarcation lines between departments and job functions to a situation where for example someone who used to work solely in a marketing environment now has enough knowledge to work in an IT, accounting, legal, or Human Resources (HR) environment. Indeed there is some evidence of employees involved in “...cross-functional teams focusing on customer issues...” and “...the rotation of personnel across traditional functional boundaries...” as a result of internal marketing [12].

Within a software requirements engineering context such an employee would have considerable knowledge that could be used to identify current and future software business needs. This might offer some assistance to the often cited problem that users know what they want but are unable to communicate it properly [14] to IT personnel because such an employee maybe able to view a system as both user and designer. The following statement indicates such skills are in short supply: “Hybrid managers are people who can cope with the business, aren’t afraid of the technology – to the extent that they have been there, done that – and have the personality to talk to a wide range of people, be credible to users and technologists, and get their cooperation” [1]. This statement goes on to mention that these “...hybrid managers... are in short supply” [1]. “Multidisciplinary training for requirements practitioners” has been described as a one of the “...major challenges for Requirements Engineering in the years ahead” [14].

A main drawback of the holistic approach is that the full potential on offer for requirements elicitation is reliant on the whole organization adopting it.

2.2. Internal Marketing – IT Department

An IT department could adopt the holistic approach outlined above even if the rest of the organization has not. This would at least offer the potential that the IT department is internally aligned and that it was working in a customer focused way to gain knowledge about the needs of its customers.

Another form of internal marketing is where the IT department literally internally markets its services to the rest of the organization [25, 26]. Making other employees aware of what IT products and services are available may for instance result in the capture of previously unknown needs for software already in use in another part of an organization. The interaction from this kind of internal marketing between the IT department and others may mean IT gains a better understanding of internal customers/users needs [26]. Some evidence indicates that “involving users and customers as the source of information is related to project success”[16]. “The most successful teams always involve customers and users in the RE process and maintain a good relationship with stakeholders” [34].

Using marketing methods internally to improve the image of the IT department may mean other employees are more aware of the functions it performs [26]. For instance the use of public relations from the promotional mix by organizing informal lunch-time events involving other employees “...can give IT people a chance to make a presentation and show the value of their work......” [26]. This may also help to increase “buy-in” to the requirements elicitation process.
2.3. Internal Marketing – Analysts / Developers

An analyst has been defined as “Generically, any individual who performs elicitation. Also known by many other names, e.g., requirements engineer” [13]. Many of the attributes in Table 1 have direct relevance to the work carried out by an analyst/requirements engineer. For instance “The requirements engineer must possess both the social skills to interact with a variety of stakeholders, including potentially nontechnical customers, and the technical skills to interact with systems designers and developers” [14]. This emphasizes the need for good internal relationships and communication as well as a customer focus.

A relatively recent study found that “...developers become more aware of their limited knowledge of user issues when they start to visit users” [16]. The success of such visits will to some extent rely on the way they market themselves. Good knowledge of user issues may result in less problems throughout the requirements engineering process. This in turn may reap time and cost savings within a software project brought about through the need for less development iterations [16].

For a developer or analyst the internal marketing attributes shown in Table 1 might act as a useful check list to grasp the depth of knowledge and thoughts a user or other stakeholder has about the organisation. This might for example elicit/identify issues with internal processes or knowledge management that could justify a software project.

2.4. Potential Barriers

Problems that act as barriers to the use of internal marketing have the potential to affect the value it has to offer in the identification of conceptual needs for internally used software. This section highlights some potential problems that may have this effect.

There is a de facto tendency in written material about internal marketing to refer to permanent employees of an organization and never mention other kinds of worker. An important issue that may impact on the success of internal marketing is the use of contract and other staff who work for an organization alongside permanent staff but who are nevertheless not employees. For example it is not uncommon for such staff not to be re-employed by an organization in the UK after a successful software project is completed [35]. Although such staff might be treated as if they were permanent employees for the duration of their contract it is perhaps questionable how well the internal marketing attributes set out in Table 1 can be applied to them as intended by the theory and possibly the spirit of internal marketing. For instance they might not want to participate in an open exchange of ideas because by giving away their knowledge they might consider this reduces the need for the organization to employ them. Also dependent upon the duration of their contract they might not have the chance or motivation to fully gain the knowledge implied within some of these attributes such as staff awareness of marketing campaigns or the product/service portfolio.

The “domino effect” of an holistic approach to internal marketing might be considered both an advantage and disadvantage. If the attributes in Table 1 are misused then the whole organization is likely to be affected in an acute perhaps even fatal way. For example an organization might seek to use “candidate administration software” to sort through the curriculum vitae (CV) of prospective employees, which has been sent on line [36]. The use of such software might mean the organization is better able to deal with the recruitment process and identify valuable prospective employees. However if the criteria for selecting suitable candidates to join the organization is faulty due to for instance poor requirements capture then the organization could find itself hiring employees without the right skills. Developers may be less keen to involve users in the requirements elicitation process whom they perceive lack the right skills and knowledge.

When an organization is involved in a takeover or merger issues may occur. For instance the culture and make up of organization A that has merged with organization B may be totally different. This might make it difficult for organization A to carry on with the way it uses the attributes shown in Table 1 because organization B has not been using these attributes in the same manner. Until conditions stabilise and a common approach to internal marketing is adopted parts of the new organization might miss valuable opportunities for software projects.

3. Conclusion

Since communication at the start between stakeholders is thought to result in more satisfaction with systems and a demand for fewer corrections [15], and the clear requirement of software systems to reflect the real needs of those stakeholders [1, 2, 6, 8], it is clear that a mechanism for this is required. Results of a pilot study tend to confirm that at least some firms consider internal marketing (and marketing [21]) as such a mechanism for identifying initial needs for internal software. It is not yet apparent that all internal marketing attributes as given in Table 1 are directly linked to elicitation but it might be possible to better map these across for practical use in requirements elicitation/engineering.

Another facet of internal marketing considered by this paper is the way that it might offer an environment that gives assistance to certain perennial problems in require-
ments engineering. For example emphasis “...to work as a team to provide customer satisfaction” [30] may help to overcome some of the issues caused by social and political factors prevalent within organizations. For instance unsatisfactory software that “...has stressed the requirements of one group of stakeholders at the expense... of others” [31].

In theory the greater the maturity level of internal marketing the greater the number of multi-skilled employees within an organization due to the interaction of the attributes in Table 1 causing a corresponding spread of knowledge across the organization. Indeed evidence exists of employees involved in “...cross-functional teams...” and “...the rotation of personnel across traditional functional boundaries...” as a result of internal marketing [12]. This might help overcome the current shortage of staff who have both the business and technical skills needed for requirements engineering [1, 14].

There are some possible problems that act as barriers to the use of internal marketing. For example it is not clear from a review of the literature what impact the use of those who are not permanent employees of an organization has on internal marketing. Another possible problem is the serious implications the misuse of internal marketing may have on an organization. Finally if internal marketing achieves nothing more than reinforcing focus on business needs within the requirements elicitation/engineering process then it may have justified its value to elicit conceptual needs for internal software.

References


Risk Identification – Challenges and Cure

A. Sana Khan¹, B. Syed Jawad Hussain²

¹Department of Computer Science, COMSATS Institute of Information Technology, Abbottabad, N.W.F.P, Pakistan
²Comsoft, COMSATS Institute of Information Technology, Abbottabad, N.W.F.P, Pakistan

Abstract - Risk management is the process of identification, analysis, planning, mitigating and monitoring risks. Successful risk identification and management is a very critical issue for a successful software project. In time identification of risks, and improved decision making regarding risk management are some of the factors that contribute toward successful software projects. This paper presents a detailed analysis of software project failures and how poor risk identification is contributing to the high rate of failure in the software industry. A Risk Data warehouse is proposed as a solution as it is enabling quick and efficient access to the required information and better decision making in several other industries. An industry survey on Risk Management and Project Failure in the Pakistani software industry is also presented.

Keywords: Risk Management, Risk Identification, Decision Making, Project Failure, Data Warehousing

1. Introduction

A software project is considered successful if it meets the requirements of its users and is completed with in time and budget [1]. There are few projects that prove to be successful in delivering the required functionality and performance with in time and budget but there are many projects that end in total failure, some are cancelled before they are completed and some are not completed with in promised time and budget [2], [3].

According to CHAOS report 2004, only 29% of software projects were delivered close to their time and budget. 53% of the projects were challenged, and there were another 18% projects that were total failures [4]. This situation proves that risk management is a serious problem for software projects as risk management is aimed at reducing the failure rate [4], [5]. Definitely some potential risks are maturing into actual problems and derailing software projects. The reason behind the constant use of unimproved IT services is also the poor risk management [6].

To tackle this issue there is a need of methodology that must facilitate the in time identification of project risks and also provide significant help in improving risk management decisions. Data warehousing is currently facilitating solutions for timely information retrieval and improved decision making in different domains [7], so this facilitation can also be utilized for software risk management. What data warehousing is facilitating in the other domains is the key motivation for its utilization from the perspective of risk management. For this purpose, different case studies regarding the data warehousing solutions in different domains have been studied. Industry survey regarding the current status of risk management and project failures has also been done. The next step is to determine how the objectives of in time risk identification and improved risk management decision making can be achieved through data warehousing.

2. Related work

Research shows that risk management is becoming acknowledged as an essential element in the software industry for reducing the failure rate [4], [8], [9], and [10]. It is the process of identifying risks, performing risk analysis and then implementing plans to control them [3]. Miller J et al term risk management as a real challenge [11]. Unmanaged risks or lack of proper risk management is one of the reasons that causes software projects failure [12].

Different methods or techniques are used to perform steps involved in risk management. For risk identification commonly used techniques are brainstorming, interviewing, documentation reviews, cross functional teams, checklists, assumption analysis, diagrammatic techniques [13] and SWOT analysis [14]. Methods that are commonly used to perform risk analysis are risk probability and impact, impact risk rating matrix, project assumption testing, data precision ranking, interviewing, sensitivity analysis, decision tree analysis and simulation. Methods that help in performing risk planning of a software project are avoidance, transfer, mitigation and acceptance. Project risk response audits, periodic project risk reviews, earned value analysis, technical performance
measurement, additional risk response planning are the techniques that are used to perform risk monitoring [14]. Reasons, impact, examples of software project failures and measures to reduce them have been described in various books [15], [16]. Researchers have proposed different techniques, frameworks and methodologies to enhance the effectiveness of software risk management [17], [18], [19], [20] & [21].

Different factors are identified and analyzed that contribute to effective risk management or that effect project success [8], [10] and [22].

![Organizations without Proper Risk Management Methodology](image)

**Figure 1 – Organizations without Proper Risk Management Methodology**

Different concepts and opinions are presented regarding risk management on the basis of industry surveys [23]. Sheila Wilson (1998) focused on the failure of software development projects from the perspective of human factors by survey findings gathered from already performed case studies [24]. Bernd Freimut et al (2001) presented the results from a case study that introduced the RiskIT method, into a large German telecommunication company and results showed that the RiskIT method adds value to the project by performing risk management comprehensively [25].

Main focus of this research is in time risk identification. Risk identification is an important step of risk because it ensures which risks commonly affect the project and document their characteristics [13, 26]. It is important to ensure that at the risk identification step the large number of risks are identified, as risks that are not identified at this step may not be analyzed and treated in later steps, and advocates of software project risk management assure that by properly identifying and analyzing threats to success, measures can be performed to minimize the chance of failure [26].

Purpose of utilizing the data warehousing for software risk management is the in time identification of risks and improved decision making, as it is currently providing this solution (timely information and better decision making) in various domains. For example it is providing its services in business for supporting strong customer relationship management, business performance management and managerial decision making [27], [28], [29], [30], medical industry [31], & academia [33].

### 3. Motivation

Data warehouses have emerged to successfully support timely information and improved decision making in many domains like academia [32], [33] health care [34] business [35], [36] and banks [37]. It is facilitating decision support by gathering data from distributed sources and extracting information from this data. The main motivation to incept this methodology for software risk management is to bank on the facts that data warehousing is currently facilitating in other areas and software risk management is in dire need of such facilitation.

In-time risk identification and improved decision making are the two issues that are to be mainly focused by exploiting the benefits and solutions of data warehousing for effective risk management. Risk management is the process that facilitates better project outcomes by providing confidence for better decision-making and the purpose of utilizing data warehouse is to make effective the decisions related to software risk management. This requires obtaining the right information at the right time. For this purpose, it is necessary to identify all the risks associated with a project at the right time, because if the risks are identified at the right time, decision can be made at the right time to mitigate them.

### 4. Survey of the software industry

The objective of industry survey was to explore the fact that software projects are failing and poor risk management is the major contributor toward their failure.

<table>
<thead>
<tr>
<th>Table 1 – Industry Survey</th>
</tr>
</thead>
<tbody>
<tr>
<td>Facts</td>
</tr>
<tr>
<td>Total Projects Studied</td>
</tr>
<tr>
<td>Total Number of Projects Failed</td>
</tr>
<tr>
<td>Projects failed because of poor risk management</td>
</tr>
<tr>
<td>Total number of organizations</td>
</tr>
<tr>
<td>Organizations using risk mgmt methodology</td>
</tr>
<tr>
<td>Organization not using any methodology</td>
</tr>
</tbody>
</table>

Data was collected through the use of questionnaires, which aimed at achieving the above objectives. Experts and managers from the Pakistani software industry were asked various questions.
First two questions were about whether the proper project management and risk management methodology is in use in their organization, total number of projects done and total number of failed projects to determine that how many projects were failing if proper project management and risk management methodology were not used and how many organizations are using proper risk management methodology. Some factors of software projects failure were presented including poor risk management to get the figure about the projects that were affected by these factors. Then some questions were asked regarding risk identification and mitigation planning to know about the current status of risk management in a specific organization. Results obtained on the basis of collected data are presented below in Table 1 and Figures 1, 2 and 3.

5. Proposed Idea

In [38] according to Boehm (1991) the problems of failed software projects could have been reduced, if their high risk elements had been identified and resolved earlier. Effective risk management is acknowledged as an important tool to reduce the software project failure rate [38]. Risk identification is the starting point for effective risk management and accurate risk identification is the first step to be carried out for effectively risk management [39].

Various techniques are in use for risk identification. Techniques that are commonly in use are interviews and brain storming. Some other techniques are diagrammatic techniques; Delphi technique checklists etc and research shows that there is no single best method for risk identification [13] and most of these techniques are based on the assumption that managers have the necessary knowledge to be aware of all significant risk factors [10]. Tools that are commonly used for software risk management are RiskIT, Risk Radar, Risk Matrix, Risk Nav, Risk Track, @Risk and Risk Register. RiskIT is a tool that automates the risk management by identifying risks and proposes actions to control them. In spite of the automation of risk management process still the software projects are getting failed.

There is need for a technique that automates the risk identification process by providing suggestions for different risks. Data warehouse will be used to input the information about risks that are identified in different software projects and measures that are taken to control them. Whenever a new software project is started, information can be extracted about the risks that were identified in the pervious projects having the same nature and type. Proposed system will allow the software project team to be aware of the risks that may be identified in a software project at the right time, and if they experience any new risk, it will be updated into the data warehouse to benefit the future projects carried out by the same organization or some other organization.

6. Conclusion

This paper presents the research in progress, to enhance the effectiveness of software risk management through data warehousing. First, different data warehousing applications and case studies in different domains are studied and analyzed to determine that software risk management is also in need of such facilitation. The initial questionnaire traced the facts that projects are getting failed, and poor risk management is the major contributor to their failure. Next step is to collect data regarding the risk identified in different software projects (nature, type, technology and other parameters), how they are mitigated to determine how this information can be utilized through data warehousing to benefit the future projects by enhancing software risk management effectiveness. The last step that is included in the scope of this research is to devise the architecture for such data warehouse to incept it for software risk management.
7. References


[24] Sheila Wilson, Failed IT projects (Human Factor), University of Maryland Bowie State University, 1998


Customer-Centric Data Warehouse – an Architectural
Approach to Meet the Challenges of Customer
Orientation, Proceedings of the 36th Hawaii International
Conference on System Sciences (HICSS’03), 8 pp, 2003
Far(2005) Building data warehouses with incremental
maintenance for decision support, CCECE/CCGEI,
Saskatoon, May 2005 IEEE
warehouse, Proceedings of the IDEAS Workshop on
Medical Information Systems: The Digital Hospital
(IDEAS-DH’04) pages 8-14 , 2004
warehouse for an academic medical center. By Jonathan
S. Einbinder, MD, MPH; et al
[33] www.sun.com/products-n-solutions/edu
/casestudy/pdf/eScholar.pdf ; Case Study: Data
Warehousing Solution for Western New York State
Regional Information Centers
[34] http://www.dmreview.com/issues/
19980301/696-1.html
[35] Dan Pollack (1997), A Large Scale Data
Warehouse Application Case Study: System
Administration Conference: Proceedings of the 11th
USENIX conference on System administration
p=443594
v07n01/Features/CaseStudies/BankingOnSuccess.aspx,
Case study: Banking on Data warehouse successes, Data-driven culture impacts every department within St.George
Bank. By C.A. Doyle
[38] Y.H. Kwak , J. Stoddard , Project risk
management: lessons learned from software Development
environment , Technovation 24, 915-920, 2004
[39] Martin Stevens Project Management Pathways,
Association for project Management, 2002
Integration of Proposed Software Risk Model in Waterfall Process

Prof (Dr) P K Suri\(^1\) and Manoj Wadhwa\(^2\)

\(^1\)Professor, Department of Computer Science and Applications, Kurukshetra University, Kurukshetra, Haryana, India
\(^2\)Assistant Professor & HOD, Department of Computer Science and Engineering, Echelon Institute of Technology, Faridabad, Haryana, India

Abstract - Today software development projects fail to be delivered on time, within budget and with desired quality. Many software risk methodologies have been proposed for mitigation of risks but they do not fulfill the aim of software society because problems still exist. Herein, an attempt is made to propose a software risk model and implement in Waterfall process. It may have advantages over other models and provide opportunities to developers to use this model in Waterfall process.

Keywords: Software risk methodology, Waterfall process

1 Introduction

Today, software development has been the troubling technology and involves many risks. Risk identification, assessment, tracking and mitigation of these risks is very cumbersome task. For doing this, integrate the risk management in the software development planning. Risk Management is that area which identifies, categorizes, assesses, tracks and controls the risks [6]. Several software risk methodologies have been proposed for mitigation of risks. Barry Boehm [7] proposed risk management methodology in conjunction with risk driven spiral development model. It offers a six steps risk management process that comprises two main steps:

- **risk assessment** consisting of
  - identification of those risks likely to cause problems
  - analysis to determine the loss probability and loss magnitude for each risk and to develop a compound risk
  - prioritization to rank the identified risk items according to their compound risks

- **risk control** consisting of
  - management planning to bring the risk items under control
  - resolution to eliminate or resolve the risk items
  - monitoring to track the project’s risk reduction progress and to apply corrective action where necessary

Richard Fairley [10] proposed seven steps for mitigation of risks:

- identify the risk factors
- assess risk probabilities and effects on the project
- develop strategies to mitigate identified risks
- monitor risk factors
- invoke a contingency plan when a quantitative risk factor crosses a predetermined threshold
- manage the crisis by possibly drastic corrective action if the contingency plan fails
- recover from a crisis, e.g. by rewarding personnel, reevaluating schedule and resources

Rockwell [12] has proposed risk management process of five steps based on the principles of Dr. Robert Charette:

- identify risks
- characterize risks
- prioritize risks
- avert risks
- track/control risks

F/A-18E/F [11] has proposed the four steps risk management methodology:

- risk identification
• risk analysis by quantifying the risks and determining overall risk level
• risk planning by developing a list of risk mitigation tasks
• risk tracking via monitoring and updates

The Software Engineering Institute (SEI) has also developed a risk management guidebook for software managers, with steps very similar to Boehm's [9]. In addition to guidebook, SEI has recommended the list of tools for each step in risk management.

2 Drawbacks of Risk Methodologies and Models

The following drawbacks are identified during the study of risk methodologies and models:

• These are related to specific process models e.g. Boehm Risk methodology particularly related to spiral process model
• Every software practice elaborates the large number of steps for mitigation of risks
• It is very difficult to manage all risks during planning and can not resolve all risks at same time.
• Separate tool for each activity become very costly in the development of software

3 Proposed Risk Model

An attempt is made to propose a model which may be named as SDM (Select, Describe, Mitigate). It is a concise risk model with many advantages and is described as

• Select the Risk.
• Describe the Risk.
• Mitigate the Risk.

For this historical project, data is very important and a sample of data is shown in Appendix. This model can be implemented in the form of software by using VB and Microsoft Access or J2EE.

Select Risk
• Business Domain
  o Technology
  o Project Management
  o Customer
  o Natural cause
  o Quality
  o People/Personnel
  o Communication
  o Infrastructure

Diagram 1: Interface 1 of Select in SDM risk model
Select the right option among risks as shown in diagram 1. In this interface, black dotted option is selected; now go to next interface and show all possible descriptions related to that selected risk.

Description of Risk
• Lack of Business/Application knowledge which may impact the quality of work products
• Lack of documentation of requirements may result in under scope
• Unavailability of users during requirement analysis leading to incomplete requirements

Diagram 2: Interface 2 of Description in SDM risk model
After assessment of all descriptions as shown in diagram 2, identify appropriate description i.e. reason due to which risk occurs. In this interface, black dotted option is chosen; now go to next interface which shows the mitigation steps of that risks.

Mitigation Steps for Risk
• Prepare the team’s understanding of the requirements. Get the required document reviewed.

Diagram 3: Interface 3 Mitigation in SDM risk model
Interface 3 shows mitigation step of selected risk with chosen description. It is concise risk management process based on SDM risk model.
4 Implementation of Proposed Risk Model in Waterfall Process

Waterfall Process is linear and sequential process in which each activity takes place one after another. Each phase has distinct goals [25]. It is found that requirements in waterfall process freeze early. Due to this reason, mostly developers ignore this model, so an attempt is made to implement the SDM model based on historical data in waterfall process and use it in each phase. In feasibility phase, use SDM model, identify the risks related to feasibility phase, if any, mitigate risks and get refined feasibility phase. Follow similar steps in all phases as shown in diagram 4. It may have many advantages:

- Developer can identify risks early in waterfall process on implementation of SDM risk model
- Customer gets initially involved in process and can register his/her views.
- There is no need of changing the configuration during each phase.
- It is suitable for small and middle level projects. For developing these projects, large volume of historical data is available.
- It decreases the communication with customers and developer can concentrate on the project.
- It decreases the overhead cost of software project.
- It provides quality to software up to some extent on implementation of SDM model

5 Discussion

It is found that integration of SDM risk model in waterfall process is suitable in small and medium sized projects. It reduces the planning time of each phase. It considers the risks in starting i.e., in feasibility study phase. After implementing SDM, get refined feasibility report regarding development of software and further go to requirement phase, implement SDM risk model, get refined requirements and so on. It limits the communication with customers but provides more opportunity to developer to use its maximum creativity and efficiency. It is dedicated to software quality and gets refined phase within time. It is seen that

- The SDM model provides refined requirements, design, coding and testing which reduce the development efforts
- It reduces development cost of software projects.
- It gets project data for their own data repository of SDM model.
- It reduces development time of software projects

6 Conclusion

It is found that SDM risk model is concise and focuses on Select, Description and Mitigation. It can be successfully implemented in waterfall process. It has advantages to provide software within schedule and requirement can be corrected in early phase. In addition to this, the customer can register his/her views about software, due to which there is no need of software configuration management and SDM risk model is also able to provide quality software.
7 References


Author’s Profile

**Dr. P.K. Suri** received his Ph.D. degree from Faculty of Engineering, Kurukshetra University, Kurukshetra, India and Master’s degree from Indian Institute of Technology, Roorkee (formerly known as Roorkee University), India. He is working as Professor in the Department of Computer Science and Applications, Kurukshetra University, Kurukshetra - 136119 (Haryana), India since Oct. 1993. He has earlier worked as Reader, Computer Sc. & Applications, at Bhopal University, Bhopal from 1985-90. He has supervised five Ph.Ds. in Computer Science and thirteen students are working under his supervision. He has more than 100 publications in International / National Journals and Conferences. He is a recipient of 'THE GEORGE OOMAN MEMORIAL PRIZE' for the year 1991-92 and a RESEARCH AWARD –“The Certificate of Merit – 2000” for the paper entitled ESMD – An Expert System for Medical Diagnosis from INSTITUTION OF ENGINEERS, INDIA. His teaching and research activities include Simulation and Modeling, Software Risk Management, Software Reliability, Software Testing & Software Engineering Processes, Temporal Databases, Ad hoc Networks, Grid Computing, and Biomechanics.

**Manoj Wadhwa** received M.Tech in Computer Science and Engineering from Kurukshetra University, Kurukshetra, India and is currently pursuing Ph.D from Kurukshetra University, Kurukshetra- Haryana (India). Presently, working as Assistant Professor & HoD, Department of Computer Science and Engineering in Echelon Institute of Technology, Faridabad, Haryana, India, he possesses more than ten years experience of Teaching, Research, and Industry. His areas of interest include Software Engineering, Simulation and Modeling and Operating Systems.
### Appendix

List of Top risks with categories & mitigation steps:

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>Risk Category</th>
<th>Risk Description</th>
<th>Mitigation Steps Suggested</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Business/Domain</td>
<td>Lack of Business/Application knowledge which may impact the quality of work products</td>
<td>Increase interaction with the client and Organize domain knowledge training Simulate/prototype the business transaction &amp; Make pro-active efforts to build System Level Documentation and Program Level Documentation.</td>
</tr>
<tr>
<td>2</td>
<td>Business/Domain</td>
<td>Lack of documentation of requirements may result in under-scope</td>
<td>Prepare the teams understanding of the requirements. Get the Design document reviewed.</td>
</tr>
<tr>
<td>3</td>
<td>Communication</td>
<td>Code development by team split across locations may result in unorganized delivery with a possibility of delivering incorrect source code.</td>
<td>Ensure strict configuration management Maintain communication &amp; coordination between locations. Ensure controlled delivery to client</td>
</tr>
<tr>
<td>4</td>
<td>Customer</td>
<td>A large user group may result in delay in getting consensus on a decision in required time</td>
<td>Negotiate with the customer to have a single point of contact for such decisions. All the decisions shall be routed through the customer point of contact.</td>
</tr>
<tr>
<td>5</td>
<td>Infrastructure</td>
<td>Shortfalls in bought-out &amp; client supplied components (s/w and other) may result in schedule impact</td>
<td>Study the feasibility before committing to the project. Follow-up with CCD/client to get S/W early to evaluate Notify the customer about the exact expectations from the tool/component in terms of quality, performance &amp; schedules</td>
</tr>
<tr>
<td>6</td>
<td>Infrastructure</td>
<td>Vendor Software Crashing may impact schedule</td>
<td>The software has to be configured properly while installation with the help from vendor to avoid the crashing-problems If problem persists identify reason with vendor and re-install software after fixing problem</td>
</tr>
<tr>
<td>7</td>
<td>People/Personnel/HR</td>
<td>Employee dissatisfaction may lead to low team morale</td>
<td>Identify reason for dissatisfaction Satisfy employee expectation within The company Policy framework Have counseling sessions</td>
</tr>
<tr>
<td>8</td>
<td>People/Personnel/HR</td>
<td>Team is not a cohesive unit may result in low employee morale</td>
<td>Proactively plan to create the team as early as possible. Team building and training should be done together to build Have regular team building exercise bonding</td>
</tr>
<tr>
<td>9</td>
<td>Quality/Process</td>
<td>Execution of all identified LC Stages may impact schedule</td>
<td>Perform cost benefit analysis up front tailor the process according to the risks. Try to execute some LC stages in parallel Plan additional resources if required Identify most critical activities and less critical, execute most critical activities Inform all stakeholders well in advance</td>
</tr>
<tr>
<td></td>
<td>Project Management</td>
<td>Requirement changes/Creeping user requirements leading to unplanned rework and schedule slippage</td>
<td>Develop a prototype/ Get the requirements reviewed by the client. Understand from the customer’s perspective what the primary objective is. Define a procedure to handle requirement changes &amp; Negotiate for T&amp;M</td>
</tr>
<tr>
<td>---</td>
<td>------------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>11</td>
<td>Quality/Process</td>
<td>Unrealistic quality targets/SLAs will result in their slippage and customer dissatisfaction</td>
<td>Understand and define client quality/service level expectations in terms of a process as quantitative. Identify suitable project processes.</td>
</tr>
<tr>
<td>12</td>
<td>Technology</td>
<td>Bugs in development tools used in application development leading to excess effort during testing</td>
<td>Try to capture the bugs during regression testing and inform customer about possible defects due to this while reporting</td>
</tr>
<tr>
<td>13</td>
<td>People/Personnel/HR</td>
<td>Cap on number of resources that can be utilized due to which all the customer requirements may not be met within the expected time</td>
<td>Communicate schedule slippage due to lack of resources on a regular basis and extend timelines. Setup a process where the customer prioritizes the requirements on a regular basis.</td>
</tr>
<tr>
<td>14</td>
<td>Business/Domain</td>
<td>Unavailability of users during requirement analysis leading to incomplete requirements</td>
<td>Get a sign off on the requirements. Communicate the customer on the potential of the risk.</td>
</tr>
<tr>
<td>15</td>
<td>Project Management</td>
<td>Project Lead has to time share between Technical activities and Project management activities resulting in inefficient PM</td>
<td>Identify a technical lead. Train persons for handling technical role. Delegate the technical role to team members</td>
</tr>
<tr>
<td>16</td>
<td>Customer</td>
<td>Delay in obtaining customer Signing Off may result in idling resources</td>
<td>Obtain Reverse Sign Off</td>
</tr>
<tr>
<td>17</td>
<td>Communication</td>
<td>Foreign Language issues may lead to improper communication with the customer</td>
<td>Organize language training to the team. Hire a translator if required. Use any automated translation tool.</td>
</tr>
</tbody>
</table>
Criteria-Based Requirements Prioritization for Software Product Management

A. Samer I. Mohamed¹, B. Islam A. El-Maddah², and C. Ayman M. Wahba²
¹Department of computer and system engineering, Ain Shams University, Cairo, Egypt
²Department of computer and system engineering, Ain Shams University, Cairo, Egypt

Abstract - Meeting stakeholders requirements and expectations becomes one of the critical aspects on which any software organization in market-driven environment focus on, and pays a lot of efforts and expenses to maximize the satisfaction of their stakeholders. Therefore identifying the software product release contents becomes one of the critical decisions for software product success. Requirements prioritization refers to that activity through which product releases contents that maximize stakeholder satisfaction can be identified. This makes it one of the most important component of software requirement decision support in incremental software development [8]. This paper illustrates the Criteria-Based requirement prioritization approach for software product management. The technique proposed on this paper designed based on the Hierarchical Cumulative Voting (HCV) and Value-Oriented Prioritization (VOP) techniques. The proposed technique, Value-Oriented HCV (VOHCV) can select the best candidate requirements for each release based on the stakeholders input values for each requirement. These values reflect the importance of each requirement in terms of associated anticipated cost, technical risk, relative impact, market-related aspects and perceived value to the stakeholder. The VOHCV inherits the strengths of the HCV and VOP techniques. It also provides a mechanism that takes different stakeholders’ aspects into account while selecting the best candidate release requirement, to maximize stakeholders’ value and satisfaction [11].

Keywords: Requirements prioritization – Cumulative Voting (CV) - Hierarchical Cumulative Voting prioritization (HCV) Value-Oriented prioritization (VOP) – Value-Oriented HCV (VOHCV).

1 Introduction

Due to the continuous increase in the number of software requirements for market-driven products, there is an increasing need for methods capable of prioritizing candidate requirements. Since not all requirements can usually be met with available time and resource constraints in one software release [9]. Thus many organizations believe that it is not only important to enable their customers to assign priorities to requirements and to make decisions about them but also to provide them with different alternative solutions tailored for their own needs [7]. By this way they will provide more value for their customers through selecting the most valuable requirements to be implemented in each one of the product releases [31].

Managing requirements for any software product becomes a key factor that identify not only the project success or failure but also the organization destiny. The critical portion of this process is to identify those requirements that balance the stakeholders’ needs, customer expectations, business values, total cost and schedule [8]. Therefore requirements prioritization and selection processes that maximize the stakeholder value have a great impact on the product success [18].

Value-Oriented Prioritization (VOP) is a terminology refers to that process which evaluates the requirements from different stakeholders based on the impact on specific business core values for both the organization and to the stakeholders themselves [28]. Since focusing on value provide the opportunity to create a strategy to achieve long-term profitable growth and sustainable competitive advantage [6]. VOP also supports the stakeholders with a visible mechanism during decision-making to be able to provide their values and weights for each requirement. Using both the quantitative and visible approach of VOP, it becomes much easier for the stakeholders to emphasize the core business values and achieve their goals [17].

The organization of the paper is as follows. In the next section, we will refer to the related work for our research, In section three we will elaborate on the rationale for the VOHCV algorithm, and the research methodology we have applied to develop it. In section four, we will discuss the VOHCV algorithm specifications. In section five, we will illustrate the practical advantages from the VOHCV algorithm through a case study. In section six, we will validate the practical benefits from VOHCV algorithm through a comparison between HCV and VOHCV algorithms. The final section will summarize our conclusions and introduces our future research.
2 Related work

2.1 Hierarchical Cumulative voting prioritization

Hierarchical Cumulative Voting (HCV) prioritization technique was designed to overcome the drawbacks for Analytical Hierarchy Protocol (AHP) and Cumulative Voting (CV) techniques, and to inherit the advantages and good features of both techniques [4]. By other means HCV is taken to be an extension for the CV technique by supporting hierarchy, this feature enables HCV to solve multi-aspect decision problems like AHP. Having HCV provides relative priorities based on a ratio scale, gives it the opportunity to calculate the total importance of a set of requirements by adding together their priorities. It also helps to combine the different aspects and calculate ratios in between these aspects. For example, you can calculate the cost-value ratio for the requirement that represents how much value each requirement adds relative to the implementation anticipated cost [19].

The main concept behind HCV is to quantify the requirement importance by distributing points between the requirements to reflect this importance. However, when prioritizing with HCV, not all requirements are prioritized at the same time. Instead, prioritizations are performed at different levels of the hierarchy, and within different blocks of the requirements in the hierarchy as shown in figure 1.

![Figure 1. HCV requirements hierarchy](image)

As illustrated in figure 1, the requirements are distributed over two levels of the hierarchy; high-level requirements (HLR) and low-level requirements (LLR). Only those requirements within the same block (grey area in the figure) are prioritized at the same time. This will make the prioritization process much easier and the risk of neglecting any requirement will decrease [5].

2.2 Value Oriented prioritization

Value-Oriented Prioritization (VOP) proved to be the process which align product demands with company goals and stakeholders expectations through providing a visible and defined process for prioritizing and managing requirements over the product life cycle [28]. It helps out the stakeholder to view the whole picture for the sake of the organization targets and vision, rather than arguing over which product requirements to implement [29].

The whole idea behind VOP is to focus on the core business values that leads to stakeholders satisfaction while prioritizing the product requirements as indicated by Karl Wiegers [30]. Examples of these core business values are the customer value gained from implementing the requirement, the implementation cost, risk associated with implementing this requirement, Impact that will occur if this requirement is not implemented and other market-related aspects that will be affected if this requirement is not implemented [25]. A requirement attractiveness is proportional to the value it provides and inversely proportional to it’s cost, associated risk, impact and market aspects. Each business value is given a weight based on the organization objectives and vision. Each stakeholder put his estimate against each business value for each requirement. All these input values are consolidated together while generating the requirement rank [3].

3 Rationale and research technique

3.1 Rationale

The rationale behind the VOHCV is to combine both HCV and VOP techniques to gain the advantages of both. VOHCV will not only take value and cost as business values into account while prioritizing the requirements [19], but it also will take into account the other business values like associated risk, relative implementation impact and market-related aspects. This will yield to a higher quality results because it takes the different features that affect the requirement throughout the product life cycle into account while producing whole release ranks [10].

3.2 Research methodology

The research methodology we followed for the conception of this technique is based on the incremental software delivery approach and described as follows:

1-Literature review for the current and practical challenges for the software product management industry form both business and strategic perspectives. The outcomes from this review point out to the importance of the prioritization process in handling these challenges [17,20,21,23,24].

2-Literature review for the prioritization techniques that helps to achieve the software product management challenges. The
outcomes from this review pointed out that both HCV and VOP techniques are the best candidates which focus the value gained while prioritizing the product requirements [1,12,16,18].

3-Identifying the pros and cons for those prioritization techniques [13,14,15,22].

4-A prototype implementation for the VOHCV technique based on the knowledge gained from the previous points.

5-Design a framework with the core engine based on the proposed prioritization technique to facilitate testing and evaluating the effectiveness of VOHCV.

6-Use the designed framework to address a group of the open issues with HCV to help maintaining the robustness of VOHCV [3].

4 VOHCV algorithm details

To handle the requirements prioritization using VOHCV, there are a series of steps needs to be followed as follows:

- **Step 1:** Assign the core business values global weights. The supported business values for each requirement in VOHCV are, the anticipated implementation cost, associated implementation risk, perceived customer value, relative impact and market-related aspects. These weights are assigned based on the organization strategic goals and future vision. These weights will range from 1 to 10 such as 1 reflects lowest importance and 10 reflects highest importance.

- **Step 2:** Assign the weights for each business value features. These weights will reflect how much each feature important to the stakeholders and controlled by the organization objectives. These weights will be common to all the stakeholder sharing in the requirements prioritization process. These weights will range from 1 to 10 such as 1 reflects lowest importance and 10 reflects highest importance.

- **Step 3:** Each stakeholder will enter his point of view for each business value feature in terms of feature value. This value will reflect how this feature will affect the requirement from his own point of view. All business values mentioned before have different features except (Value) business value which has only one feature. These values will range from 1 to 10 such as 1 reflects lowest importance and 10 reflects highest importance.

- **Step 4:** Calculate the requirement distribution points assigned to each requirement based on the above feature weights and values. This should be done by each stakeholder sharing in the requirements prioritization process. To show how this distribution points calculated, let us assume the following parameters:

1- \( W_c \): Weight for the global (Cost) business value.
2- \( W_v \): Weight for the global (Value) business value.
3- \( W_r \): Weight for the global (Risk) business value.
4- \( W_i \): Weight for the global Impact business value.
5- \( W_a \): Weight for the global (Aspect) business value.
6- \( W_{ij} \): Weight assigned to requirement \( R_i \) with respect to business value feature \( F_j \).
7- \( V_{ij} \): Value assigned to requirement \( R_i \) with respect to business value feature \( F_j \).
8- \( N_c \): Count of features per (Cost) business value.
9- \( N_r \): Count of features per (Risk) business value.
10- \( N_i \): Count of features per (Impact) business value.
11- \( N_a \): Count of features per (Aspect) business value.
12- \( N_{vb} \): Count of business values that affect requirement.
13- \( CBV_{avr} \): Average value for the (Cost) business value affect requirement.
14- \( RBV_{avr} \): Average value for the (Risk) business value affect requirement.
15- \( IBV_{avr} \): Average value for the (Impact) business value affect requirement.
16- \( ABV_{avr} \): Average value for the (Aspect) business value affect requirement.
17- \( TN_{dist} \): Total distribution number for the requirement.
18- \( AN_{dist} \): Average distribution number for the requirement.
19- \( CF \): Compensation factor to control the range of the distribution number. It will be set to 10 to have the distribution number range between 1 and 100 similar to ordinary HCV.
20- \( R_{Pt} \): Number of points assigned to each requirement.

The process of calculating the average distribution number for each requirement can be shown as follows:

1- Calculate the average business value for each requirement.
   - Average value for the (Cost) business value
     \[ CBV_{avr} = \left( \sum W_{ij}XV_{ij} \right)/N_c \] (1)
   - Average value for the (Risk) business value
     \[ RBV_{avr} = \left( \sum W_{ij}XV_{ij} \right)/N_r \] (2)
   - Average value for the (Impact) business value
     \[ IBV_{avr} = \left( \sum W_{ij}XV_{ij} \right)/N_i \] (3)
   - Average value for the (Aspect) business value
     \[ ABV_{avr} = \left( \sum W_{ij}XV_{ij} \right)/N_a \] (4)

2- Calculate the total distribution number for each requirement.
   \[ TN_{dist} = W_cX CBV_{avr} + W_rX RBV_{avr} + W_iX IBV_{avr} + W_aX ABV_{avr} + WvXVij \] (5)
3. Calculate the Average distribution number assigned to each requirement. We will refer to this number later as the assigned priority.

\[ \text{ANdist} = \frac{\text{TNdist}}{(\text{NbXCf})} \]  

4. Calibrate the distribution number between the different LLRs of the same HLRs to have the sum of all the LLRs points equals 100 as indicated by the Cumulative Voting prioritization algorithm [4]. This can be calculated by using the relation between the LLRs distribution numbers and the below equation.

\[ \sum_{\text{HLR/LLR}} R_{Pt} = 100 \]  

- **Step 5:** Calculate the intermediate priorities for the requirements either through the straight or compensated calculation. To show how the intermediate priority been calculated, let us assume the following parameters:

1. \( \Pi,\text{LLR}_u \): Intermediate priority value for the Lower Level Requirement (LLR) \( (u) \).
2. \( \text{Pa,LLR}_u \): Assigned priority value for the Lower Level Requirement (LLR) \( (u) \) calculated from the previous step.
3. \( \text{Pa,HLR}_v \): Assigned priority value for the Higher Level Requirement (HLR) \( (v) \), or the parent of \( \text{LLR}_u \).
4. \( \text{CHLR}_v \): Block specific compensation factor, this could be the number of requirements within the prioritization block.

\[ \Pi,\text{LLR}_u = \text{CHLR}_v \times \text{Pa,LLR}_u \times \text{Pa,HLR}_v \]  

- **Step 6:** Calculate the final priorities for the requirements at the level of interest. The calculation is performed across the blocks within the same level. This indicates that all requirements located at this specific level will be prioritized relative to each other. To show how the final priority been calculated, let us assume the following parameters:

1. \( \text{Pf,LLR}_u \): Final priority value for the Lower Level Requirement (LLR) \( (u) \).
2. \( \Pi,\text{LLR}_k \): Intermediate priority value for all the Lower Level Requirement (LLR) \( (k) \) of the (HLR \( v \)).

\[ \text{Pf,LLR}_u = \frac{\Pi,\text{LLR}_u}{\sum \Pi,\text{LLR}_k} \]  

- **Step 7:** Calculate the final priorities based on the consolidated stakeholders weighted priorities calculated from the previous steps. To show how the final priority been calculated, let us assume the following parameters:

1. \( \text{Pmf,LLR}_u \): Final priority value for all stakeholders of the Lower Level Requirement (LLR) \( (u) \).
2. \( \text{Pf,LLR}_u,S_k \): Final priority value for stakeholders \( (S_k) \) of the Lower Level Requirement (LLR) \( (u) \).
3. \( \text{Wk} \): Stakeholder normalized weight.

\[ \text{Pmf,LLR}_u = \sum_k \text{Wk} \times \text{Pf,LLR}_u,S_k \]  

- **Step 8:** Calculate the final ranks based on the final priority value assigned to each requirement.

5. **Case study**

In order to show the practical advantage from VOHCV, we will illustrate that through an example based on the requirement hierarchy structure shown in figure 1. In this example, there are two abstraction levels and one stakeholder. Furthermore there are two high-level requirements (HLRs) and five low-level requirement (LLRs). Given the business value weights, features weights and features values for each requirement from table 1, we will be able to calculate the requirements by following the VOHCV algorithm steps mentioned in the previous section.

The data given in table 1 indicate five core business values will be considered while ranking the requirements as indicated by the organization. These core business values are implementation Cost, relative risks, impacts, market-related aspects and requirement value/importance. These core business values are assigned weights as indicated by the first row of table 1. Each one of these core business values can be subdivided into detailed features as indicated by the second row of table 1. C1 refers to development cost while C2 refers to the testing cost. R1 refers to over budget risk while R2 refers to overrun risk. I1 refers to the performance impact. A1 refers to the market share aspect while A2 refers to the profit aspect. V1 refers to the requirement importance to the stakeholder. All these features are assigned weights as indicated by the third column in table 1. The stakeholder input weights for the requirements (LLRs, HLRs) are given in the rest of the table for each feature of the different business values. These all will be used as arguments for the VOHCV algorithm to identify the requirements ranks.

**Table 1. Input values and weights**

<table>
<thead>
<tr>
<th>Business value</th>
<th>Cost</th>
<th>Risk</th>
<th>Impact</th>
<th>Aspect</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.V. Weight</td>
<td>9</td>
<td>10</td>
<td>6</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Feature type</td>
<td>C1</td>
<td>C2</td>
<td>R1</td>
<td>R2</td>
<td>I1</td>
</tr>
<tr>
<td>F. Weight</td>
<td>5</td>
<td>8</td>
<td>10</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>HLR( (Value) )</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>
Table 2. VOHCV Requirements output ranks

<table>
<thead>
<tr>
<th>HLR/LLR</th>
<th>HLR points</th>
<th>LLR point</th>
<th>Comp. factor</th>
<th>Inter. priority</th>
<th>Final priority</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>HLR1/LLR1</td>
<td>30</td>
<td>40</td>
<td>2</td>
<td>2400</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>HLR1/LLR2</td>
<td>30</td>
<td>60</td>
<td>2</td>
<td>3600</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>HLR2/LLR3</td>
<td>70</td>
<td>13</td>
<td>3</td>
<td>6930</td>
<td>26</td>
<td>2</td>
</tr>
<tr>
<td>HLR2/LLR4</td>
<td>70</td>
<td>15</td>
<td>3</td>
<td>3150</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>HLR2/LLR5</td>
<td>70</td>
<td>52</td>
<td>3</td>
<td>10920</td>
<td>40</td>
<td>1</td>
</tr>
</tbody>
</table>

The first step in the VOHCV algorithm is to calculate the number of points assigned to each requirement of the same block, or by other means distribute 100 points over the requirements of the same block. This can be done by applying equations 1 through 7, given the values and weights of table 1. These assigned points illustrated in the first two columns in table 2 (HLR points, LLR points). After all the requirements at the same level of hierarchy in the prioritization blocks have been assigned priorities, the next step is to calculate the intermediate LLR priority (Inter. Priority) in the fourth column of table 2 using equation 8, given that the compensation factor (Comp. factor) is equivalent to the block size as illustrated in the third column of table 2. The next step is calculating the final normalized LLR priority as indicated in the fifth column of table 2 using equation 9. The last step is to rank the LLRs based on the final LLR priority in the fifth column of table 2. The LLR ranks priority illustrated in the sixth column of table 2.

As shown from the first two columns of table 2, HLR2 (70%) is more important than HLR1 (30%) and LLR5 is considered to be the most important LLR and account for (40%) of the importance of all the LLRs while LLR (9%) is considered to be the lowest important LLR over all the other LLRs.

6 Evaluation of VOHCV versus HCV

In order to show the strength of the new proposed technique (VOHCV) compared to the ordinary (HCV), an empirical evaluation should be conducted. The main drawback of HCV is that it takes only the “Value” perspective indicated in the ninth column of table 1 into account while prioritizing the requirements and neglecting the other business perspectives. On the other hand VOHCV fix this by taking the other business perspectives into account through the prioritization process.

To show that, we will use the example introduced in the previous section and exclude all perspectives except “Value” perspective to gain the HCV ranking as indicated in table 3. After that a detailed comparison between the two techniques will be conducted based on the results from table 2 and 3.

In order to calculate the distribution points for both LLRs and HLRs for HCV based on the values mentioned in the ninth column of table 1 that related to (Value/importance business value), we will use the relation between these values that belongs to the same prioritizing block. For example both LLR1 and LLR2 are belonging to the same block and have value equal to 4 and 10 respectively. To distribute 100 points over these two LLRs with the same ratio between the assigned values, we concluded by simple mathematical calculation that LLR1 can be assigned 71 points and LLR2 can be assigned 29 points. The same can be done for LLR3, LLR4 and LLR5 as they are belong to the same block. The calculation for the later case will yield to LLR3 will be assigned 43 points, LLR4 will be assigned 14 points and LLR5 will be assigned 43 points. These values are indicated in the first and second columns of table 3. The prioritization and ranks calculation for the same HLRs/LLRs as indicated in section 5 for the VOHCV will be shown in table 3. These results will be used for our evaluation of the VOHCV algorithm against HCV algorithm.

Table 3. HCV Requirements output ranks

<table>
<thead>
<tr>
<th>HLR/LLR</th>
<th>HLR points</th>
<th>LLR point</th>
<th>Comp. factor</th>
<th>Inter. priority</th>
<th>Final priority</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>HLR1/LLR1</td>
<td>22</td>
<td>71</td>
<td>2</td>
<td>3124</td>
<td>11</td>
<td>4</td>
</tr>
<tr>
<td>HLR1/LLR2</td>
<td>22</td>
<td>29</td>
<td>2</td>
<td>1276</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>HLR2/LLR3</td>
<td>78</td>
<td>43</td>
<td>3</td>
<td>10062</td>
<td>36</td>
<td>1</td>
</tr>
<tr>
<td>HLR2/LLR4</td>
<td>78</td>
<td>14</td>
<td>3</td>
<td>3276</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>HLR2/LLR5</td>
<td>78</td>
<td>43</td>
<td>3</td>
<td>10062</td>
<td>36</td>
<td>2</td>
</tr>
</tbody>
</table>

Comparing the results between HCV and VOHCV as illustrated from table 2 and 3, we conclude the followings:

1-LLR5 is considered as the most important of the LLRs and accounted for (40%) of the importance of all the LLRs as result of applying VOHCV technique.

2-LLR3 is considered as the most important of the LLRs and accounted for (36%) of the importance of all the LLRs as result applying HCV technique.

3-LLR1 is considered as the least important of the LLRs and accounted for (9%) of the importance of all the LLRs as result of applying VOHCV technique.

4-LLR2 is considered as the least important of the LLRs and accounted for (4%) of the importance of all the LLRs as result of applying HCV technique.
5-Neglecting the effect of the other business perspectives rather than “Value” yield to a miss leading result. This can be shown by comparing the features values for both LLR5 (The most important of the LLRs from VOHCV) and LLR3 (The most important of the LLRs from HCV) from table 1. The values of LLR3 indicates that the stakeholder assign to them small values to indicate low importance of this LLR in compare to the others. While the values for LLR5 indicates that the stakeholder assign to it a large values to indicate how important this LLR in compare to the others which detected by VOHCV.

6-The results quality from VOHCV is much higher than HCV and reflects the real business case. The real business case not take only the effect of “Value” and neglect the other perspectives, but it takes all the different perspectives that affect the requirement throughout the product life cycle into account.

7-Adopting VOHCV as a methodology for prioritizing the requirements yields to higher credibility with the release contents that results in customer

7 Conclusions and future work

In this paper, we have presented the strengths and weaknesses of both Value-Oriented Prioritization (VOP) and Hierarchical Cumulative voting (HCV) techniques. From these strengths and weaknesses, we come up with the new prioritization technique which is Value-Oriented Hierarchical Cumulative Voting (VOHCV). VOHCV combines the strengths of both VOP and HCV to improve the quality of the requirements prioritization process. The main difference between the HCV and VOHCV is that, VOHCV uses the criteria-based approach not only to get the effect of value but also to get the effect of different other core business values like cost, risk, aspect and impact [10]. By this way VOHCV enables the product manager to take all the aspects that affect the requirement into account while selecting the best candidate for product release [23].

We already designed a release management framework with as impeded VOHCV to get benefit from the results produced by VOHCV in the release planning and software product management [30]. VOHCV acts as the main core engine for the designed framework. The results from the VOHCV taken as inputs for further release planning activates to support and enable product manager to easily control and manage the product [27].

Since there are only a few studies have been performed to evaluate the efficiency and suitability of VOHCV, there is a need to do further studies for some issues that can affect the algorithm efficiency [2,12,22]. Examples of these open issues like how many points we should distribute over the requirements of the same block, how many hierarchy levels should be prioritized, how large priority blocks are possible to prioritize and also the effect of the requirements order on the requirements ranking, will be our objective for the next phase of our research.

8 References


SESSION

INTEGRATING HUMAN FACTORS IN COMPUTING TECHNOLOGIES

Chair(s)

Dr. Vincent Schmidt
Wright Patterson Air Force Base
Uncovering Expectations to Support Trust in System Development

Dr. Janet E. Miller
Air Force Research Laboratory
Wright Patterson AFB, OH

Abstract - As complex socio-technical systems become ubiquitous and as world events become more dynamic, the division of labor in human-automation systems begins to weigh more heavily toward the automated system. Nevertheless, as many of the systems involve life-and-death decision-making, humans often remain as the final decision-makers. Therefore, developing a positive trust relationship between all stakeholders in the development and implementation of a system is important to encourage initial trust in the system. Expectations are an integral part of trust and so uncovering and understanding all stakeholders’ expectations are important for system development and implementation. This paper will discuss three methods explored to directly address expectations thereby enhancing initial trust in the resulting human-human and human-automation relationships.

Keywords: Expectations, Trust, Participatory Design

1 Introduction

As complex socio-technical systems become ubiquitous and as world events become more dynamic, the division of labor in human-automation systems begins to weigh more heavily toward the automated system. Nevertheless, as many of the systems involve life-and-death decision-making, humans often remain as the final decision-makers. Therefore, a tight, trusting relationship must exist between the automation and the human decision-maker. To develop a system that has such a relationship, what the relationship entails must be understood prior to system development. A myriad of methods exist [1] to discover requirements needed to perform tasks so that better support, such as for these largely cognitive activities can be developed. The complexity of understanding the environment and the tasks, combined with the fact that experts performing cognitive tasks have difficulty reliably articulating about the task when asked, contribute to making discovering the full set of requirements including the expertise hard.

Despite having these established methods of gaining understanding about a domain, the systems engineering community struggles with the difficulties of handing the research results to designers and also of handing designs to developers so that shared understanding of the problem and possible solutions exists. A more frustrating challenge occurs when a developed system is implemented but is not enthusiastically embraced by the end-user. Participatory Design (PD), an established, diverse research and practice area, has a goal of engaging researchers, designers, developers, practitioners and end-users in the various activities leading to the successful development and implementation of systems. PD is an umbrella methodology which includes studies, theories, conferences and practices [2], [3]. Using methods which involve all of the stakeholders in the cradle to grave development of a system improves the trust relationship as all have had their input to the system. Developing a trust relationship between all stakeholders in the development and implementation of a system is important to encourage trust in the resulting system which eventually becomes implemented.

Trust itself has been investigated and reported in many conference proceedings and journal articles [4], [5], [6]. Trust has been defined in many ways such as Madsen & Gregor, 2000 [7]:

“Trust is the extent to which a user is confident in, and willing to act on the basis of the recommendations, actions, and the decisions of a computer-based tool or decision aid.”

Other definitions include the concept of anticipation of reliance on an action, based on what a party knows about the other party. Involving the concept of anticipation introduces the concept that expectations are an integral part of trust. Muir [8] examined the concept of trust and attempted to find the common threads between the diverse definitions of trust by determining the underlying recurring themes. One theme consists of the expectation of or confidence in another that is oriented towards the future. A second involves the idea that trust always has a referent, such as when our trust is particular to something or someone. Lastly, trust may be related to the characteristics or properties of the referent, such as honesty, reliability, and motivations. Muir later revised the original framework using a combination of Barber’s [9] taxonomy of the components of trust and Rempel, Holmes & Zanna’s [10] taxonomy of the dynamics of
trust. Table 1 shows Muir’s updated framework with an inserted emphasis on the concept of expectation. This framework ties the concepts of expectation and trust together allowing the concepts to be applied to both human-human and human-automation relationships.

Building on Muir’s framework theory, Lee & See [6] defined appropriate reliance on automation and specified the differences between human-human and human-automation trust. Over trust exceeds the system’s capability. Distrust falls short of the system’s capabilities. Calibrated trust is the ultimate goal which matches trust with the system’s capabilities. Three main differences in how trust in automation differs from trust in humans are proposed as (a) automation lacks intentionality, (b) human trust is usually a social exchange, and (c) human trust typically evolves while automation trust may a going-in expectation. These generalizations along with the Jian, Bisantz & Drury [11] universal definition of trust summarize the problem of improper trust in automation.

These same general concepts apply when considering the trust, or expectation, relationship between members of an engineering team. One member may have over trust in another member due to reputation which may not be transferrable to the current project. Distrust may occur for a variety of personality reasons. Calibrated trust is the goal for the human to human team relationship as well as to the human to automation relationship in the system eventually fielded. Teams have been researched especially as teams are being used more to address one-of-a-kind situations. Communication between team members has been recognized as being important in building relationships and reaching goals [12].

Expectations are held by all of the stakeholders in the development and deployment of a system. This includes the designer, the programmers, the developer, the program managers and the user of the actual system. Expectations are important as they are the hidden agenda in many actions. As stated in Huron [13], “Expectation is an omnipresent mental process; brains are constantly anticipating the future.” Expectations facilitate three purposes:

- Motivation to take action to increase the likelihood of positive outcomes.
- Preparation to react in appropriate ways
- Representation of expected events for evaluation of the various mental representations.

Revealing the expectations of all stakeholders in a system’ development is important so that all expectations can be addressed and trust calibrated. If unrealistic or uncalibrated expectations exist, the time to uncover them is prior to the actual coding and definitely before the final implementation of a system. Below, three methods are discussed which can be used to uncover expectations at different stages of development. All three methods can be used as participatory design methods by involving all stakeholders when using the methods.

2 Method

The methods described below were used as a basis for the Air Force Research Laboratory’s (AFRL) program, Commander’s Predictive Environment (CPE). The program was funded with two main goals. The first goal, which has the purpose of strengthening AFRL’s capabilities to develop human-centered capabilities, is for a strong bi-directorate partnership. The two directorates are the Human Effectiveness Directorate and the Information Directorate. The application of this partnership is to work toward the second goal that of building tools to enhance the Joint Forces Commander (JFC), Joint Forces Air Component Commander (JFACC) and senior commanders’ decision making process by supporting their ability to envision future operational environment options. The description given for this second goal was very grand and very broad in vision making program success difficult. Therefore, the CPE Program Management Office decided to use structured methods to pare the scope down into a manageable program by engaging Air Operations Center experts upfront in requirements identification and concept development as well as during the development of the potential automated capabilities. The program managers and the software developers were also involved in the methods described below. By using these methods, the CPE program co-managers expect to get a better understanding of everyone’s expectations for the resulting decision support systems and avoid the frustrating challenge of developing capabilities that are not enthusiastically embraced by the end-user.

2.1 Method 1: Pre-Mortem

Pre-Mortem refers to discussing the end result of a program before the effort has begun. Team members are asked to envision the likely outcome of the program which makes them think about what might influence reaching that end. A Pre-Mortem was done at the initial meeting of the CPE team whose members are from two geographically separated locations. Participants also included those not in the research and development of the system but who were representative of end users. In order to identify the critical issues, the participants were asked to answer two hypothetical questions which bookend the two potential outcomes of the program. The participants were only given five minutes for each question so that they did not have time to evaluate their responses and so would be more of a subconscious response.
Question 1: It is 6 years in the future. The Program team has just presented with the Outstanding Scientist Award for the technical breakthroughs and outstanding customer support over the last 6 years. What was the single most important decision made in today’s first technical working session, and what was the single most significant factor during the succeeding time period that led to this disaster for the program?

Question 2: It is two years in the future and the Program has been cancelled for a total lack of progress in advancing its vision. What was the single worst decision made in today’s technical working session, and what was the single most significant factor during the following years that led to this disaster for the program?

Each participant answered in round-robin fashion until all responses were gathered. Discussion was not held until all responses were in.

2.2 Method 2: Value Focused Thinking (VFT)

Value-focused thinking (VFT) is a multi-attribute utility theory methodology that can help identify what is needed in an interface for a particular application and can be used to compare different potential interface solutions or can be used to judge how well an interface currently meets the customer’s needs. Fully describing the methodology is beyond the scope of this paper so the reader is invited to read [14]. The methodology provides a means to reveal and address the multiple objectives of an interface design effort and includes eliciting from all stakeholders. The primary benefit that VFT provides is its ability to identify and convert the goals of a project or values of an organization into an objective realm. Its structure lends itself to handling multi-objective problems even if the objectives are of a subjective nature. Using VFT, high-level objectives are broken down into smaller values. In general terms, a VFT methodology uses a five-level delving by asking ‘why’ five times to get at the basic rational, or in this case, expectations of a person or group. Once articulated, the values can be measured and put to a common scale, allowing their contribution to the overall objective to be evaluated. By assigning quantifiable measurements to the components, the multi-objective goal can be evaluated. Value focused thinking (VFT) is a proven decision analysis methodology that can be applied to a variety of multi-criteria situations.

This methodology was used in two ways. One was to reveal the expectations of the potential users of a developed system. The other was to reveal and reach agreement on the expectations of the program. In both cases, the discussions were led by an objective facilitator and the participants included all identified stakeholders including the system researchers, program manager and a group of intended users of the system. The facilitator asked members to clearly identify their expectations for the intended system and ensured good communication methods so that all had a chance to participate.

2.3 Method 3: Vignette Framing

Vignettes, or simplified scenarios, help a person get their head around a particular situation or problem. In this case, a vignette was used to elicit previously unstated expectations concerning a prototype tool to support military intelligence requests for information (Figure 1), one of the specific automated capabilities expected from CPE. The vignette described a military intervention scenario where the US military was tasked to establish a no-fly zone and establish and sustain airhead under the complications of lack of near-by basing and political constraints. The program managers, system developers and intended users of the system had to envision using such a tool to help them gain information such as information about the political objective of the rebel and pro-government forces, the likely location of SA-2 batteries, and the warfighting equipment of the rebel forces. The intended end users mentally went through the stages of receiving requests and envisioning how they would use such a tool to do their job while program managers and developers listened and made notes.

3 Results

3.1 Result of Method 1: Pre-Mortem

Although many of the responses were light and “tongue-in-cheek,” they helped to identify the really important expectations of a joint research and development program. Responses for explaining reason the Program was an outstanding success included:

- We agreed to truly collaborate instead of just consulting with each other.
- The team streamlined the process for knowledge engineering, design, AND development.
- The research created scientific value in several technology areas.
- The team pushed the science, advanced the state of the art, and succeeded in producing something commanders can use.
- We made and followed through on a commitment to advance scientific knowledge for predictive analysis.
- We delivered something the customer loved.
- We produced a coordinated program plan that included a top-level view of all environments.
Responses for why the Program was a failure:

- We failed to agree upon and implement an executable program.
- We failed to agree upon a process and framework for collaboration in the team.
- We lacked clarity of vision and failed to agree upon an end product.
- We couldn’t get past status quo organizational and funding issues.

3.2 Result of Method 2: VFT

Applying the method goes through several layers, generally five, of questioning. See Figure 2, which only goes through 2 levels but gives a general idea of the responses. Table 2 describes the further breakdown of the 3rd and 4th Levels of Engine Process, which is a Level 2. The fundamental objective in this case was to identify what is valued in a software system for a complex analytical domain, which is descriptive of the intended CPE domain. The questioning interaction between the facilitator and the participants determined that the input process of software, the processing part of software, and the output process of software were necessary components of the system. These two levels did not probe into expectations but helped focus the group on future system. Level 3 elicited what were the users expectations for these Level 2 entities and the responses included simplicity, pleasing presentation, intuitive feel, observable engine process, simple user control, and choice of delivery. Eliciting Level 4 pushed the users to define what was meant by the responses in Level 3. Terms included in Level 4 included aesthetics, forgiveness, efficiency, flexibility, similarity to other domain software, traceability of the engine process, comprehension of the engine process and confidence.

Level 5 pushed even further into what the user’s expectations were and obtained the terms of directed input, interpretation, error alert, impact, reliable automated features, readability, attention-directing, customization, consistency, logical ordering, consistent context and readability. For a complete description, refer to [15].

3.3 Result of Method 3: Vignette Framing

The participants evidently generated a mental model of the current system for comparison to the envisioned system as an area greatly discussed was the lack of specific requirements in the system being currently used. Gathered comments included:

- Nothing with certain classifications can be entered
- Unclassified requirements not handled
- If required information is outside of the normal, defined process flow, there is no traceability
- Having a tool that can handle a variety of domain types would be excellent
- Building situation awareness quickly is needed
- No advocacy for requirements to be filled

4 Discussion

System developers are often stymied when systems they develop for needs they have identified never become strongly embraced and oftentimes not used at all. Participatory design methods demand that all of the stakeholders be involved during the whole process and have been shown to address this challenge. The three methods described above were used as participatory methods and revealed interesting expectations. One important insight was that the intended users of the automation wanted to have insight into how the algorithm was producing the output. The developers had expected that trust in their capability to develop the system was sufficient and had not planned on providing transparency. If this expectation had not been revealed, the likelihood of the system being used would have been decreased.

The Pre-mortem allowed an atmosphere of informality and congeniality thereby allowing inhibition in the comments. Discussion followed comments as each participant stated his fears and hopes. The questioning and delving in the VFT session at times got a bit uncomfortable as the user was made to think hard. However, working through the challenges was beneficial and the format allowed a free exchange of information. The Vignette Framing allowed the users to do a cognitive walk-through of the system with developer stakeholders present. Again, an informal, congenial atmosphere pervaded which helped the exchange of expectations. The result of using these methods was that the user community built a level of trust with the researchers and developers because the expectations were revealed and discussed which passes on to the applications being built. The intended user’s expectations of the system was calibrated with the stakeholders be involved during the whole process and have been shown to address this challenge. The three methods described above were used as participatory methods and revealed interesting expectations. One important insight was that the intended users of the automation wanted to have insight into how the algorithm was producing the output. The developers had expected that trust in their capability to develop the system was sufficient and had not planned on providing transparency. If this expectation had not been revealed, the likelihood of the system being used would have been decreased.

Strong advocacy for the automated systems, which resulted from applying these methods, has been built with the intended user base. This attitude is in contrast to other attempts of implementing systems developed for the same user base. The uncovering and discussing of expectations that these methods allowed was undoubtedly part of the reason the intended community is willing to push for deployment of the applications.
On the other hand, the diverse team working on CPE had a difficult time getting past the issues that were listed as likely causes of failure for the program. This was for a variety of reasons including geographical separation, having to coordinate with other programs with different goals to obtain resources, and cultural differences in the two directorates. Often the program managers would go back to the list of likely causes of failure, admit that these are known barriers to the program’s success, and discuss ways to overcome the issues. Solutions included weekly video-teleconferences, on-line shared repository of documents, and semi-annual program team meetings.

For automated systems in general, and decision support systems in particular, the program management team is at the core of trust as they are the ones who are defining the capability. When a person first uses any automation, from a software application to a global positioning system to a blood pressure monitor, an underlying unstated expectation is that the stakeholder developers, ranging from the programmers to the designers to the user representatives, understood the end users’ full suite of expectations and fulfilled those expectations. Ensuring the program management team understands those expectations by using a combination of methods such as described above supports developing a system that meets those expectations and works to build trust.

Disclaimer

The views expressed in this paper are those of the authors and do not reflect the official policy or position of the Department of Defense, the United States Air Force or the U. S. Government.

References


Basis of expectation at different levels of experience

<table>
<thead>
<tr>
<th>Persistance</th>
<th>Predictability (of acts)</th>
<th>Dependability (of disposition)</th>
<th>Faith (in motives)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Natural physical</td>
<td>Events conform to natural laws</td>
<td>Nature is lawful</td>
<td>Natural laws are constant</td>
</tr>
<tr>
<td>Natural biological</td>
<td>Human life has survived</td>
<td>Human survival is lawful</td>
<td>Human life will survive</td>
</tr>
<tr>
<td>Moral social</td>
<td>Humans and computers act “decently”</td>
<td>Humans and computers are “good” and “decent” by nature</td>
<td>Humans and computers will continue to be “good” and “decent” in the future</td>
</tr>
<tr>
<td>Technical competence</td>
<td>j’s behavior is predictable</td>
<td>j has a dependable nature</td>
<td>j will continue to be dependable in the future</td>
</tr>
<tr>
<td>Fiduciary responsibility</td>
<td>j’s behavior is consistently responsible</td>
<td>j has a responsible nature</td>
<td>j will continue to be responsible in the future</td>
</tr>
</tbody>
</table>

Table 1. Muir’s (1987) two-dimensional framework for studying trust with annotation on expectations framework

Figure 1. Prototype tool for intelligence analysts
Figure 2. Value Focused Thinking Hierarchy for CPE

<table>
<thead>
<tr>
<th>Engine Process – 2nd level</th>
<th>What is Valued in Software Interface for a Complex,</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visibility (3rd Lever)</td>
<td>To be able to display what the engine is doing.</td>
</tr>
<tr>
<td>Traceability (4th Level)</td>
<td>The ability to show the algorithms that the engine is using, to see them, and to be able to step through each step of the algorithm.</td>
</tr>
<tr>
<td>Comprehendible (4th Level)</td>
<td>The ability to trace where the algorithm comes from and see where it is used in the processing.</td>
</tr>
<tr>
<td>Confidence (3rd Level)</td>
<td>The ability to explain the algorithm and its uses in an understandable way.</td>
</tr>
<tr>
<td>Appropriate (4th Level)</td>
<td>The ability to see that the algorithm is being used correctly in the software to provide confidence in the software’s processing ability.</td>
</tr>
<tr>
<td>Verification (4th Level)</td>
<td>To show the engine is using the appropriate or the correct algorithm</td>
</tr>
</tbody>
</table>

Table 2. Engine Process Further Level Breakdown for Figure 2
Integrating Cognition into System Design

D. O’Malley          J. Zall
Dept of Systems and Engineering Management
Air Force Institute of Technology
Wright-Patterson AFB, OH, USA

J. Colombi, PhD     J. Carl, PhD, PE, CSEP
Riverside Research Institute
Air Force Institute of Technology
Wright-Patterson AFB, OH, USA

Abstract - Early system design is a complicated process that lays the foundation for eventual system success. Today’s design efforts often fail to fully account for the unique characteristics of human involvement in systems and system architectures. Highlighting, defining, then integrating, the specific human cognitive and system pseudo-cognitive functionality can ensure that human capabilities and requirements are not added too late in the design lifecycle, or as an afterthought. Cognitive models should improve early technical decision making, and provide traceability to manpower, personnel and training decisions. This research develops early cognitive models, for potential future extensions to the DoD Architecture Framework (DoDAF).

Keywords: Cognition, (DoDAF) DoDAF, HSI, Systems Engineering

1 Introduction

New system design teams should always integrate their efforts with the insight and knowledge of human factors engineers. The integration, however, often comes at a point far too late into the design process to gain the true advantage of their knowledge [1]. The focus of this research looked at inclusion of one aspect of human factors, cognition, into the Department of Defense Architecture Framework (DoDAF) [2]. DoDAF compliant architectures are mandated throughout the defense acquisition lifecycle.

Although all elements of human systems integration (HSI) need to be addressed by systems engineers during the design process, cognition has been overlooked in practice. However, it should be pointed out that cognitive engineering has a long international history of publication – with the goal of enhanced man-machine performance and reduction of human errors [3]. Human system integration goes beyond the ergonomics-centered challenges of just physical interfaces. HSI includes such domains such as Human Factors, Manpower, Training, Personnel, Survivability, Habitability, Safety and Health [4]. At a high level of abstraction, system design engineers make trade-offs on what system functions will be accomplished through automated means and which ones will be accomplished by humans. This is as true now as it was in the 1950’s (Fitt’s List on what Human/Machines do well).

2 Why Model Cognition?

The nature of cognition provides the justification for choosing that one area of human systems integration (HSI), while all elements of HSI can benefit the design if included early in the lifecycle. If they know the cognitive elements of a system or architecture, designers can answer the question, “Do humans play a direct or indirect role in this system and if so, where?” Determinations about all of the other aspects of HSI depend on the answer to this question. Identifying specific elements of cognition provides design traceability for including humans in different parts of the overall system. Those decisions ultimately impact trade-offs between what types of people can interface with the system, what levels of training eventual users will require and what individuals are integral to system performance and functional success, as well as resilience to unknown environmental conditions. The end result is a system that better utilizes its human cognitive element(s) cost effectively. Initial research on HSI elements has been documented for the Ministry of Defense Architecture Framework (MODAF), but a focus on cognition for either MODAF/DoDAF architectures is an open issue [5,6].

Systems Engineers have long known that Operations and Maintenance account for over 70% of lifecycle costs on most large programs. DoD cost studies have identified manpower, personnel, and training (MPT) as 40% to 60% of the total system’s life cycle cost. In another study, Dray states that there is a direct tradeoff between MPT costs and investment in the human interface. She cites a commercial project in which an improved user interface on a large-scale internal application resulted in a 32% overall rate of return stemming from a 35% reduction in training and a 30% reduction in supervisory time [7]. The importance of HSI and cognitive understanding as well as modeling cannot be overstated.

3 Cognitive Artifacts

Human cognition spans a wide range of research areas, necessitating the difficult task of bounding this problem for DoDAF purposes, i.e., deciding what specific aspects of cognition would most benefit initial system design. These aspects of cognition modeling will eventually provide the foundation for system trades, complexity definitions and,
ultimately, manpower, personnel, and training decisions. This research uncovered five areas in the field of human cognition from which models can be constructed that are salient to DoDAF architectures. These five cognitive areas are: Cognitive Tasks, Cognitive Inputs, Cognitive Outputs, Cognitive Roles, and Cognitive Environments.

3.1 Cognitive Task (Activity)
A Cognitive Task is an identification of what specific type of cognition is required. Section 4.0 within this document further describes a method to model and describe the types of cognition. For early conceptual designs, all cognitive tasks can be left unallocated to humans or machines. As the design matures, engineers can differentiate the tasks to either cognitive (human-accomplished) or pseudo-cognitive (machine accomplished). For this article, references to the general term “cognition” should be understood to include both human cognition and pseudo-cognition.

In some cases it is easy to attribute a task to a machine. In other cases, it may be important to maintain a human in the loop. Categorizing a task as cognitive or pseudo-cognitive permits system engineers to view what percentage of system tasks are accomplished by humans and what percentage by machines. This information can be valuable when considering multiple concepts with varying complexities, varying levels of automation, and varying levels of human supervision and control.

3.2 Cognitive Input
Cognitive Input describes the specific information that the Cognitive Task requires in order to be completed. The input may be a result of other cognitive or pseudo-cognitive tasks. Tracking the Cognitive Input elements permits system engineers to identify and quantify all enabling items for each Cognitive Task. Recognizing each of the inputs can help identify shortfalls in other areas and other critical shortages that could threaten system functionality. For example, in design of Human Computer Interfaces, Cognitive Inputs would include the information conveyed to the operator by the set of all system “displays.”

3.3 Cognitive Output
A Cognitive Output is the set of information that results from the cognitive task. Information resulting from cognitive activities permits design engineers to gather a sense of system complexity and reliability. It also specifically identifies information that emerges from human cognition that must be conveyed back to the machine. That identification can help to separate the data into groups of cognitive, pseudo-cognitive, and system produced data. In cockpit design, cognitive outputs, such as Hands on Throttle and Stick (HOTAS) controls, multi-function display buttons and all switches receive and use Cognitive Output sets of information.

3.4 Cognitive Role
As the Cognitive Tasks develop, they can be grouped into Cognitive Roles. Initially, the role may simply be “Decision Maker” or “Role A”. Eventually, the Cognitive Role can take on a specific personnel position description and help to define the MPT system requirements. This may lead to early definition on knowledge, skills and abilities (KSA).

The Cognitive Role can also help design engineers define the right number of people to accomplish specific elements of a system. Too many cognitive tasks grouped into “Role A” might provide rationale for the need of a second “Role A” or “Role B” to share duties. It might also help engineers realize that the exact personnel who fit the needs of “Role A” do not exist. Such a realization early in the design process is simple to repair. Discovering that during the later stages of design can be difficult to overcome.

3.5 Cognitive Environment
The Cognitive Environment includes a location attribute in which the Cognitive Role(s) will accomplish Cognitive Tasks. It can be “Environment A” and help design engineers identify cognitive teams within the system. It can also resolve collaborative team requirements if it becomes apparent that a necessary team cannot all exist in the same location.

3.6 Forming a Cognitive Logical Data Model
The five elements of cognition can be considered information entities, with descriptive attributes and relationships. See Figure 1. A Cognitive Task requires Cognitive Inputs for execution and produces Cognitive Outputs. The input and output is information passing across the Cognitive Environment boundary. The Cognitive Tasks are accomplished by the Cognitive Role within the Cognitive Environment.
The five elements can be further developed from Figure 1 into a greater inter-relationship as depicted in Figure 2. In this situation, multiple Cognitive Tasks have been identified and grouped into specific roles. In this situation, all roles exist within a single Cognitive Environment.

The five elements (as entities in a logical data model) also have specific attributes. For example, the Cognitive Environment has the attribute: location. This can reflect the operational node or the system node (OV-2 and SV-1 in DoDAF, respectively). Cognitive Role shares the location attribute.

![Figure 2: Cognitive Elements can be Interactive](image)

**4 Cognitive Task Types**

One of the important attributes of a Cognitive Task is “Type”. There have been many efforts to define the types and bounds of cognition. Human behavior, including human cognition, is often characterized in one of two ways: using a hierarchy or taxonomy [8,9,10]. For a hierarchical approach, the Cognitive Type domain is defined as the collection of its “leaf-level” or lowest level elements. For a taxonomic approach, the domain is simply the one-dimensional list of “species” that makes up the taxonomy. Some approaches are a blend of hierarchies and taxonomies; however, all of these approaches face two similar challenges.

The first problem has to do with mutual exclusivity. Human behavior and cognitive hierarchies and taxonomies are riddled with ambiguity. This lack of precision, for engineering design purposes, is a consequence of industry’s incomplete understanding of human cognition. In taxonomies, different “species” often overlap. Consider the two cognitive tasks, “choose” and “decide”. Different hierarchies and taxonomies treat these as separate activities. However, the commonalities in their American English definitions make it difficult to assign one or another to a candidate cognitive task. For example, upon encountering a junction while navigating a maze, does a person “choose” a direction, or does she “decide” on a direction? These ambiguities not only exist between the leaf level elements of existing human behavior hierarchies, but also in between levels on different branches of the hierarchy. This imprecision is an accurate reflection of the nature of cognition, as the science currently understands it. Unfortunately these overlaps and ambiguities present a hindrance to a useable method for incorporating cognition into DoDAF-compliant architectures.

Usability is at the core of the second problem that plagues hierarchies and taxonomies. In most cases, the leaf level elements or taxonomic “species” are legion (typically 30-50 human or cognitive behaviors). These lists are purported to be exhaustive. If we accept that such lists are comprehensive, that leaves architects and engineers the daunting task of discerning subtle differences of meaning between words like “abduce”, “induce”, and “infer”. Even if an individual feels confident in their ability to characterize a cognitive activity, he still faces the challenge of communicating those different meanings to others.

Considering these challenges, a solution needs to be found which clearly articulates differences in human cognitive tasks. The ability to discriminate between tasks will be required when making trades between manpower, personnel, and training. The solution must be useful in the context of DoDAF and understandable to architects. The solution for defining Cognitive Task Type should meet the following criteria:

- The domain must provide sufficiency to describe any human cognitive activity within a DoDAF model.
- Elements of the domain must be mutually exclusive (i.e., they must avoid the ambiguity problems described earlier).
- The domain must allow for some means to assign different values and weightings to cognitive tasks.
- The domain must provide a means to trace manpower, personnel, and training requirements back to the cognitive tasks within the concept or system.
- The domain’s size should be limited so as to be easy to use.
- The domain’s elements must be understandable by non-engineer architecture customers and stakeholders.

Instead of a hierarchy or taxonomy, this research uses a building block, or atomic approach. Instead of characterizing every type of cognitive activity in a long list, a building block approach decomposes activities into a set of common elemental tasks which can be combined and rearranged to form any other cognitive activity. Our proposal for such a building block approach is summarized in Table 1. It is comprised of eight fundamental cognitive tasks that we assert can be combined and arranged to form all cognitive activities within a concept or system, thereby defining the domain for Cognitive Task Type.
Before describing the domain in detail, it is important to point out that by advancing this proposal, we do not claim to fully understand the intricacies or actual mechanisms of internal human cognition. This approach is possible, because the objective is to better model and evaluate alternative designs, and devise a method to best account for human thought and its affect on a concept or system. The construct adopted for this research focuses on the inputs and outputs of cognitive tasks. Those inputs and outputs are in fact used to define the task type, even though the mechanism (i.e., how a task is actually accomplished in a human brain) is hidden using a “black-box” approach. This solution does not depend on details of human biology or neuro-science. Instead it applies systems engineering principles to develop a useful and relevant solution for characterizing the five cognitive factors listed above.

It is natural to focus on the names assigned to the Cognitive Types. However, those names should only be viewed as inexact labels. This is because of the difficulty with word meanings discussed earlier. Most hierarchical and taxonomic solutions rely on common, dictionary definitions for the labels they use, while others use their own specialized definitions that differ from common dictionary definitions.

Both of these methods are problematic. The first relies on the English language (in our case), in which words have myriad different meanings, connotations, and subtleties. The second method requires users to either memorize new definitions for words they already know and use often, or to keep some kind of unwieldy guide that must be frequently consulted to avoid confusion.

This solution relies on the inputs and outputs to define a particular task, instead of some phrase tied to the task’s English word label. As an analogy consider a vending machine. There are many types of vending machines and what call them (“snack machine”, “automat”, “candy machine”) is less important than what goes in and what comes out. A machine that takes only quarters and produces gumballs is distinguishable from a machine that comes out. A machine that takes only quarters and produces cans of soda.

The same goes for cognitive tasks. As you read about the tasks do not be confused by the English word labels. They are only there to give readers a general sense, albeit imprecise, of what a particular type of cognitive task does. Instead, use the inputs and outputs as the true definition of a particular type (e.g., “Compare” can be thought of as a cognitive task that takes a set of information and produces a relationship tensor.)

The eight fundamental tasks are broken into to two major categories: translation (as in geometric translation of an object through space and/or time without altering the object) and transformation (as in a change in an object such that the output is distinct from the input object). There is only one translational cognitive task, which takes a set of information and outputs the same set of information (labeled “Convey”). Translation has two different forms: translation through time and translation through space-time. Translation of a set of information through time provides memory and recall. Translation of a set of information through space-time provides communication. (Advances in quantum teleportation of information notwithstanding, we have chosen to adopt relativistic constraints on information communication for now.) Here we encounter our first English language obstacle.

The word “Communication” connotes a variety of processes. Is speaking on a telephone a form of communication (between a human brain and another human brain)? Is day-dreaming a form of communication (between one part of the brain and another)? To surmount this first hurdle, we place the following constraint on cognitive tasks: they only occur inside the defined cognitive environment. While cognitive inputs and outputs can traverse the cognitive environment boundary, cognitive tasks occur entirely with it. So if we define the cognitive environment as a human brain, then our previous example of day-dreaming can be characterized, at least in part, by the cognitive task we call “convey”. With the same environment definition, however, our example of talking on the telephone is not an example of the cognitive “convey” task. Some cognitive task (or combination of cognitive tasks) produces the words to be spoken, but then a different process (one not encompassed in the scope of our problem solution) produces the physical actions (nerve impulses, muscle movements, lung compression, vocal cord tension, etc.) that result in audible speech over a telephone.

However, the definition of the cognitive environment is not limited to one human or one human brain (or one computer for a pseudo-cognitive environment). If we define the cognitive environment as a pair of humans who must act together as a team (i.e., they accomplish one or more cognitive roles together), then our telephone conversation example can be defined, at least in part, by cognitive communication. The flexibility in defining the cognitive environment will allow architects to address not only human system integration, but human-human interaction (team work, crew functions, etc.). Depending on the scope of the concept or stage of development, cognitive environments may be decomposed (i.e., a cognitive environment of a “crew” can be decomposed into “teams” which can be further decomposed into individuals. In some concepts it may be necessary to carry that decomposition even further by separating parts of the brain into different centers that are inside their own cognitive environments (e.g., a system may require differentiating between the inputs and outputs of the sensory cortex versus the motor cortex of one person’s brain). This concept allows the architecture designer to choose the appropriate level and still derive meaningful information about cognitive factors within the architecture.
The other fundamental process that happens within the cognitive environment is the transformation of sets of information. We have adopted the mathematical connotation of a set \( (X_a) \), which is any collection of objects. Those objects, for the purposes of this cognition construct, can be thought of as discrete ideas. The seven transformational cognitive tasks take a set of information as an input and transform it in some way. In some cases that set is the empty set, as in the cognitive task we label “create”, where something (i.e., a non-empty set of information) can be generated out of “nothing” (i.e., an empty set of information). In order to understand the differences between these transformations, we must first define the inputs and outputs that characterize the tasks.

In addition to sets of information, inputs and outputs of cognitive tasks include attribute tensors, relationship tensors, triggers and construct. These inputs and outputs are defined below with an informal mathematical description in brackets at the end of each definition. To understand what we mean by each of these elements, consider the example of purchasing a new car and the cognitive elements that are involved with the selection of a particular car to purchase.

---

**Table 1 : Proposed eight elemental cognitive/pseudo-cognitive tasks, for use during early conceptual design**

<table>
<thead>
<tr>
<th>Cognitive Task Type Label</th>
<th>Description</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Prime Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convey (ReCall)</td>
<td>to translate a set of information through space-time from one spatial and/or temporal location to another without transforming the information</td>
<td>( X_a ) ( \Theta ), ( A_{a_0} ), or ( R_a )</td>
<td>( X_a )</td>
<td>2</td>
</tr>
<tr>
<td>Classify</td>
<td>to group objects within a given set of information according to a given set of attributes with values/ranges</td>
<td>( X_a ) ( A_a )</td>
<td>( G_x(A_a) = X_b ) (where ( X_b \subseteq X_a ))</td>
<td>3</td>
</tr>
<tr>
<td>Characterize</td>
<td>to determine a set of attributes with values/ranges from a given set of attributes with values/ranges, or of a given set of information</td>
<td>( \Theta ) or ( X_a ) ( \Theta ) or ( A_a )</td>
<td>( A_a )</td>
<td>5</td>
</tr>
<tr>
<td>Choose</td>
<td>to select a subset of a set of information based on a given set of relationships between the objects within the given set of information</td>
<td>( X_a ) ( \Theta ) or ( R_a )</td>
<td>( X_b ) (where ( X_b \subseteq X_a ))</td>
<td>7</td>
</tr>
<tr>
<td>Combine (Calculate)</td>
<td>to transform a given set of information based on a given set of relationships between the objects within the given set of information or between the given and transformed sets of information</td>
<td>( X_a ) ( R_a )</td>
<td>( X_b )</td>
<td>11</td>
</tr>
<tr>
<td>Compare</td>
<td>to determine a set of relationships between the objects/subsets of a given set of information</td>
<td>( X_a ) ( R_a )</td>
<td>( X_b )</td>
<td>13</td>
</tr>
<tr>
<td>Create</td>
<td>to generate a set of information from nothing, or from a given set of attributes with values/ranges and/or a template set of information</td>
<td>( \Theta ) or ( X_a ) ( \Theta ) or ( A_a )</td>
<td>( X_b, A_{b_0}, ) or ( R_b )</td>
<td>17</td>
</tr>
<tr>
<td>Construe (Interpret)</td>
<td>to generate a set of information which is the interpretation or meaning of a given set of information</td>
<td>( X_a ) ( \Theta ) or ( R_a )</td>
<td>( M(X_a) = X_b )</td>
<td>19</td>
</tr>
</tbody>
</table>

The other fundamental process that happens within the cognitive environment is the transformation of sets of information. We have adopted the mathematical connotation of a set \( (X_a) \), which is any collection of objects. Those objects, for the purposes of this cognition construct, can be thought of as discrete ideas. The seven transformational cognitive tasks take a set of information as an input and transform it in some way. In some cases that set is the empty set, as in the cognitive task we label “create”, where something (i.e., a non-empty set of information) can be generated out of “nothing” (i.e., an empty set of information). In order to understand the differences between these transformations, we must first define the inputs and outputs that characterize the tasks.

In addition to sets of information, inputs and outputs of cognitive tasks include attribute tensors, relationship tensors, triggers and construct. These inputs and outputs are defined below with an informal mathematical description in brackets at the end of each definition. To understand what we mean by each of these elements, consider the example of purchasing a new car and the cognitive elements that are involved with the selection of a particular car to purchase.

- **Information set** \( (X_a) \) – A collection of information (discrete ideas or concepts) that may be the empty set \( (\Theta) \) or be composed of other sets (subsets) of information. There are two specific subcategories of information sets: groups and meanings. (E.g., the set of all new 2008 automobiles for sale in the local area.)

- **Group** \( (G_i) \) – A subset \( (X_b) \) of a collection of information \( (X_a) \) where the objects in the subset are described by an attribute tensor \( (A_i) \). \( [G_i(A_i) = X_b \) where \( X_b \subseteq X_a \), \( \& \) \( A_i = i^{th} \) set of components of \( A_i \) (see definition of \( A_i \) below)] (E.g., the subset of all Volkswagen automobiles for sale in the local area.)

- **Meaning** \( (M_i) \) – A set of information \( (X_b) \) which is the meaning or interpretation of a collection of information \( (X_a) \). \( [M_i(X_a) = X_b = f(X_a)] \) (E.g., the “meaning” of the set of VW turbo diesel vehicles is “cool.”)

- **Attribute Tensor** \( (A_i) \) – A collection of attributes which may also include the domains and corresponding values and ranges of those attributes. If \( A_i \) describes the objects within \( X_a \), then \( A_i \) is a function of the objects within \( X_a \). \( [A_i = f(X_a)] \) (E.g., the set of attributes for all of the
Volkswagens for sale, including color with the domain, engine type, engine size, options package, etc.)

-Relationship Tensor (R) – A collection of relationships which may also include the corresponding values and ranges of those relationships. If R describes relationships between the objects in X, then R is a function of the objects’ attributes (Ai). [R = f(Ai), where Ai = f(Xi)] (E.g., which cars are “cooler” than others.)

-Trigger (Ti) – A set of information that causes (directly or indirectly) a cognitive role to execute a cognitive task or cascade of cognitive tasks. Ti may include the iteration count or limit for a task, the time limit for a single iteration, output conditions, and input orientation with respect to the construct within the cognitive environment. It may designate other specific sets of information as inputs (as a convention, we separate those sets of information from the trigger). It may also include bounds on inputs (i.e., what information to ignore). (E.g., the realization that your current car is not “cool” enough.)

-Construct (Ci) – The set of information, attributes, and relationships that inform the execution of a cognitive task. Construct forms the mental substrate onto which input information is projected (and subsequently altered or distorted). For the cognitive environment of an individual human mind, the construct (ΣCi) is the aggregation of the knowledge, memories, training, experience, prejudices, and attitudes of that individual. (E.g., a predisposition to liking sporty, environmentally conscious sedans and memories of good experiences with VW vehicles in the past.)

Based on these elements, we can now define the rest of the cognitive tasks as combinations of these elements. Reading the “Definition” columns in Table 1, we define, for example, the cognitive task labeled “Construe” as the task that takes a set of information, X, (along with trigger, Ti, and construct, ΣCi) and produces a different set of information, M(ΣCi) = X, that is the meaning of the first set. These formal definitions are paraphrased in Table 1 as the task type descriptions. Figure 3 shows the construct applied to the tasks labeled “convey”, “classify”, and “characterize”. The seven fundamental transformations plus the translational task, “convey”, provide the ability to produce any of the cognitive elements (as outputs), which may then become inputs for another cognitive task, or may exit the cognitive environment as a cognitive output. This allows the combination of these tasks in any arrangement.

Our assertion is that any cognitive activity can then be characterized by a combination of one or more of these fundamental tasks, in a way that is consistent with the meaning of the task and useable in DoD-driven developments. For example, take the cognitive task of “prioritize.” Again the English language offers us too much ambiguity in its meaning, so we will apply the same technique we used with the fundamental tasks in order to define “prioritize”. Therefore, we define the cognitive task labeled “prioritize” as the task that takes a set of objects and outputs a list that consists of the same objects in the input that are arranged in rank order. This could apply to cars or tactical courses of action. Now we can take what we know about the inputs and what we know about the outputs to determine what fundamental tasks make up the task labeled “prioritize”.

Figure 3: Illustration of 1st Three Cognitive Task Types

The inputs consist of one set of information (the original list of objects to be prioritized) along with some set of attributes that can be used for prioritization, and a relationship (the rank order condition) which is a function of a set of attributes that describe the objects. [Inputs: X, R = f(Ai), λ; Outputs: X] The next step in decomposing “prioritize” is to determine if it can be characterized as a special case of just one of the fundamental tasks. In terms of inputs and outputs, the task label “combine” has similar inputs and outputs, but on its own, the description of “combine” fails to encompass our description of “prioritize”. Therefore we look for a combination of fundamental tasks that can take the appropriate inputs, and provide the desired outputs. In order to create combinations of tasks, the following conventions are adopted:

-All information elements that enter the cognitive environment must be input to at least one cognitive task. Elements produced within the environment must also be input to at least one task or they must exit the cognitive environment as a cognitive output.

-All elements within the cognitive environment are included within the construct (lines showing this
aggregation are omitted for clarity except when explicitly required.

Elements may be used as inputs for more than one cognitive task within the cognitive environment. All cognitive outputs (\(Y_{n,p}\)) are sets of information which may include attributes and relationships.

Armed with these rules, a candidate combination of fundamental tasks can be proposed to achieve the definition of “prioritize”. Figure 4 shows how the combination of “classify” and “combine” captures the essence of the description earlier, while maintaining the right internal and external composition of inputs and outputs. For the task, “prioritize”, \(X_0\) is the list of items to be prioritized. \(A_0\) defines the specific attributes used to rank order the items in \(X_0\). \(R\), defines how the values of the attributes in \(A_0\) should be ranked (small to big, big to small, best to worst, worst to best, fastest to slowest, slowest to fastest, etc.). \(X_0\) represents the groupings of items into attribute value levels or classes. (In many cases those “groups” will only consist of one item.) \(X_0\), then is a new set of items, nearly identical to \(X_0\), except that the items in \(X_0\) have a new attribute of “rank” or “priority number” with an associated value. This translates to \(Y_0\) being an ordered list of the items in \(X_0\). This process can be repeated for any cognitive activity as long as the inputs and the outputs of that activity can be characterized.

Figure 4: Cognitive task “prioritize” can be decomposed into elemental tasks “classify” and “combine”.

5 Conclusions

This elemental, building block approach does not just solve the problem of establishing a useful list of cognitive task types. It solves the problem of defining how to model cognition for early design, and it is intended as a usable modeling tool for architects and systems engineers, with the aid of HSI experts. Cognition (and pseudo-cognition), for the purposes of DoDAF-compliant architectures, can now be sufficiently defined as any task or activity that is on the list of eight fundamental cognitive/pseudo-cognitive tasks (Table 1), or that can be decomposed into a combination of those tasks. As early concepts develop, Cognitive Tasks, Input/Outputs, Roles, and Environments will provide architects a better understanding of humans as an integral part of the overall system and enable more efficient and successful system developments. Use of this model will have the greatest impact during concept development and solutions analysis. Future research will continue to extend DoDAF by applying cognitive models to perform conceptual trade studies and architectural analysis on human factors, manpower, personnel and training requirements.

Acknowledgement: The authors thank our sponsor, Air Force Research Laboratory (AFRL/RH), especially Dr. Vince Schmidt, as well as Dr. Chris Hale from SAIC for his crucial contributions.

Disclaimer: The views expressed in this paper are those of the authors and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government.

6 References

Cognitive Design Patterns

Christopher R. Hale  
Science Applications International Corporation  
Dayton, OH 45431

Abstract We introduce the concept of cognitive design patterns and discuss ways in which these patterns can better integrate early work analyses with software development. Cognitive design patterns are units of work that, in combination, enable human operators to accomplish the range of tasks needed for success in complex systems. Each pattern consists of a normative model of the relevant cognitive competency, expressed in terms accessible to software design and practice. Our proposal is that these patterns be included as resources in GUI builders, thereby adding standardized design capabilities to the software engineering toolkit.

Keywords: Cognition, HCI, Design Patterns

1.0 Introduction

One of the enduring frustrations of both the Cognitive Systems Engineering (CSE) and software development communities is the apparent inability to communicate with one another when designing complex systems. The CSE community has developed many methods and tools that can lead to a thorough understanding of the nature of work and the human-centered requirements for a system. However, this understanding often is not adequately conveyed to the development community in a way that is actionable. That is, there exists no good way to translate “requirements,” as defined by the cognitive systems engineer, to code used by the software engineer to instantiate a working system. Fundamentally, this problem is comprised of two parts, both of which fall primarily at the feet of the cognitive system engineer. First, the content produced by the CSE community often is inadequate for the needs of the system and software engineering communities, often addressing issues that do not bear on design needs at a level of analysis that the engineering community can effectively use. Second, the information produced by the CSE community often is not communicated to the engineering communities in ways that enable incorporation of the information into ongoing design process. To be useful, CSE analysis results should be articulated in the language of requirements, specifications and normative principles. Instead, these analyses often are stated in the language of theories, controlled experiments and idiosyncratic results.

One solution to these problems would be to develop the kinds of content that lend themselves to requirements, specifications and normative principles and then to communicate this content to the engineering development process in the language of engineering development. An important concept used by the software engineering community to define and document the content of useful concepts is that of software design patterns. We are attempting to extend the design pattern concept to the problem of integration between CSE and engineering development by defining Cognitive Design Patterns (CDP). These are normative units of cognition that, when combined in different ways, can account for the work required in any context. Work elements upon which such patterns are based are identified and characterized during the CSE analysis that is carried out in the early stages of system design. Cognitive units are derived from these work elements and are “parameterized” for the specific design problem at hand. These normative cognitive units, along with other information obtained from the CSE analysis, are combined with other elements of system analysis to formulate system requirements and build a system model.

In addition to using the work elements to define and specify cognitive components at the design level, it is also possible to use the elements within a structured software development environment. Contemporary graphical user interfaces (GUIs) are being created by programmers using GUI-builder programs that specialize in GUI layout and corresponding code generation. Since cognitive work elements often model processes that include visual or procedural components, a list of work element “widgets” could be added as a GUI builder module. Such an implementation makes the cognitive component of design look even more like “cognitive design patterns” to both the designer and the implementer.

Adding a set of cognitive work element widgets to a GUI builder application brings two distinct advantages. First, relevant visualizations for the corresponding cognitive tasks could be suggested to the coder. This enables the programmer to more easily select appropriate techniques for implementing the cognitive specifications without fear of misrepresenting the requirements. Once a cognitive work element widget is chosen, a coding wizard might be used to assist developers with selecting and parameterizing an appropriate visualization, and a code template could
provide the basis for the resulting code in the programming language of choice.

Second, direct selection of the work element from a list of cognitive work element widgets promotes direct traceability back to the requirements. This level of traceability is useful for ensuring that all requirements have been accounted for in the system. Nearly as important, this traceability can be used to justify the reasons a system has a particular “look and feel,” answering such questions as why a screen or system function looks or behaves a certain way.

The next section of this paper introduces the concept of cognitive work elements in more detail, providing an enumerated list of (what the authors would claim as) a comprehensive set of cognitive operations. These work elements are a valuable part of the initial system specification, providing early systems engineering inputs into the system design process.

For software systems, the paper also describes a mechanism by which developers can directly include these work elements in the application by selecting work elements as “widgets” within a GUI builder. This approach provides programmers a guided method for implementing cognitive requirements, and also shows end-to-end traceability from the requirements to the final software application.

Before proceeding to our own ideas and research, it must be mentioned that others, especially within the human factors community, also have worked to define design patterns. Most of this work concentrates on high-level patterns of work and cognition, however, and falls short of introducing a mechanism by which technologists have direct access to the patterns for use in implementing software systems, as we highlight below. See [1 – 4] for several excellent examples of this parallel work.

2.0 Work Elements
From the point of view of Cognitive Systems Engineering (CSE), traceability is a crucial challenge to successful design. By traceability we mean the ability to relate originating (from the Cognitive Work Analysis (CWA)) and functional requirements defined early in a development program to the final artifacts that constitute the system under development. Ideally, one should be able to relate these requirements to the resulting artifacts through the design commitments made at the various stages along the development path of the system. Thus, when customers and users of the system ask “why does this artifact look the way it does and behave the way it does,” the designer of a traceable system should be able to “reverse engineer” the artifact back through each series of design decisions to the original requirements contained in the CWA.

How does one explicitly relate the information contained in the CWA to the resulting artifact, and do so in terms of the engineering process that (should) form the bridge between these two points? Our approach has been to find ways to integrate the cognitive requirements with all stages of the system engineering process, thereby ensuring that the artifact that is eventually built is formed and constrained by this information. We do this by developing a matrix that explicitly relates the Cognitive Workflow Elements (CWE) identified in the original CWA to the system requirements resulting from early system analysis and modeling.

We define CWE as “units” of workflow required of the human system component to carry out elements of work that a system is being built to accomplish. The CWE for a particular system are identified by analyzing the contents of the CWA, critical decision analysis and other analyses carried out in the early stages of a development effort. Typically, a small set of CWE will result from this analysis. For example, consider a CWA for a visualization system designed to support operational assessment. Based on a set of concept maps developed through documentation, observation and detailed interviews with Subject Matter Experts (SME), we identified the CWE shown in Table 1.

<table>
<thead>
<tr>
<th>Acquire</th>
<th>Communicate</th>
<th>Compare</th>
<th>Infer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decide</td>
<td>Discriminate</td>
<td>Estimate</td>
<td>Integrate</td>
</tr>
<tr>
<td>Assign</td>
<td>Aggregate</td>
<td>Evaluate</td>
<td>Identify</td>
</tr>
<tr>
<td>Choose</td>
<td>Describe</td>
<td>Generate</td>
<td>Interpret</td>
</tr>
<tr>
<td>Classify</td>
<td>Detect</td>
<td>Match</td>
<td>Plan</td>
</tr>
<tr>
<td>Monitor</td>
<td>Recognize</td>
<td>Prioritize</td>
<td>Verify</td>
</tr>
</tbody>
</table>

Table 1. CWE for Operational Assessment

These elements are adequate to encompass all of the cognitive workflow required to carry out operational assessment. After the element set is identified, we develop conceptual definitions for each element. For example, our definition for detect, as carried out within the context of operational assessment, was: Become aware of the existence of an object, value or attribute. We then create a matrix that juxtaposes the CWE against the system requirements. A fragment of this matrix for an operational assessment visualization system is shown in Table 2, with system requirements in the left-most column and the CWEs from Table 1 across the top.

Notice that some cells of this matrix contain check marks. These indicate that the corresponding CWE...
participates in satisfying the system requirement in the adjacent row. Thus, each row specifies the various combinations of CWE required for effective operation of the system, where effective operation is a function of adherence to the system requirements. This will explicitly link the CWE to the resulting system design through the system requirements, thereby enabling traceability. There is one further, and crucial, step to be taken once the CWE have been identified, conceptually defined and mapped to the system requirements. This is to develop models of the CWE.

Table 2. Traceability Matrix for an Operational Assessment Visualization System

<table>
<thead>
<tr>
<th>System Requirement</th>
<th>Acquire</th>
<th>Aggregate</th>
<th>Assign</th>
<th>Choose</th>
<th>Capacity</th>
<th>Communicate</th>
<th>Compare</th>
<th>Decide</th>
<th>Devise</th>
<th>Describe</th>
<th>Detect</th>
<th>Discriminate</th>
<th>Evaluate</th>
<th>Generate</th>
<th>Infer</th>
<th>Integrate</th>
<th>Interpretable</th>
<th>Plan</th>
<th>Monitor</th>
<th>Predict</th>
<th>Recognize</th>
<th>Verify</th>
</tr>
</thead>
<tbody>
<tr>
<td>The system shall aid in determining the potential effect of weather, terrain, and/or air space current/future on possible plan changes</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The system shall aid in determining if tactical objectives will be achieved</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The system shall aid in determining if operational objectives will be achieved</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The system shall allow and aid in the assessment of both direct and indirect effects of air, space and IO on the mission</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The system shall allow and aid in the derivation of the intended and unintended consequences of air ops, wet platforms, munitions, culture, population</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The system shall allow and aid in determining when and what to report to JFACC</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The system shall aid in determining the difference between the plan and actual</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The system shall aid in determining how mission failures will affect the overall plan</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The system shall aid in determining how effective mission successes were</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>The system shall aid in determining if we are ahead of schedule</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Development of such models makes the system requirements executable. With executable system requirements, expressed through rigorous models of the CWE, it is possible to create integrated system simulations that will enable tradeoff analyses to be carried out prior to specification of detailed software requirements or development of physical system concepts. To facilitate this process we develop normative models of each CWE that will allow exploration of the variables and parameters expected to affect system effectiveness.

For example, consider the following requirement for the operational assessment visualization system mentioned above:

*The system shall provide a way to derive intended and unintended effects from tactical assessment results.*

Referring to concept maps developed in the initial CWA, we find that this requirement can be satisfied with a combination of the following CWE: Inference, acquisition, classification, detection, evaluation, interpretation and recognition. The executable model of this requirement will be comprised of submodels for each CWE, organized into a task network model of the overall work environment. We normally use a higher-level, discrete-event performance modeling package, such as the Combat Automation Requirements Testbed (CART) to carry out these simulations. When combined with environment and system models these human operator modeling packages enable us to express broad ranges of human performance within the context of overall systems, thereby allowing study of the kinds of constraints that will limit performance. Consider, for example, the interpretation element of the above requirement, where interpretation is defined as determining the task-relevant value of data or information. We assume that interpreted value is a function of the timeliness (T) and credibility (C) of the data or information received. Further assume that timeliness follows a sigmoid function in which the timeliness of information ranges from 1 immediately after it becomes available down to an asymptotic value.
approaching zero after many hours. In this case we can define interpreted value as:

\[ V = TC \]
Where: \( T = 1 - \left[ \frac{1}{1 + e^{-b}} \right] \) and
\( C = \log_b(\Sigma z_i v_i) \)

Developing similar models for each CWE allows us to model ranges of performance for each of the requirements, with individual low-level models being connected together through the human system model. This allows us to model a wide range of work demands on the human component of the system and to produce estimates of human performance that can be used by other members of the system development team to conduct trade studies. These models also allow us to evaluate the conformance of human operators to requirements as the requirements are further refined to include system performance or effectiveness specifications, thereby connecting the humans to measurement of overall system effectiveness.

With executable models of the requirements in hand, the cognitive system engineers can then begin developing visualization concepts for system interfaces. Each cognitive work element has basic visualization requirements defined for it that we assume are consistent across contexts. For example, the cognitive element compare involves examining two or more objects in terms of their similarities and differences. The visualization requirements for this element include displaying to users the attributes and values of objects being compared. Further, the display should facilitate the comparison being carried out, for example, by highlighting the similarities and differences through some method of ranking, coding or some other means. As this example shows, there will be a basic structure associated with each element as well as performance parameters for the elements. Parameters for compare might include the number of to-be-compared elements that can be held in short-term memory and sensitivity limitations on attribute similarity used in comparisons.

The requirements provide the context, constraints and boundaries for visualization design for each CWE. Thus, while the basic requirements will not change across elements, the values of parameters associated with modeling of elements will change according to the context of each requirement. Consider a requirement to compare an air attack result against a target, located close to a mosque, with the intended point of attack to assess progress toward an effect. In this case the sensitivity parameter for the comparison would be set to a high value, since collateral damage to the mosque would lower the assessment of success toward effect. The comparison of planned to actual result would indicate success only if the attack were extremely precise, that is, resulted in no damage to the mosque.

By this method we develop visualization concepts for each primitive within the context of the system requirements. Common combinations of requirement and cognitive work elements are collected together into common visualization concepts. The individual concepts then are aggregated into higher-level collections to form visualizations at the screen level. This process is iterated against the CORE system model, thereby allowing validation of visualizations by ensuring that the system follows the processes outlined in that model.

### 3.0 Integrating with GUI Builders

Good human factors designs are frequently “lost in translation” between the original interface designers and those ultimately responsible for system implementation, much to the dismay of all parties. This is due largely to a mix of communication and technical issues. In fact, many software and systems engineers are ignorant of human factors issues altogether, which makes the inclusion of carefully designed solutions practically impossible. One potential solution to this problem is to include the work elements concept (which is one part of a full human systems interface (HSI) solution) into the tools used by the software community.

GUI builder applications are often the centerpiece for software development. In addition to providing a visual method for designing a software system’s interface, many high-quality GUI builders offer features such as round-trip software engineering, direct access to software repositories, and the inclusion of robust integrated development environments (IDEs). Some of these utilities even support multiple programming languages. Further modularity allows additional features to be added by the vendor, or even by third parties.

A practical mechanism for including CWE components is to include them within the GUI builder framework. One way to do this is to make the CWEs available as widgets within the framework, much like traditional button and menu elements are presented as widgets. A modular plug-in extension with the additional CWE widgets could be used to implement this approach.

Detailed specifications and high-level descriptions would be accessible for each CWE, either as a part of contextual popup dialogs or explicit help text. These specifications can be used to assist with the selection of relevant CWE widgets, an especially valuable feature for those implementers possessing a limited background in cognitive science. Widget tooltips and representative
ic images also will help the coder to quickly identify the specific CWE desired.

Figure 1 depicts a representative GUI builder (Glade, in this case). The area highlighted by the oval shows where a modular CWE toolbar would be loaded into the application. The intent is to make CWE as well-integrated into the application as the rest of the widget set. Therefore, the “look and feel” of the CWE module is expected to be as similar to that of the native widgets as possible, while simultaneously providing the specific capabilities of the CWE functions.

A programmer’s selection and placement of a specific CWE into the design accomplishes several tasks. First, suggestions for implementing the CWE are presented to the programmer. (There may be several ways the CWE can be implemented or represented.) The CWE itself may be a visually-oriented or algorithmically-oriented component. For algorithms, a wizard will assist the programmer by providing a selection of relevant algorithm, their descriptions, specifications, and references. For visual components, the wizard might display the various visual representation options to the programmer.

In addition; operational code, code templates, or pseudo-coded (comment-based) solutions can be directly inserted into the code base. The options selected from the wizard indicate the CWE implementation to be included in the code. The generated code is managed within the GUI builder just as with the other drag-and-drop widgets.

Another benefit is that comments can be injected into the code surrounding the CWE implementation. These comments are explicit notations that CSE issues are directly addressed within the coded application. The construction wizard should allow the programmer to include clear text, to be added as comments corresponding to the CWE being constructed.

Finally, comments in the code provide a record of traceability back to the system’s requirements. This encourages quality assurance by linking the specific requirement to the reason a particular feature or operation is implemented in a certain way. The requirement identifier is assumed to be a clear text string that can be captured by the CWE construction wizard and automatically included in the generated code.

Again, consider the CWE “compare.” When the programmer has selected and placed this widget, a wizard introduces a series of questions: What requirement does this meet? Is this a visual or a conceptual comparison? Visual comparison might include options to show items side by side within a tabbed window, as a popup, or interlaced (as in visual code diffs). Should the display be hard-coded, or selectable by the end user? Perhaps the end user’s requirement is only to examine the differences between several items, or perhaps the end-user must make a selection based on the comparison. If a selection is to be made, how will that selection be indicated to the system? Does the system make a recommendation to the user? (If so, what is the name of the method or function to be called to assist with that comparison?) If the “compare” CWE is conceptual only, then the wizard might ask the programmer for the method or function names to call for the comparison, and how to indicate the status of the operation. The wizard can be used to guide the coder through all of these issues, and decisions can be captured and annotated as comments in the code. This information will be valuable for justifying implementation decisions.

4.0 Conclusion

We have described an often-ignored, but important, component of software system design: The cognitive aspect. In this paper, we described a method that not only encourages the inclusion of cognitive components into the design, but also introduces a practical mechanism through which software implementers can directly incorporate key cognitive aspects into the code.

One reason it is important to define such cognitive components is to ensure that human cognitive needs and expectations are properly included at the system specification level, early in the design process. Systems engineers can use these definitions as key inputs by
incorporating them into early definition processes and products (DoDAF, etc.) See [4--7] for examples of integrating cognitive work requirements into the design process.

Another reason a mechanical approach to including cognitive components is needed is to provide a mapping from the software directly to the cognitive requirements. Such traceability is an important part of a unified systems engineering process.

5.0 References


Implementing UI Design Patterns as Widget/Templates for GUI Builders

Donald West
Consortium Research Fellows Program
Washington, D.C, USA

Vincent Schmidt
Wright-Patterson AFB, AFRL/RHCS
Dayton, OH, USA

Abstract - During the development of a software system, something is lost in translation of the Human-Computer Interface (HCI) between the human factors engineer’s analysis and the software developer’s implementation. Since the developer touches the product last, part of the human factors engineer’s contribution is frequently lost. Graphical User Interface Design Patterns (UIDP) are templates representing commonly used graphical visualizations for addressing certain HCI issues. These patterns include substantial contributions from human factors professionals, and using these patterns as widgets within the context of a GUI builder helps to ensure that key human factors concepts are quickly and correctly implemented within the code of advanced visual user interfaces. This paper introduces the concept of the UIDP and describes how this concept can be implemented to benefit both the programmer and the end user by assisting in the fast generation of error-free code that integrates human factors principles to fully support the end-user’s work environment.

Keywords: design patterns, GUI, HCI, Widget

1.0 Introduction

Early software benefited from the simple use of abstract data types (ADTs) such as stacks, queues, and linked lists. Even with these basic ideas enhancing the coding for more advanced machines with improved graphics and processing power, programmers still struggled to find ways to reliably describe concepts frequently adopted within the core of the code. The 1994 “Gang of Four” book Design Patterns: Elements of Reusable Object Oriented Software [1] took the software development community by storm with its popular and accurate description of frequently used software design patterns. Using the patterns described in the book (and later, other patterns identified by the community at large), programmers could code reusable components with a high degree of confidence that the models were complete and descriptive.

The design pattern concept is not new. Christopher Alexander is frequently cited as the inspiration of documented patterns with his 1979 book The Timeless Way of Building [2] that describes patterns of architectural design (Alexander, 1979). Even if documented ADTs are considered as a form of early design patterns, there is no argument that the Gang of Four [1] book significantly impacted the state of the art. Along the same lines, Jenifer Tidwell introduced a book and website describing design patterns as applied to user interface design [3,4]. The first chapter is of this book, Designing Interfaces, is devoted to describing the relationship a user interface should have with the end user.

Human Factors Specialists (HFSs) focus on, among other things, the design of excellent user interfaces targeted towards the end user. The traditional approach to building computing systems is for the HFS to generate conceptual interfaces, and then pass these designs along to the programmers for implementation. Unfortunately, inadequate specification and gross miscommunication is the norm, so the well-designed interfaces are frequently generated in such a way that many of the elements intended to contribute to a well-designed interface are left out of the final implementation. The result is a new software system that does not optimally support the end-user’s work environment.

One solution to this problem is to have a User Interface Design Pattern (UIDP) [5] library that goes beyond the mere description or specification of Graphical User Interface patterns (GUI). With the exception of ADT libraries, most design pattern work currently ends with the description of the patterns and small snippets of sample code. This leaves implementation details up to the programmer, introducing the strong potential for misrepresentation of the final GUI elements. A UIDP library would include templates that can be parameterized to generate a “90% solution” to specific GUI elements in the design. Each one of these templates would be vetted by HFS experts, and their use would guarantee that human factors components are correctly represented graphically to the end user.

Our contribution to this solution is to provide these library components as “widgets” in a popular drag-and-drop GUI builder. Programmers could drag these
human-factored UIDP elements into the applications being built, and an integrated wizard would guide the programmer through a series of tailored queries to parameterize the template. The code generated by the GUI builder would include not only the code necessary to design the GUI’s standard widgets (pull-down menus and button elements), but also code to generate the specialized UIDP elements. This will result in an application where human factors designs are not lost due to implementation miscommunications. Furthermore, the introduction of Human Factors into the standardization of design patterns encourages coders to use these pre-designed elements, since these elements will be more user-friendly and less prone to coding error. The details of our progress to date and anticipated future direction are documented here.

2.0 User Interface Design Patterns

Good human factors design dictates that data be portrayed in a manner consistent with the work being performed, such that those accessing the information be able to quickly and easily read and understand the situation. There are common methods for visualizing certain types of data, where minor alterations in the visual display can be performed to suit a wide variety of situations and software applications. These visualizations are called User Interface Design Patterns, and they represent a class of repeatable, general solutions to commonly occurring end-user display needs.

As new user-interface design patterns are discovered, they can be documented in a design pattern library. This library can serve as a foundation to all user interface creation, much as the Gang of Four’s [1] documentation of elements did for the creation of object-oriented software. A good pattern library would include a list of the patterns, a description and formal specification for each pattern, sample pattern usage code, and a reusable template.

Pattern software objects can be made with good human factors design already implemented, effectively representing a “90% solution” to a programmer’s visualization requirements. The key benefit to such patterns is that high-quality code with built-in human factors design is available to the coder to support rapid software implementation. The resulting application will be well-suited to the user, meeting good human factors design criteria. (Coders are not traditionally trained in human factors sciences, so many of their interfaces poorly support the user or the work environment.)

The subsections that follow describe the user interface design pattern concepts by way of example. The Map and Timeline patterns are used within our own organization as custom-coded components, but we are working to show how they can be used by a wider audience using a general software development framework.

2.1 Map Pattern

The Map pattern is useful for displaying spatial information as layers of imagery, symbols, and corresponding text. As the name suggests, this pattern is found primarily in maps and mapping software. Since geographic paper and electronic maps are commonly used, most people are already familiar with the map concept.

In general, maps are frequently used for driving directions and road map scenarios. However, this pattern could also be used in a wide variety of other scenarios: creating a weather map for a new area, mapping underground tunnels, visualizing something as small as a microchip and schematic diagrams, or displaying something as large as the night sky.

The map pattern can be implemented as a widget that is a collection of layered images and corresponding text, along with extra control options and map legend data. The example shown in Figure 1 [6] could easily be created with two layers. The first would be a base layer consisting of a picture of the area. The second would be a drawn image of roads that coincides with the base layer’s picture.

![Figure 1 Google Maps Example](image.jpg)

Each layer could have a collection of related attributes and controls, such as:

- Image
- Opacity
- Vertical Pan
- Horizontal Pan
- Vertical Wrap
- Horizontal Wrap
- Zoom qualities
These characteristics might be explicitly coded into the map implementation by the programmer, or they may be configurable in real time by the user at the user-interface level.

As an additional example, the map pattern could portray a floor plan for an entire multi-floor building (Figure 2). The pattern is instantiated by adding a layer for each floor and controlling layer opacity. Clearly, the map is a good pattern because of its ability to be reused in a multitude of situations.

![Figure 2 Floor Plan Concept](image)

The map pattern could be implemented as a template that is instantiated within a GUI builder. When the programmer adds the map widget to the display, a wizard guides the GUI designer through a series of decisions to fill out the code template. The map's layer concept is represented by providing the GUI designer an opportunity to continue to add new layers until all desired layers have been described. For each layer the user adds, the wizard provides a series of options and attributes that can be selected for this layer, and indicates how the layer relates to other layers. (For example, should opacity be defined by the programmer, or should the end-user be able to change opacity dynamically for this layer? Should this layer be registered to pan and zoom independently, or tied explicitly to the corresponding values of other specified layers?) The wizard also allows the layers to be prioritized, stacked, and placed as a cohesive unit. While there are only a few attributes, adjusting them can lead to an optimized “map” for a vast number of different uses.

Suppose there is a need to display weather information on the map shown in Figure 1. By adding a layer containing a dynamic image of a Doppler readout, giving this new layer a high priority (or top location on a visible stack), and changing it’s opacity to 50%, clouds and precipitation would be shown along with the referenced geographic data. This layer could also be deselected so it is not displayed. Clearly, the layers need to be able to display both static and dynamic information, as well as textual and symbolic data. The map pattern template must provide a mechanism for these displays.

Some characteristics certainly need to be determined by the programmer. The map pattern template should provide code for these decision points and implementing the capabilities required. Navigation and perspective are prime examples. Additional navigation options may be necessary when an image becomes too large for the screen. This is where the pan, zoom, and horizontal and vertical wrap attributes come into play. A programmer may add a slider for measurements that do not pan with the other layers (such as distance or angle overlays and other “heads-up display” information). Similarly, a programmer may require layers to pan at different rates (perhaps to approximate differences in distance between successive layers; a rotatable first person perspective and pan is similar to rotating the point of view, so closer objects will pan more slowly than farther objects).

Another common map pattern usage is zoom capability. The programmer will need to decide if the user has a requirement to zoom into the image of layers, or zoom from one layer to another, or both. If the programmer makes interactive blueprints for an office building as shown in Figure 2, then it may be desirable for the zoom function to go up or down one floor, whereas the Google Maps application zooms into the layer images until the image quality cannot be maintained, then switches to another layer.

The map pattern is common and versatile, and its many implementation characteristics make it a clear candidate for incorporation into a user interface design pattern library. This discussion treats the map pattern as a two-dimensional pattern; it is unclear at this time if it is a good idea to directly extend the map pattern into a 3-D (or other multi-dimensional) pattern, or if such usages should eventually be implemented independently.

### 2.2 Timeline Pattern

Timelines are an integral part of today’s professional world, whether project management, corporate planning, military missions, or even just trying to reserve room C-3 down the hall for a conference next Tuesday. This pattern is hidden everywhere in our daily lives as well, and can be seen whenever we look at a calendar or make an entry in our PDAs. At its simplest, a timeline pattern is nothing more than a standard way of visualizing time along with constraints and
obligations. Incorporation of human factors considerations into the implementation adds tremendous value to the timeline pattern as a software component.

The timeline pattern shows time linearly using a collection of simple numberline-like graphs: lines and points correspond to important moments or time spans. Dependent timelines can be used in conjunction with each other to account for objects, people, or events that mutually constrain the same time periods. Consider a scenario where you want to set up an office recognition party for Jim during business hours. The facility has three conference rooms suitable for the event, but they are intermittently reserved for meetings. The party is expected to last an hour and a half. Bob and Dave, Jim’s best friends, must also be able to attend. Jim, Bob, and Dave all have to oversee the rebooting of their servers at 4 p.m., 1 p.m. and 3 p.m., respectively, on top of their various other daily commitments and meetings.

There are several ways someone could deal with scheduling Jim’s party with this information. They could try to “think it through” in their head, but this grows increasingly difficult with each new constraint or additional piece of information. The data could be placed in an electronic spreadsheet, but this format is often sloppy, complex, prone to error, and could take a long time to analyze when cross-referencing all the data. This type of data could be easily visualized with a timeline, such as in the example of Figure 3.

The basis of this pattern is a simple linear reference timeline, shown by the notched line labeled from “8” to “5” in Figure 3. This base timeline will act as the point of reference for all related timelines. Each labeled row (shown below the reference timeline) contains a visual representation of the relationships and constraints of a specific activity to the reference timeline. These activities could also be related to (or constrained by) one-another.

Within these rows, there are three separate markers to identify important points or tracts of time:

1) **Lines**: These generally signify the primary information of an informative line.

2) **Points**: These signify an instantaneous action, restriction, or event.

3) **Backgrounds**: These are similar to lines, however they are placed behind lines and point to better use them in conjunction with the other two markers. Backgrounds are generally used to identify an acceptable period of performance for a desired action.

Lines, points and backgrounds can have constraints placed upon or between them among the timeline’s rows. In Figure 3 the constraints can easily be seen; all people must be available, and at least one conference room must be available for an hour and a half period. As more informative lines are added and constraints grow more complicated, it will be necessary to let a computer keep track of the constraints. This is where a Timeline pattern becomes even more useful.

The example in Figure 3 uses Backgrounds, when all three employees are available, to highlight the meeting rooms and aid in the visualization of the constraints of this party. The slider bar can be moved to find the desired time. If the desired time breaks constraints, the slider bar will change color and an error will be given along the bottom of the timeline. A find algorithm could also be included to allow the computer to find acceptable times for the user.

Just as with the Map pattern, the Timeline pattern involves pre-implemented human factors design. By progressing through a wizard, a programmer can easily create a base timeline, informative corresponding rows, and constraints amongst them to match any scheduling or planning needs.

Further aspects of the Timeline allows for tooltips. This enables detailed, relevant, contextual information of a line or object to be displayed by hovering over the object with the mouse. Tooltips provide easy access to
contextual information without confusing the user and cluttering the screen when the information is not needed.

In addition to the look, feel, and functionality features, a simulation mode can also be easily added. The simulation mode allows the user to graphically move lines and points (time constraints) without changing the actual configuration of the timelines. This capability allows the user to rearrange obligations and experiment with timeline configurations without impacting the actual data. The programmer is responsible for specifying the lines, points, are variables that can be moved, as well defining any restriction upon moving these objects. The simulation mode also implements warning to indicate broken relationship constraints.

A Timeline pattern can be used to represent immense and intricate temporal relationships for extremely complicated scheduling tasks. The United States Air Force intends to use timeline patterns for important coordination of aircraft, flights, and refueling [7]. The timeline pattern described in this section is a parameterized template based on a custom-coded Timeline Tool application (see Figure 4) being developed within the Air Force Research Laboratory. Our goal is to implement the timeline pattern such that the Timeline Tool could be created using the Timeline pattern wizard.

![Figure 4 Timeline Tool Application](image)

### 3.0 Implementation Decisions

Having defined these design patterns, a method of implementing them must be found. Our desire is to implement UIDP objects as drag-and-drop widgets within an existing GUI builder framework.

Implementation platform decisions are largely guided by our selection of the target programming language (i.e. Java, C++, Python, Perl, etc.). The list of potential languages to be used narrowed by our needs: we must be able to create visual objects that are easily manipulated by the typical programmer. The most commonly known object oriented, visually effective languages include Java, C++, and Visual Basic.

Java was selected as the language of choice because:

1) Contemporary programmers are being trained primarily in Java.
2) Many projects used within our organization and by our customers are coded in Java.
3) Java code is platform independent, meaning it can be used on any computer architecture with a JVM.

Once the computing language is chosen, the next step is to choose an integrated development environment (IDE). Not only does an IDE provide us (the pattern implementers) an easy, consistent coding environment by highlighting Java syntax, properly indenting lines, providing real time syntax checking, offering debugging tools, and providing a compiler, it also serves as the baseline for the GUI builder [8].

The selected IDE must support a GUI builder for Java, be extensible, and be relatively inexpensive (if not free). An extensible GUI builder is required, as these are visual design patterns that we are attempting to create. Extensibility is vital towards allowing us to incorporate extensions for our new pattern widgets when they are created.

As long as the GUI builder is a technologically sound product, low GUI builder cost will encourage frequent use of our patterns within the community at large. The cost of the IDE is also a convenience for us as implementers. If the IDE is free, that also takes a burden off of the end-user as they will be able to use the widgets without dipping into their wallets, or filing corporate purchase requisitions.

The most commonly used IDEs that are suitable for our needs include Sun Microsystems’ Netbeans and the Eclipse Foundation’s Eclipse IDE. Both Netbeans and Eclipse are free, extensible, and commonly used in the Java coding world. Eclipse was chosen because many of our own staff and customers are already familiar with this environment.

Eclipse offers a Java programming environment with very customizable windows and views. On top of its default customizable design, a user can download or create extensions and attachments for nearly every aspect of the IDE. This extensibility allows us to search through a wide variety of GUI builder plug-ins for Eclipse. Using the same criteria as for the IDE we compiled a list of seven possible GUI Builders and ranked them according to cost and extensibility. Ranked from best to worst, they are shown in Table 1. GUI Builders from [9] were also taken into consideration.
Visual Editor was chosen for its clear extensibility and free cost. Cloudgarden’s Jigloo was a close second. Unfortunately, its extensibility could not be fully determined without downloading and testing the software. With our own software installation approval process to go through, and limited time, it seemed prudent to use Visual Editor, which is already available and approved for use within our organization. SWT Designer and Jvider were not explicitly tested due to the additional costs. V4All and Matisse for Eclipse may be good GUI Builders, but not enough information was readily available, so they were also dismissed.

4.0 Design Architecture

Implementation of the UIDP concept is currently in the design and initial implementation phase, using Visual Editor as a baseline. Visual Editor is an easy to use Eclipse-based drag and drop GUI Builder which keeps manual GUI coding to a minimum. The Eclipse Visual Editor Platform allows for round-trip engineering: visual modifications are reflected immediately in the code, and code changes are displayed visually. Visual Editor also contains the fundamental Java widgets for graphical user interfaces such as buttons, scrolls bars, text boxes, check boxes, and radio buttons.

Our goal is to use Visual Editor to create UIDP tools as widgets to be placed in its existing library of widgets (Figure 5). These tools can then be used in a drag and drop fashion along with the IDE’s pre-existing widgets. With the addition of a wizard to these new widgets, a fully customized high level pattern could be implemented in a matter of minutes.

Our rationale is that GUI-based software is frequently built using GUI-builder tools, and it seems reasonable to extend these drag-and-drop interfaces to include a collection of “super widgets” that implement the User Interface Design Pattern concepts. The coder would use these UIDP elements within the GUI builder just like the common widget set. UIDP elements themselves will be stored in a local library of coding templates. When the UIDP element is selected and placed on the display, a wizard will guide the programmer through a series of selections that instantiate the templates based on the desired characteristics.

The GUI builder will save the UIDP code along with all other generated code. Wizard selections should be saved (as XML, for example) so UIDP instantiation choices can be revised if needed. This information might be saved as an additional file, or it may be saved as comments within the generated code.

Implementation of the UIDP “widget set” within Visual Editor is module-based. The UIDP capability can be loaded into Eclipse as an additional module extension to the Visual Editor modules. We are currently developing an understanding of how to interoperate with Visual Editor, and also how to add new visual elements and generate code within this environment. Our final loadable module product is expected to be uploaded for use by the entire Visual Editor community.

All widgets within the UIDP module will already be vetted by human factors experts. These experts will be heavily involved in the design and evaluation of UIDP components. As coders who work closely with human factors engineers, we are somewhat uniquely qualified to produce the UIDP elements for this library in such a way that the instantiated and generated library code results in visualizations with built-in human factors considerations that support the end-user’s work environment and needs.
Efficient and effective software design is the grail sought by software professionals and technologists worldwide. Incremental steps are continually being made toward this objective: more accurate computing paradigms and information theory, specialized computing languages, and more capable and more complex software libraries. Together, these enable programmers to generate and maintain software systems faster, with likelihood of fewer bugs, and with greater flexibility than ever before.

A user-interface design pattern library is the next evolutionary step in software libraries. With design pattern tools such as the Map tool and Timeline tool in place of small descriptions of patterns with a snippet of code, an easier method of implementing these documented patterns is formed. Instead, an interactive template will guide the creator to produce the desired user-interface with minimal programming errors, and a Human Factors Engineering component is already included in the code templates.

The implementation architecture is simply an effort to create a proof of concept for User Interface Design Patterns, and these preliminary implementation efforts are still subject to change. Even as this paper was being written, our research has uncovered a potential new method for pattern implementation, based on the Glade User Interface Builder. Glade seems to have the potential to build and add new widgets quickly and easily while allowing these additions to be language independent through the use of XML [10, 11].

The development and documentation of User Interface Design Patterns has the potential to decrease build time, and increase product quality, reduce programming errors, and improve coding efficiency. Most of all, use of these patterns at the coding and design levels will improve the user experience, since these design patterns include a human factors component that supports the ability of the end-users to better complete their work.

5.0 References

SESSION

LATE PAPERS

Chair(s)

TBA
A Formal Framework for Software System Modeling, Analysis and Realization

Xudong He

School of Computing & Information Sciences
Florida International University
Miami, FL 33199, U.S.A.
hex@cis.fiu.edu

Abstract. This paper presents a formal framework for software system modeling, analysis, and realization. The major software architecture design perspective is outlined. Modeling methods for several popular software paradigms are presented. Major validation and verification techniques used in the framework are introduced. A translation approach for generating Java code from a SAM design is described.

1 Introduction

Software has been and will be a major enabling technology for the proper functioning of our society. Many software systems are often mission and safety critical and thus need to be highly dependable. These highly dependable systems need to be highly reliable, efficient, secure, and robust. How to develop and ensure the dependability of these complex software-based systems is a grand challenge.

This paper presents a formal framework for software system modeling, analysis, and realization shown in Fig. 1. This work is related to several major research areas including formal specification, formal verification, software architecture, and translation. Due to the page limit, we omit the discussion of many highly relevant research works of other researchers and the technical details of our own research results.

![Diagram](image)

Fig. 1 – A Formal Framework for Software Development

2 Software Design in SAM

SAM, a formal software architecture model, has been developed at Florida International University in the past decade ([WHD99], [HD02], [HYD07]). Like many architecture definition languages, SAM provides an architecture modeling language to define the behavior of components and connectors, and defines a static architectural configuration through a hierarchical composition structure. Furthermore, SAM also provides a specification language to define constraints representing required behavioral properties. This design and verification integrated approach in software architecture design is a distinct feature of SAM and assures the correctness of the resulting SAM architecture design.
In SAM, a software architecture design is defined by a hierarchical set of compositions, in which each composition consists of a set of components, a set of connectors, and a set of constraints to be satisfied by the interacting components. Fig. 2 shows a graphical view of a SAM architecture model.

A SAM model consists of a set of compositions. Each composition corresponds to a design level or the concept of sub-architecture and a hierarchical mapping relating compositions. Each composition consists of a set of components, a set of connectors, and a set of composition constraints. An element \( C = (B, S) \) (either a component or a connector) in a composition has a behavior model \( B \), and a property specification \( S \). The interface of a behavior model \( B \) consists of a set of ports that is the intersection among relevant components and connectors. A component can be refined into a lower-level composition. A SAM architecture description is well-defined if the ports of a component are preserved (contained) in the set of exterior ports of its refinement and the proposition symbols used in a property specification are ports of the relevant behavior model(s). An element is correctly design if the behavior model \( B \) satisfies the property specification \( S \), denoted by \( B \models S \). The correctness of a SAM architecture description is defined recursively from the correctness of all elements and then the correctness of all compositions.

Petri nets are chosen as SAM’s modeling language to define the behavior models of components and connectors. Petri nets are well suited for modeling concurrent and distributed systems. More importantly, Petri nets have been extended in many different ways to study system performance, reliability, and schedulability; which are essential for complex dependable systems. Over the past forty years, many types of Petri net models have been proposed, which are suitable for specific types of systems and applications. We have used a variety of Petri net models in our study, which include place transition nets (a type of low-level Petri nets) for modeling simple control structures, predicate transition nets (a type of high-level Petri nets [HLP02]) for modeling control and defining data, time Petri nets for studying issues in real-time systems, and stochastic Petri nets for analyzing performance and reliability. Petri nets have a simple graphical representation consisting of two types of nodes (places – denoted by circles and transitions – denoted by boxes or bars), and directed edges. Petri nets are executable. An execution of a Petri net is defined by a sequence of steps starting from an initial marking, a distribution of proper values (called tokens) in places, in which each step is a simultaneous firing of enabled transitions and the firing results in a new marking. Thus an execution can be viewed as a transition firing sequence or simply a marking sequence.

Temporal logic is used to define system properties in SAM. Despite many different types of temporal logic, for example, propositional vs. first-order, linear time vs. branch time, timed vs. un-timed, probabilistic vs. non-probabilistic, it is widely accepted that temporal logic in general is an excellent property-oriented formal method for specifying behavioral properties of concurrent systems. We have used a variety of temporal logics in our prior work including Manna & Pnueli’s linear-time first order temporal logic ([MP92], [MP95]), and Clarke and Emerson’s branch time propositional logic CTL and its extension CTL* [CES86]; and various timed versions of the above temporal logics. To integrate Petri nets and temporal logic in SAM, predicates in temporal logic need to be places in Petri nets. The semantics of a temporal logic formula is defined on an infinite marking sequence of a relevant Petri net by possibly repeating the last marking of a finite marking sequence.

Fig. 2 – A SAM Architecture Model
3 Modeling Methods

High level Petri nets are used to model the structures and behaviors of components and connectors in a SAM software architecture description. The general modeling approach was provided in [HD02] and has been applied to many systems [He05]. In the following sections, we briefly discuss how to use high level Petri nets to model object-oriented systems and agent-oriented systems.

3.1 Modeling Object-Oriented Systems

Object-oriented systems have become predominant software style in the past two decades. We have applied hierarchical predicate transition nets [He96] to model essential features of object-oriented systems [HD01] and a variety UML diagrams used to represent object-oriented systems including class diagrams [He00a] and state chart diagrams and collaboration diagrams [DFH03].

To model a single class (objects), we can use a single high level Petri net such that attributes are denoted by certain places with appropriate types and each operation is defined by a transition with proper constraint specifying the functionality of the operation. For a complicated operation, a subnet may be needed. Fig. 3 shows an example of the partial net structure of a class Reader, which has three attributes and two operations. The access relationships (directed edges) and the types of the attributes and functionalities of operations defined by net inscriptions are not shown here.

To model class relationships, we use net structures and net inscriptions. Simple association relationships (operation calling relationships) can be modeled using the following net structure:

An inheritance relationship is achieved through reusing part of the defined parent net structure and inscription. Polymorphism is realized through either the constraint of an operation with a disjunction that distinguishes a parent or child object or a choice net structure with two transitions representing the parent and child functionalities respectively.

To model a component or a connector consisting of multiple classes, we use the following procedure: (1) define each class with a high level Petri net, (2) define various class relationships with additional net structures and inscriptions, and (3) integrate the above net structures and inscriptions of individual classes and class relationships into a single high level Petri net.

3.2 Modeling Agent-Oriented Systems

Many real world systems have the following main characteristics including multiple actors, distributed control, decentralized data, and required cooperation among the actors. These systems can be naturally and nicely modeled using multiple agent systems (MAS), in which each agent is (1) autonomic – it has its own independent behavior, (2) reactive – it responds to its environment in a timely fashion, (3) pro-active – it takes initiative in carrying out its goal-oriented behavior, and (4) social – it interacts and cooperates with other agents. In the past decade, agent oriented software engineering has emerged as a new research paradigm [WC01] to construct multiple agent systems.
Different high-level Petri net models have been used in modeling MAS including colored Petri nets [XS03] and predicate transition nets [DCX06]. In our recent work [CDH08], we have extended predicate transition nets with channel concepts and employed a net-within-net paradigm [KMR03] for defining the architecture of MAS. The upper level Petri net is used to define the system view of MAS, including the essential mechanisms of interactions, communications, and cooperation among multiple agents. The lower level Petri nets are used to model the behaviors of individual agents. The key idea is the channel concept, which supports the communication between an agent with its environment. We have borrowed the input and output commands in Hoare’s CSP for representing channel expressions. Thus a channel expression is \( n!e \) (output) or \( n?x \) (input), where \( n \) is a channel name, \( e \) is an expression, and \( x \) is a variable. Channel names are the identifications of net tokens and the identification of the system net. A synchronized communication occurs when two fireable transitions at two different net levels have a matching pair of input and output channel expressions, i.e., a transition in the system net with identification \( \text{sys-id contains net-id} \ x \) and a fireable transition in a net token with identifier \( \text{net-id contains sys-id} \ x \). To enforce well-definedness of communications, the channel names in agent nets must be constants; however, a channel name \( n \) in the system net can be a variable ranging over the net tokens’ identifications, which is instantiated with an enabling net token identification. This allows great flexibility and concise representation of synchronized communications. The following example shows an agent net (left) representing the typical behavior of an individual buyer/seller, and a broker net (right) representing system wide coordination.

![Diagram of an agent net and a broker net](image)

The partial net inscription of the agent net includes the constraint: \( R(send) = S!om \), and the partial net inscription of the system net (broker) includes the constraint \( R(receive) = aid?im \). A synchronous communication takes place between an agent with identification instantiated with value in label \( a \) and the system net with identification \( S \) through the simultaneous firing of transition \( send \) in the agent net and transition \( receive \) in the system net.

4 Analysis Techniques

To ensure the correctness of a Sam software design, we have explored a variety of analysis techniques including model checking techniques for finite state systems [HYS04] and testing techniques ([ZH01], [DAC08]) for infinite state systems.

4.1 Modeling Checking Technique

To use a model checking technique, the behavioral model \( B \) (in Petri net) needs to be translated into a state transition system represented in a particular language \( L \) supported by a chosen model checker, denoted as \( L(B) \). Once such a translation is done, both \( L(B) \) and \( L(S) \) (\( S \) is a temporal logic specification) are input into the chosen model checker, which either reports the success of the checking or produces a counter example. We used several well-known model checkers in our previous work [HYS04] including SMV [McM93], STeP [BNS01], SPIN [Hol03], and Maude [FDD08].

The essential ideas of translating a SAM description to a Promela program in SPIN are (1) flatten the hierarchical structure of a SAM software design and integrate the behavior models into a single high level Petri net \( B \), (2) define each type in the underlying specification of \( B \) using a Promela provided type, (3) declare an array variable in Promela for each place in \( B \) and define two Promela functions to add and remove tokens from each place, (4) define the initialization section in Promela to capture the initial marking of \( B \), (5) define functions \( \text{is_enabled}(t, \text{pre-set}) \).
and \( \text{fire}_{t}(\text{pre-set}, \text{post-set}) \) for each transition \( t \) in \( B \) based on the net structure and the transition constraint, and (6) translate property specification \( S \) into a Promela formula. Fig. 6 shows a translation example.

![Fig. 6 – A SAM to Promela translation example](image)

We have developed a tool to carry out this translation and proved the correctness of the translation approach [ACH08].

### 4.2 Testing Technique

Petri nets have an operational semantics and are executable, which supports testing. Testing is an extremely valuable approach for quality assurance when formal verification is infeasible. In [ZH01], we developed a theory and a methodology for testing high-level Petri nets. The theory was based on a concept of behavior observation scheme for concurrent systems, and was formulated using the complete partial order sets. A set of properties of this formulation was proposed. The methodology consisted of four classes of testing strategies: transition-oriented testing, state-oriented testing, flow-oriented testing, and specification-oriented testing were defined. For each strategy, a set of schemes to observe and record testing results and a set of coverage criteria to measure test adequacy are defined. For example, we can test whether a particular transition sequence is feasible or not. In general, testing cannot ensure whether \( S \) is satisfied or not, however can provide useful information and insight into the assurance of \( S \).

In [DAC08], we used SPIN’s simulation capability to validate the testing theory developed in [ZH01]. We have studied several transition-oriented and state-oriented coverage criteria. To test a SAM design, we need to translate the behavior model \( B \) in high level Petri net into a Promela program in SPIN with proper instrumentation code based on a given testing coverage criterion. The translation here is similar to that in the model checking described earlier. The behavior model \( B \) is translated into a process in the Promela program. The types of the underlying specification of \( B \) are translated into integer types and structured types in Promela. Places in \( B \) are translated into fixed-length array variables. The transitions in \( B \) are translated into a process. The transitions are enclosed in a `do .. od` statement. Each transition is defined as a guarded atomic statement within that process. This atomic statement defines the firing rules of the transition. The combination of the `do .. od` statement and the guarded atomic statements ensures the non-deterministic firing of the transitions. The init process is used to assign initial values for the program according to the initial marking of \( B \). The recorded execution history of the instrumented Promela program is used to analyze the measure the coverage criterion.

### 5 Realization Approach

The assurance of the correctness of a software design does not imply the correctness of an implementation. An automatic translation from a software design to implementation is highly desirable. Automatic translation helps in controlling costs, improving productivity and quality. Although automatic programming from a formal specification is in general impossible, translating and implementing software architecture descriptions automatically is viable when architectural designs provide enough details with some reasonable restrictions.
We developed techniques to translate a high-level Petri net into CC++ (a compositional C++ language developed at California Institute of Technology for concurrent and parallel programming) in [He00b], and to Java in [LH98] and [LH00]. These earlier results provide the basis for our translation approach for SAM description shown in Fig. 7. Each behavioral model $B$ is translated into Java code. The hierarchical structure of SAM is translated into ArchJava code, which contains calls to the Java code from every behavioral model $B$. Furthermore, the ArchJava code has cut points identifying events for weaving run-time verification code in AspectJ generated from each property specification $S$. The term rewriting system Maude is used in producing the verification code. The above code segments are compiled together using an ordinary Java compiler.

We have implemented a tool to carry out the above translation [FDH07] and proved the translation correctness [FDA07].

6 Concluding Remarks

In this paper, we presented a formal framework for software system modeling, analysis and realization. We are working on several research areas to improve and fully realize this framework: (1) unifying our prior results based on different Petri net models using standard high level Petri net definition, (2) developing a systematic methodology for integrating a variety of modeling methods and the corresponding verification and validation techniques, and (3) building a software environment integrating other third party tools to support the framework.

Acknowledgements

This work partially supported by the NSF award HDR-0317692 and IIP-0534428. I would like to thank my former Ph.D. advisees Junhua Ding, Zhijiang Dong, and Yujian Fu; and current Ph.D. students Gonzalo Argote and Lily Chang for their contributions to various aspects of this research.

References


Dynamical Proxy for dynamic Services

Artur Tomusiak
Dep. of CSWE
Southern Polytechnic State University
tomusiaka@yahoo.com

Kai Qian
Dept. Of CSWE
Southern Polytechnic State University
kqian@spsu.edu

Taixi Xu
Dept. Of Mathematics
Southern Polytechnic State University
t xu@spsu.edu
768-918-7410
Fax: 678-918-7680

Abstract
This paper presents a dynamic proxy design pattern for designing dynamic service application with the features of automatic re-compilation, re-loading, and automatic execution at run time, without a need to restart the entire application when the server application is updated. A dynamic proxy not only can support dynamic automatic software reconstruction but also support testing related tasks such as performance metrics evaluation, performance optimizations, testing stubs, simulating mock target objects, testing logging, and semantic analysis. The dynamic proxy can make Web services dynamic as well.

We implement a simple Java dynamic code and present its class diagram and sequence diagrams.

Key words: dynamic proxy, design pattern, run time automatic compilation, run time automatic re-loading

1. Introduction

Dynamic behavior changes of class methods at runtime can be implemented by Java dynamic Proxies. There are many proxy patterns people have often used such as remote proxy, virtual proxy, cache proxy, protection proxy, reference proxy, synchronous proxy, etc. The proxy design pattern can be applied whenever you need to control access to an object.

Any proxy acts as an intermediary between client and target object as shown in Figure 1. The Subject interface provides an abstract method request() which is implemented by both realSubject and its proxy. The client accesses proxy’s request() method which has access control over the realSubject.request() behavior. [2][3]

![Figure 1. Proxy pattern](image)

A proxy has authority to act on behalf of client to interact with target object so that client can have an indirect control over the target object in some impossible direct access situations. Proxies provide programmers with mechanisms to make the behavior changes of target objects which are transparent to clients. The proxy above is a static proxy which is created at compilation time.

A static proxy is a dedicated interceptor which binds the interception logic with invocation logic.

Dynamic proxy can separate the interception logic from the invocation logic of a real subject so that it can be generated on-fly. Dynamic
proxies itself can intercept method calls to change the behavior of the called method. Dynamic proxy instance is a same instance of the original class but the behavior of the proxy instance will be different. We implement a dynamic proxy to support a dynamic server so that it allows programmers to change the source code of an application while the application is running and to apply the changes without restarting the application just like how JSP works. [1]

Also, it is possible to write a program that will rewrite its own code so that the application itself would change its own behavior in time. It is a very powerful design pattern. Dynamic proxies can also help with testing applications and developing them during runtime, without a need to restart the entire application. Dynamic proxy is also often used to create mock objects to assist unit testing. A mock object is passed into an object to be tested when the real world working environment is not available at the testing time.[5]

2. Dynamic proxy Basics

Dynamic proxy class implements interfaces which can be dynamically specified at runtime when the class is created. Each proxy instance has an associated invocation handler object, which implements the InvocationHandler interface. The invocation handler intercepts and handles method calls.

Once the proxy class gets an instance of InvocationHandler, it will create the new instance of the given class by calling the Proxy.newProxyInstance() method. A proxy can be created on demand at any point of runtime. When a method of that class is called, the object of a class which implements InvocationHandler interface will actually invoke it.

The class that implements InvocationHandler class has to provide the implementation for invoke() method. That method takes an instance of the proxy object, an instance of Method class, and an array of arguments. The implementation of this method can be different, depending on user’s needs. However, this method should call invoke() method of the Method class and return the result. [4] For example:

```java
private class MyInvocationHandler implements InvocationHandler {
    String className;

    MyInvocationHandler(String className) {
        this.className = className;
        try {
            Class clz = loadClass(className);
            backend = newDynamicCodeInstance(clz);
        } catch ...
    }

    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
            System.out.println("Invoking " + method.getName() + " method.");
            return method.invoke(backend, args);
    }
}
```

3. Creating a new Proxy instance

A proxy instance is returned directly to the client when the client created a new instance of his dynamic class. The sequence diagram in Figure 2 shows that the method of the proxy instance is invoked directly by the client. Therefore, when client creates the new instance of the dynamic class, he actually gets a proxy instance, which has equivalent structure, but might have a different (dynamic) behavior during the runtime. In order for a modified code to be updated on the fly, it needs to be re-compiled. The Javac class has a compile() method that takes an array of files (objects of a File class). Once the class is compiled, it can be re-loaded. How do we know whether or not the class needs to be recompiled (if the source code is changed)? One way implementation is to compile the class every time a method from that class is invoked. This would make the application too slow though. There is a better way. “Last modified” time stamp of the source code can be compared with “last modified” time stamp of the compiled class. If the source code was modified later than last compilation time, it means that it needs to be recompiled. This type of checking could be added to invoke() method of InvocationHandler class and the program will check if the code has been modified each time a method is invoked, or that type of checking could be added to the place in the code, where a new proxy instance is created. Then the changes will take effect only when we are creating new proxy object, instead of each time a method is invoked.

The event-based implicit invocation of re-compilation is another option which will notify the registered proxy to do recompilation as soon as the dynamic class is changed.

We use the second option in our implementation.
In our example, the client is responsible for creating new instance through `DynamicCode` by calling the `newProxyInstance()` method. `DynamicCode` class will then create a new instance of `MyInvocationHandler`, which implements the `InvocationHandler` interface. Then, `DynamicCode` will create new instance by calling `newProxyInstance()` method of `Proxy` class. Proxy class will get the instance from target constructor class and the instance of the dynamic class will be returned all the way back to the client.

4. Method invocation of a Proxy instance

```
MyInvocationHandler will check whether the class has been loaded. If the class is not loaded, then it needs to be re-loaded. The diagram above assumes that the class has already been loaded. Then, an `invoke()` method of Method class will be called. Method class will run that method through `MethodAccessor` class and the result of the invocation will be returned all the way back to the Client.
```

5. Class Diagram of Dynamic proxy and its client

Figure 4 shows the class diagram of a dynamic Proxy example. `MyApplication` is the driver class, which has the `main()` method. It creates a new instance of `MyPanel` by invoking `getMyPanel()` method. That method creates a new instance of `DynamicCode` class, which loads the class (`LoadedClass`) and compiles it with `javac` command if the source code was changed since its last compilation time. At anytime a method from the interface is invoked, `MyInvocationHandler` using the proxy to invoke the method from `MyPanelImpl` class that was just compiled.

6. Example Implementation dynamic proxy for dynamic code

The following is the outline of the implementation:

1. For dynamic classes, we need to create an interface with all the methods that will be dynamically invoked. The name of the interface would be for example `ClassName`
2. Then, we create a class that implements that interface and we give implementation of those methods. The name would be ClassNameImpl.

3. When we want to create a new instance of that class, we do not say “ClassName objName = new ClassName()”, instead, with this statement: ClassName objName = getClassName();

4. We need to implement getClassName() method and newProxyInstance method with the following codes:

   // inside the driver’s class
   private static ClassName getClassName()
   {
     DynamicCode dynacode = new DynamicCode();
     dynacode.addSourceDir(new File("dynamicCodeFolderName"));
     return (ClassName);
     dynacode.newProxyInstance (ClassName.class,
       "packageName.ClassNameImpl");
   }

   // inside DynamicCode class
   public Object newProxyInstance (Class interfaceClass, String implClassName)
   throw RuntimeException
   {
     MyInvocationHandler handler = new MyInvocationHandler (implCalssName);
     return (interfaceClassName);
     Proxy.newProxyInstance
     (InterfaceClass.getClassLoader(), new Class[ ]
       [interfaceCalss], handler);
   }

5. The source code of the dynamic class (eg. ClassNameImpl) should be inside of the dynacodeFolderName directory (followed by a folder with package name)

6. The rest of the code should be inside of staticFolderName directory.

7. In the parent folder of both staticFolderName folder and dynacodeFolderName folder and type in the following command to run the program:

C:\parentFolder>java -classpath . -cp staticFolderName;"{JAVA_HOME}\lib\tools.jar"
DriverClassName

Here is the screen shot before the dynamic code is modified:

![Screen shot before dynamic code modification](image)

package gui;
import javax.swing.*;

public class MyPanelImpl extends JPanel
implements MyPanel{

  private JButton okButton;
  private JButton cancelButton;

  public MyPanelImpl()
  {
    super();
    okButton = new JButton("Ok");
    cancelButton = new JButton("Cancel");
    System.out.println("Buttons created");
  }

  public void addGUIComponents(JPanel p)
  {
    p.add(okButton);
    p.add(cancelButton);
  }
}

The two left buttons belong to MyPanelImpl which is a dynamically implemented JPanel. After editing MyPanelImpl.java in a Notepad, commenting out the ok button (p.add(okButton));, saving and hitting “Refresh” button in our application, the updated code will compile and execute automatically and our refreshed window will look like this:

![Screen shot after dynamic code modification](image)

7. Conclusion

This paper presents a simple Java dynamic service implementation designed by the dynamic proxy pattern. It can respond to dynamic change
automatically at runtime including detecting the changes, re-compilation, re-loading and re-linking. This design pattern can also be applied to any server side application and Web services design if dynamic features are needed. Re-deployments for dynamic server side components and dynamic service components are required addition to re-compilation, re-loading and linking.

Reference

1. Li Yang, Add dynamic Java code to your application, www.JavaWorld.com, 06/06
2. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Proxy Pattern, http://vico.org/pages/PatronsDisseny/Pattern%20Proxy/
3. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley, 1995
4. Proxy API class, http://72.5.124.55/j2se/1.5.0/docs/api/java/lang/reflect/Proxy.html, 2004
Towards a Formal Model for Software Component Composition using Whole-Part Theory

Phillipa L. Bennett and Ezra K. Mugisa

Abstract—Software component combination or assembly has been discussed with respect to part-components being arranged into whole-components. The behavior of the whole-component is constrained by the individual properties and behavior of its constituent part-components and will thus show resultant and emergent properties and behavior. In this paper we discuss configurationality of part-components and whole-components.

I. INTRODUCTION

Software Component combination has been the emphasis of much study in component-based software engineering. In the decomposition of any software system we will invariably see a number of different components or parts; this has been the reason whole-part relationships have been key in the discussion of components. References [1], [2], [3], [4], [5], [6], and [7] discuss Parts and Wholes and characterize their properties in terms of primary, secondary, and consequent properties. The authors also start a preliminary discussion on how these properties interact when composed or assembled in a final system, or larger components, and thus apply the theory of whole-part relationships initially discussed in [8] to software components. Table I lists these properties.

Table I: Properties of Software Components in Whole-Part Relationships (not ordered in terms of significance)

<table>
<thead>
<tr>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstraction</td>
</tr>
<tr>
<td>Binary Nature/Whole-Part</td>
</tr>
<tr>
<td>Encapsulation</td>
</tr>
<tr>
<td>Overlapping lifetimes</td>
</tr>
<tr>
<td>Propagation of operators</td>
</tr>
<tr>
<td>Shareability</td>
</tr>
<tr>
<td>Antisymmetry at type level</td>
</tr>
<tr>
<td>Configurationality</td>
</tr>
<tr>
<td>Existential dependency</td>
</tr>
<tr>
<td>Ownership</td>
</tr>
<tr>
<td>Resultant</td>
</tr>
<tr>
<td>Transitivity</td>
</tr>
<tr>
<td>Asymmetry at instance level</td>
</tr>
<tr>
<td>Emergent</td>
</tr>
<tr>
<td>Mutability</td>
</tr>
<tr>
<td>Propagation of destruction operators</td>
</tr>
<tr>
<td>Separability</td>
</tr>
</tbody>
</table>

In component-based software engineering, we are interested in being able to use these properties of software components, to discuss and/or to predict the behaviors of the final system or what kinds of components may be assembled. More specifically we ask the following questions:

i. What will be the resultant or emergent set of properties of the final system of assembled components?

ii. What can we say about the behavior of the final system based on how the properties of the individual components interact?

iii. How will we know what properties of a component will indicate that it is “composable” with another?

The first and second questions address how we will look at and argue about the final system i.e. post-composition, while the third question addresses pre-composition criteria. To answer the questions in all three cases, we must have an understanding of the properties of components and perhaps form a theory of component combination within which we can discuss the various questions. This paper looks at the third question by undertaking an initial investigation, and in order to make precise statements we use the formal notation from the formal specification language Z. This discussion forms the basis of work that we will undertake to make more precise statements towards forming a theory for software component combination based on the Whole-Part theory.

Section II gives a review of components, section III begins the formalization of configurationality, and section IV shows some further directions.

II. COMPONENT REVIEW

A. Component Discussion

We want to examine how the properties of software components may be used in design and implementation activities to predict the properties of resultant systems. We therefore need a definition of components that would form a basis for this examination. We see in [9] an extensive search into how different groups define components, and we also see the authors emerge with a best practice definition that we will also use. It states:

“A software component is a software element that:

i. Exports and/or imports services,

ii. Has well defined interfaces, and

iii. Is a unit of composition”

We see in this definition that a software component exports and/or imports services and this is equal to saying that a
component may provide and/or require operations. These provided and required operations make up a component’s well-defined interface. Also we see that from (i) and (ii) in the above definition, we have an artifact that is a unit for software composition. Added to this definition is that a software component may be defined recursively, i.e. it may be made up of other software components.

The above definition alone does not tell us enough about components to say how they communicate to provide services. In [10, page 194], we see another definition of a component: a

“replaceable part of a system that conforms to and provides a realization of a set of interfaces”.

Together with the first definition we can say that a component therefore gives us a view of the service it provides through its interface. Further we see that accessing the service of these components is through ports [10, page 194] defined on the boundary of the component, thus the internal structure is not externally visible.

Communication, or the provision of component services, is through provided or required interfaces on a component’s external port. It is hence only those services published on the ports that will be accessible and this implies that the component may have additional operations that are hidden. Consequently, a component may publish many ports and attached interfaces. The internal structure of the component is not accessible, hence no changes can be made. We are therefore guaranteed that the component will not change. Its service is guaranteed. This is a desirable property of Component-Based Software Development (CBSD).

The notion of ports imposes on a component an additional constraint, that of encapsulation: in this way the only services (provides or requires) that a component makes accessible are those it publishes through any of its ports. An example of such a component is provided in figure 1 (Unified Modeling Language, UML, 2.0 diagram) where component A provides two services and requires two.

This configuration gives the component developer the choices of whether to make certain operations in its service (interface) visible/accessible or not.

B. Component Properties

Part-components having different sets of properties (taken from table 1) combine to produce some resultant and or emergent properties. A resultant property of a whole-component can be directly computed from the properties of the part-components, while an emergent property is not directly computable from the set of interacting properties in the whole-part relationship [1]. We will ultimately be interested in how to compute these resultant and emergent properties.

C. Whole-Part Relationships

A component is also characterized thus: “the global state of a whole-component is recursively the aggregation of the states of all its Part-components” [6]. This implies that a larger whole-component may contain smaller part-components. The services offered by the whole-component are also published on external ports. It is the case that these externally visible ports are connected to the ports of their internally hidden part-components and there are interconnections of the part-components inside the whole-component. Part-components also publish their services through their ports. The interconnections of part-components within a whole-component provide what is internally needed to offer to the development environment an aggregated service through the port of its owning whole-component.

III. CONFIGURATIONALITY

Having formed an understanding of components and Whole-Part relationships we move now to our examination of the properties. We start with configurationality.

Barbier and Henderson-Sellers, [3] describe configurationality as “…parts bearing a functional relationship to the whole…” and implies that there is no haphazard arrangement of parts. This property is clarified further in [1] and says:

“…for a bundle of whole-part relationships having the same Whole type, any Part type is involved in at least one connection with the other Part types in the bundle”.

Both definitions point to configurationality as a relationship with how the Parts relate to the Whole, and/or with other constituent Parts in the same Whole such that we may draw two conclusions:

i. Parts connected together to form a Whole have configurationality with respect to the Whole, and 

ii. The connections among Parts show that they are configurational with the Part(s) that they are connected to in the (same) Whole.

1 Bundle is used here within the context that “a Whole object often consists of several Part objects that do not come from the same type” [1]
Configurationality essentially defines the precondition on parts and wholes when we wish to build a new component software system made from components, in CBSD. We now have a clearer view of configurationality so that we are able to say that it is the interfaces of the respective components that are connected and thus have a functional relationship between them. This description together with how we have defined components earlier, allows us to expand the description of configurationality:

A component \texttt{compA} may have configurationality with another component \texttt{compB} if \texttt{compA} and \texttt{compB} are participating in a functional relationship such that either:

i. \texttt{compA} is a Part component and is providing its service to the Whole Component \texttt{compB}, and its required interfaces are fulfilled in \texttt{compB}, or delegated on \texttt{compB}'s required interface, or

ii. \texttt{compA}'s provided interface is connected to \texttt{compB} required interface, or vice versa

We may also say that the configuring of \texttt{compA} and \texttt{compB} may form a component system, or new component \texttt{D} that may be placed in a repository. Figure 2a and 2b below (also UML 2.0 diagrams), show these relationships.

![Figure 2a and 2b](image)

\textit{Fig. 2 (a): Defining Configurationality with Whole, (b): Defining Configurationality through Compatible Interface Connection.}

Figure 2a and 2b are representations of \texttt{compA} configured with \texttt{compB} and should not be interpreted as giving the full picture of the configuration of the components.

\section*{A. Formalizing Configurationality}

In building our model for software component composition, we want to be able to say precisely what it means for components to have configurationality. To do this we will apply the Z formalism to our discussion.

Our research model takes the view that components will be stored in a repository [12], [13], [14], and [15]. We will be able to perform the following actions in the repository:

i. add a component

ii. remove a component

and perform queries to determine

iii. if two components can be configured

iv. if a component’s required interface(s) can be fully realized by components existing in the repository, etc.

This is important when we have decided to use components from our repository to build component systems. We see here that the formalization of the repository will also be useful. This formalization is discussed below.

\section*{Given types}

We are not yet concerned with the details of a component’s functional description and actual content, hence we declare some given types:

\begin{verbatim}
[NAME, DESCRIPTION, CONTENT]
\end{verbatim}

where:

i. \texttt{NAME} – represents an identifier

ii. \texttt{DESCRIPTION} – represents a functional description of an interface

iii. \texttt{CONTENT} – represents contents of a component, e.g. precompiled statements that carry out the function defined in the description.

\section*{Interface}

We declare a type \texttt{Interface}:

\begin{verbatim}
__Interface[NAME, DESCRIPTION]__
name: NAME
desc: DESCRIPTION
\end{verbatim}

This type as a generic schema helps us to reuse this definition in different contexts [11, page 162] and is shown in the schema type \texttt{ProvidedInterface}:

\begin{verbatim}
ProvidedInterface \equiv Interface[NAME, DESCRIPTION]
\end{verbatim}

and required interface:

\begin{verbatim}
RequiredInterface \equiv Interface[NAME, DESCRIPTION]
\end{verbatim}

\section*{Component}

We declare another type schema \texttt{Component}:

\begin{verbatim}
__Component__
name: NAME
content: CONTENT
required_interfaces: \* RequiredInterface
provided_interfaces: \* ProvidedInterface

\# required_interfaces + \# provided_interfaces \geq 1
\end{verbatim}

A component has
Adding a component to the Repository

The following operation schema is used to add a component to the repository:

\[
\text{AddComponent} \quad \Delta \text{CompRepository} \\
\text{comp1}?: \text{Component} \\
\text{comp1}? \cdot \text{name} \in \text{componentNames} \\
\text{components}' = \text{components} \cup \{\text{comp1}?\} \\
\text{componentNames}' = \text{componentNames} \cup \{\text{comp1}? . \text{name}\}
\]

Adding a component to the repository changes the repository and consists of determining that it’s identifier, name, is unique and adding it to the set of components and registering its identifier.

Removing a component

We define another operation of removing a component from the repository:

\[
\text{RemoveComponent} \quad \Delta \text{CompRepository} \\
\Delta \text{ShowComponentRequiredMappings} \\
\Delta \text{ShowComponentProvidedMappings} \\
\text{comp}?: \text{Component} \\
\text{rcomp}' = \text{pcomp}' = \text{comp}? \\
\text{componentRequiredMappings}' = \text{componentRequiredMappings}' \setminus \{\text{comp}? . \text{name}\} \\
\text{ConfigurationalityMap}' = \text{ConfigurationalityMap}' \setminus \text{comp}? \\
\text{components}' = \text{components}' \setminus \{\text{comp}?, \text{comp1}?. \text{name}\}
\]

Removing a component from the repository includes:

i. de-registering it’s identifier,
ii. removing all it’s mappings from the configurationalityMap, and finally
iii. removing it from the set of components in the repository componentRequiredMappings, which returns from the configurationalityMap all the provided interfaces that each required interface may be configured with, is defined in ShowComponentRequiredMappings:

\[
\text{ShowComponentRequiredMappings} \\
\text{CompRepository} \\
\text{rcomp}?: \text{Component} \\
\text{componentRequiredMappings}!' = \text{componentRequiredMappings}' \setminus \{\text{rcomp}? . \text{name}\} \\
\text{ConfigurationalityMap}' = \text{ConfigurationalityMap}' \setminus \text{rcomp} \\
\text{components}' = \text{components}' \setminus \{\text{rcomp}?, \text{rcomp}? . \text{name}\}
\]

The schema for ShowComponentProvidedMappings returns componentProvidedMappings. It is not shown here but is similar to ShowComponentRequiredMappings.
Configurational
Our first query is the test for configurationality. We test if components are configurational:

\[ \exists \text{comp1?, comp2?} : \text{Component} \]
\[ \exists a, c : \text{RequiredInterface}; b, d : \text{ProvidedInterface} \]
\[ a \in \text{comp1?.required\_interfaces} \]
\[ b \in \text{comp2?.provided\_interfaces} \]
\[ d \in \text{comp1?.provided\_interfaces} \]
\[ c \in \text{comp2?.required\_interfaces} \]
\[ \text{comp1?} \in \text{components} \]
\[ \text{comp2?} \in \text{components} \]
\[ ((\text{comp1?.name, a}) \rightarrow (\text{comp2?.name, b})) \in \text{ConfigurationalityMap} \]
\[ \lor (\text{comp2?.name, c}) \rightarrow (\text{comp1?.name, d}) \in \text{ConfigurationalityMap} \]

Two components, comp1 and comp2, are configurational if there is a mapping in the configurationalityMap between a required interface of comp1 and a provided interface of comp2. The test for configurationality does not change the contents of the repository.

Utility of the ConfigurationalityMap
For building component systems from the components in the repository we may be interested in determining whether the components that we want to use can be fully realized from the components in the repository. To do this we have defined other queries on the ConfigurationalityMap:

i. If every required interface in a component, comp1, can be found in the configurationalityMap or the component has no required interface then we say that the component can be fully realized from the components in the repository; we say it is fullyMapped:

\[ \forall r1 : \text{RequiredInterface} \mid r1 \in \text{comp1?.required\_interfaces} \]
\[ \lor (\text{rcomp} = \text{comp1?}) \]
\[ \lor (\text{rcomp}\text{.required\_interfaces} = \text{[]}) \]
\[ \lor ((\text{comp1?.name, r1}) = \text{dom componentRequiredMappings!}) \]

ii. For a component, if for all it’s required interfaces (where there is at least one defined), there was no mapping found in the configurationalityMap then this component cannot be realized from the components in the repository; We say it is not mapped:

iii. For a component, for all its required interfaces, only some were found to have mappings in the configurationalityMap then the component can be partially realized from the components in the repository.

IV. Conclusion
As a first stage, we made a formal examination of what it means for a component to be configurational. Questions i and ii that were posed in the introduction forms our further work in building the model for software component composition.

References

[9] Agnes Freda Namulindwa Lumala, Need for a Development Environment to Support a Heterogeneous Repository of Software Components, Paper to be published


Method for Service-Oriented Analysis and Design

A. Henrique Shoiti Fugita¹, and B. Kechi Hirama²

¹Polytechnic School, University of São Paulo, São Paulo, SP, Brazil
²Polytechnic School, University of São Paulo, São Paulo, SP, Brazil

Abstract - The need for integrating heterogeneous Information Technology environments and for obtaining greater business agility have been leading organizations to adopt Service-Oriented Architecture. However, service orientation demands changes in existing methods for application analysis and design. Some service oriented development methods have been proposed, but have not yet converged towards standardization. In this context, a critical analysis of existing proposals is needed and possibly they can be combined in a consolidated proposal. This work proposes a method for service oriented analysis and design conceived from the analysis made.

Keywords: Software Engineering, Analysis and Design Methods, Service-Oriented Architecture (SOA)

1 Introduction

In order to design applications that are effectively service-oriented, and thus presenting the expected benefits brought by SOA, it is necessary to pay special attention to identification and specification of the services that compose them. These services that will support the business processes and will satisfy the requirements must be adequately defined, so that they effectively present reusability attributes, flexibility and alignment with business goals. Moreover, the services must be specified so that they can be published to a repository of reusable services and be orchestrated to support business processes.

Object-oriented and component-based development paradigms cannot be directly applied to service-oriented architectures [7]. To guide the specification, construction and refinement of business processes from services, a service-oriented development method is necessary.

This method must begin with the business requirements and goals to identify and design consistent business processes that use services available in the Information Technology (IT) infrastructure or new services to be developed. In case of using new services, the method must encompass activities to specify these services so that they can be implemented using some object-oriented or component-based method.

Such method must be predictable and repeatable and must use widely adopted tools and notations for modeling and representing information and for generating artifacts. This way, the adoption of the method would be simplified and would bring less impact to existing development processes.

2 Service-Oriented Architecture

Service-Oriented Architecture (SOA) is a paradigm for building and integrating applications based on modular elements called services. A service, SOA fundamental unit, is a software element that performs a specific business activity, is self-contained and can be used by a client as a “black-box” [6]. A SOA enables a distributed computing infrastructure, as services can be provided and consumed inside an organization or between organizations, through communication networks such as the Internet [8].

Service orientation paradigm derives from the separation of concerns theory [1]. According to this theory, a large problem must be divided in smaller concerns so that the solution can be decomposed in various parts, each part addressing a certain concern. Just like object orientation, service orientation is a way of achieving separation of concerns, with a set of principles which determine the expected characteristics of a service.

- Reuse: Reusable services are the ones that can add value to more than one business process context, inside or between organizations. For that reason, they should support more than one consumption pattern [1].

- Formal contract: A service must have a well-defined formal contract which describes the functionality it provides and encapsulates the specific details of its implementation [5]

- Loose coupling: A loosely coupled service can have its implementation replaced, modified or evolved without causing impact to the service consumers [5]

- Statelessness: A service should avoid storing state and context information [1]. This means that the service response to a message should be independent from previous messages processed by the service.
3 Service-Oriented Development Methods

Service orientation can bring many benefits that drive organizations to adopt SOA. However, to achieve all the promised benefits of SOA, it is important that the services have the appropriate characteristics and follow the principles of the service orientation paradigm. It is necessary that service Analysis and Design guarantee that the final product – the service specification – has characteristics as reusability, loose coupling and adequate granularity, besides adding value to the business.

Some works that relate development methods and SOA have been published so far. However, existing approaches have not yet converged towards standardization, which leaves room for further discussion and new proposals [9]. Therefore, an analysis of the existing proposals and the benefits brought for each one becomes necessary. Based on this analysis, it is possible to instantiate a new method that aggregate characteristics of existing methods. Some service-oriented analysis and design methods found in literature have been studied and are described in the following sections.

3.1 Papazoglou and Van Den Heuvel

Papazoglou and Van Den Heuvel describe the method called Web Services Development Life Cycle [7], that is partially based on other already established methods such as the Rational Unified Process, Component-Based Development and Business Process Management (BPM). According to the authors, a service-oriented development method must focus on the business processes, which are the reusable elements independent from applications and platforms. The method follows two principles during its life cycle to assure that the developed business processes have the expected attributes: loose coupling and high cohesion between services.

The presented method consists of a cyclical process, executed in iterations. This cyclical approach allows for a continuous discovery, creation and implementation process, with each iteration adding more artifacts to the solution, in predictable and repeatable way. Papazoglou and Van den Heuvel’s method encompasses all phases of service life cycle, starting with analysis and design, continuing to implementation, execution and provision and finishing with monitoring, which restarts the cycle.

The method offers service identification and realization alternatives from top-down, bottom-up and meet-in-the-middle. However, meet-in-the-middle approach is only considered as alternative of realization, and not considered for service identification. The method also uses BPM concepts, such as “AS-IS” and “TO-BE” process modeling activities.

In the gap analysis activity, existing resources are evaluated in terms of reusability. Additionally, non-functional requirements are taken in account during service and process specification activities, when service policies and Service Level Agreements (SLAs) are defined.

In service design phase, conformity to service-orientation principles is verified: cohesion and loose coupling. The open standards adopted by the method are the WSDL, XSD and WS-BPEL.

3.2 Rational Unified Process Plugin for SOMA

The Rational Unified Process (RUP) is Software Engineering process developed and commercialized by IBM. It consists of a method for managing activities and roles in a software development organization. RUP purposes to discipline development in order to deliver high quality software according to customer requirements within predicted time and budget [3]. RUP is based on proven software project good practices [4].

To use RUP in SOA projects, IBM has developed a plugin (RUP plugin for Service-Oriented Modeling and Architecture - SOMA) [10] that contains activities, roles and artifacts related to service-oriented development. These elements must be incorporated to the organization’s customized process. RUP plugin for SOMA brings modifications to some activities of RUP basic life cycle and adds new concepts, roles, activities and artifacts. Most additions brought by the plugin are applied to the first phases (Inception and Elaboration) of the cycle and involve Analysis and Design disciplines. Two main activities were added to deal with service-oriented development: Service Identification and Service Design.

RUP plugin for SOMA’s approach is essentially top-down and uses some BPM concepts, because it considers identifying services from business process models, however does not present any business process orchestration activity. Service Identification considers existing resources, such as legacy applications, and other sources of identification.

Non-functional requirements are handled in service policy specification. The method uses the service candidate concept, which connects Service Identification and Service Design. In this method there are no activities validating service orientation principles.
The method offers diverse approaches for service identification and also uses the service candidate concept. Like Erl’s method, Marks and Bell’s is an open method.

4 Requirements for a Service-Oriented Analysis and Design Method

From the analysis of the existing methods, it is possible to infer a set of requirements that a service-oriented analysis and design method should address. Next, such set of requirements is described. The requirements are identified by codes R1, R2, R3, etc. for reference throughout the text.

- **Meet-in-the-middle approach (R1)**: A service-oriented development method should not be limited to a top-down approach, considering only business requirements and goals for service identification and specification. Nor it should be purely bottom-up, simply encapsulating application functionality in services. A method must balance both approaches, resulting in a meet-in-the-middle approach.

- **Business Process Management compatible (R2)**: SOA can bring many benefits if adopted together with BPM (Business Process Management) practices. The WS-BPEL standard allows implementation of SOA/BPM solutions. The BPM approach increases the alignment of developed services with business goals and strategy. A service-oriented method should consider this synergy and incorporate BPM techniques.

- **Existing resources (R3)**: One of the basic service-oriented principles is reuse. Thus, a service-oriented method must leverage existing implemented logic, in the form of available services in the portfolio or in the form of legacy applications, which would have to be transformed into services.

- **Non-functional requirements (R4)**: A critical point in SOA is the non-functional requirements (NFRs), also called Quality of Service (QoS) requirements. In this area, a series of open standards are under development, such as standards to specify aspects of security, performance and transactions. A considered method must approach NFRs, possibly adopting specific standards, such as WS-Security, WS-Policy and WS-Transaction.

- **Various service identification approaches (R5)**: The method should be flexible, allowing analysts to identify service candidates from process models, business use case descriptions and data models for top-down identification, or legacy applications and existing services for bottom-up identification.

- **Various service realization approaches (R6)**: The method must take in account various possible scenarios for service realization and analyze their advantages and
disadvantages, which enables the service designer to decide which approach to adopt.

- **Service candidate (R7):** The use of service candidate concept allows a meet-in-the-middle approach, where services identified from business modeling can be reviewed according to existing technological environment constraints or other limitations.

- **Service-orientation principles (R8):** The method must enclose activities and practices to assure that developed services are adherent to SOA principles, such as reuse, interoperability and loose coupling.

- **Open Standards (R9):** Artifacts and notations used throughout the method should be based on open standards such as WSDL for service interface description, XSD for data format definition, WS-BPEL for business process orchestration and BPMN for process modeling of processes.

- **Existing implementation methods (R10):** The method outputs (service specifications) should serve as input for existing implementation methods [8], such as object-oriented or component-based development.

- **Open Method (R11):** The method must be open, that is, not-proprietary. The access to its description and its use must be free.

It can be noticed in the analysis of the existing methods that none of them completely addresses all of the raised requirements, as it can be seen in Table 1. Each one emphasizes different sets of requirements.

## 5 Proposed Method

As it can be inferred from the analysis of the existing proposals, it is necessary to consolidate the methods in a unified approach that covers all of the requirements of a service-oriented analysis and design method. This work describes the Method for Service-Oriented Analysis and Design, a method proposal that is under elaboration as part of a dissertation thesis, based on the analysis of existing methods and principles of service-orientation. The method is designed to be open, fulfilling de requirement R11, previously described. The method intends to offer flexibility in its application by means of iterations during the activities of Analysis and Design. The described life cycle begins with Business Modeling, continues with Analysis and Design, following with activities of Implementation and Test, as it can be visualized in Figure 1.

In the following sections, the activities of the method are described, beginning with Business Modeling that precedes service-oriented Analysis and Design.

### 5.1 “AS-IS” and “TO-BE” Process Modeling

Business modeling occurs before Analysis activities of the method and is divided in two activities. Business modeling activities are performed by business or process analysts. In the “AS-IS” Process Modeling activity, the current state of involved business processes is determined. The resulting process model represents how they are executed before the SOA project.

In the “TO-BE” Process Modeling activity, the business or process analyst performs simulations and considers modifications to the process with the intention to obtain performance improvement and cost reduction. Such modifications can affect process structure (flow) or automate part of the process. As product of this activity, a new process model is elaborated.

The models can be represented as business use cases or using a process notation, such as BPMN (Business Process Modeling Notation), that can be directly converted to WS-BPEL language by various off-the-shelf tools. The use of open notations as use cases and BPMN fulfills the requirement R9.

### Table 1: Requirements addressed by analyzed methods

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Papazoglou &amp; Van den Heuvel</th>
<th>RUP Plugin for SOMA</th>
<th>Erl</th>
<th>Marks &amp; Bell</th>
</tr>
</thead>
<tbody>
<tr>
<td>(R1) Meet-in-the-middle</td>
<td>Partial</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(R2) BPM Compatible</td>
<td>X</td>
<td>Partial</td>
<td>Partial</td>
<td></td>
</tr>
<tr>
<td>(R3) Existing resources</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>(R4) Non-functional</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(R5) Identification</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(R6) Realization</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(R7) Service candidates</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>(R8) Principles</td>
<td>Partial</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(R9) Open standards</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(R10) Implementation</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(R11) Open Method</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
5.2 Service Candidate Identification

Service Candidate Identification is the first activity of Service-Oriented Analysis and receives as input the “TO-BE” Process Model produced during Business Modeling. For using a process model as a source for service identification, the method covers the requirement R2, which means that it is compatible with the BPM approach.

The “TO-BE” process model must be decomposed in a series of low granularity steps and then the SOA Architect must identify which steps could be automated. Process steps that must be performed by a person will not be considered in this stage, because they will not be executed by service operations.

The SOA Architect must list the process steps that will be automated, so that each one of these steps will correspond to the invocation of a service operation. The operations that relate to each other must then be grouped, so that each group of operations will belong to a service candidate. Operations must be grouped so they share the same logical (semantic) context or they work with the same data entities [1]. While using the concept of service and operation candidates, the considered method fulfills the requirement R7.

5.3 Gap Analysis

In Gap Analysis, service candidates are compared to existing resources and the possibilities of reuse are evaluated [7]. Each operation candidate is verified against existing applications, leading to the following scenarios:

- Operation realized by a service: The functionality of the operation is already implemented by an existing service, which can then be reused.
- Operation partially realized by a service: The service can be reused, but it will be necessary to modify the existing service or to implement some mediation logic.
- Operation realized by legacy application: To be able to be used by process, this legacy functionality will have to be encapsulated in the form of a service.
- Operation partially realized by legacy application: To be able to be used for the process, this functionality will have to be encapsulated in the form of a service and some type of mediation logic or integration will have to be implemented.
- Not existing operation: A new service must be created or a new operation must be added to an existing service.

The grouping of operation candidates in services may need to be revised after Gap Analysis. This activity addresses requirement R3 when considering existing resources. By combining top-down identification and bottom-up realization, a meet-in-the-middle approach is characterized, fulfilling requirement R1. Together with Service Candidate Identification, it covers requirement R5 so that services are identified from process models and existing applications.
5.4 Realization Analysis

Based on Gap Analysis, the Realization Analysis evaluates the realization scenarios for each service candidate in terms of cost, risk and return of investment. For each gap scenario, the following considerations must be made:

- Operation realized by a service: This is the scenario with the lowest risk, causing only the cost of reusing the service. This scenario leverages return of previous investment.

- Operation partially realized by a service: The existing service can be modified, which brings a risk (existing impact in other processes and services). Alternatively, mediation logic can be implemented to reuse the service in its current state.

- Operation realized by legacy application: This scenario demands the creation of a service that encapsulates the functionality of the legacy application.

- Operation partially realized by legacy application: This scenario also offers the possibility of modifying the legacy to correspond to the operation candidate, demanding only the creation of the encapsulating service.

- Not existing operation: The risk is low, therefore there is control over the new service specification and realization. The cost corresponds to the necessary effort to specify and to implement the new functionality.

After the analysis of all the operations candidates, the SOA Architect must decide which realization scenarios to execute according to the analysis. This activity fulfills the requirement R6, when offering diverse service realization scenarios, depending on the availability of reusable resources.

5.5 Service Specification

Once the service candidates are defined, the Design phase begins. The objective of this phase is to create service specifications that address business requirements and are adherent to service-orientation principles [1]. The first activity of this phase is Service Specification. For each service candidate, operations and messages have to be defined. The format of the messages must be defined in the form of XSD schemas and the interfaces, represented by WSDL documents, fulfilling requirement R9.

For each service, functional specifications of each operation and service policies will also be defined.

- Operation realized by a service: In this scenario, the service already exists with defined interface and no additional specification is needed.

- Operation partially realized by a service: If the existing service is to be modified, it can be necessary to modify its interface and/or the functional specification of the operation to be called. In case mediation is used, its integration logic must be specified. In both cases, interface specification is made top-down, from the service candidate.

- Operation partially realized by legacy application: In this scenario, the interface is specified from bottom-up, having its messages and operations defined according to the characteristics of the reused legacy application.

- Operation partially realized by legacy application: If the legacy is to be modified, the interface will derive from bottom-up.

- Not existing operation: In this scenario, service interface and functional specification of its operations are elaborated top-down.

The functional specifications of each operation can be defined in the form of white-box system use cases, which contain implementation details. Defined in this form, services can be directly implemented using object-oriented and component-based techniques, covering requirement R10.

Service policies specify the non-functional requirements of the service, such as security, transactional behavior and audit log, and requirements of service level, by means of SLA contracts. The specification of non-functional requirements fulfills requirement R4.

5.6 Principles Verification

In this activity, the SOA Architect or the SOA Designer verifies the service specifications in order to evaluate their conformity to service-orientation principles and to address the requirement R8.

To assure that the services are reusable, the SOA Architect or the SOA Designer checks if it is not necessary to add operations or to modify the already existing ones so that the service can be useful in another context other than the business process under development [1]. Even if not all operations need to be implemented immediately, it is recommended that they are specified early to minimize the need for latter changes in the service interface.

Service granularity is another point of attention, therefore if a service is fine grained (small amount of encapsulated logic), is has a greater possibility to be used in more than a business process. To achieve autonomy and low coupling in a service, one must assure that the encapsulated logic is self-contained and does not have any dependencies with other services or processes.
To assure that services are stateless, the amount of state information stored in the service must be minimized. Instead, it must be stored in the messages or in the orchestration logic of the business process.

5.7 Service Review
In this activity, the SOA Designer verifies if the specified services effectively follow the service-orientation principles. As a result, some modifications to the specified service may be necessary. In this case, the process goes back to the Service Specification activity, so that the interfaces and operations can be redesigned as needed to conform to the service-orientation principles.

5.8 Service Orchestration
Once the services and their WSDL interfaces are specified, the services need to be orchestrated. The “TO-BE” process model represented in BPMN notation is converted into a WS-BPEL process model. This model must be reviewed so that service invocations are adequately orchestrated with the reviewed interfaces. This activity covers the requirements R2 when using the BPM approach and R9 when using the WS-BPEL standard.

5.9 Implementation and Test
The Implementation and Test phases that follow Design phases are executed by Developers, who will receive the Service Specification and will implement the necessary Web Services and components. After these phases, the traditional cycle of object-oriented and/or component-based development can be followed.

6 Conclusions
The evaluated service-oriented analysis and design methods do not address all of the considered requirements. However, each method brings a different approach that emphasizes certain kinds of requirements.

It becomes necessary to consolidate the existing methods to conceive a method that indeed fulfills all the requirements. This work proposes a method to unify the existing approaches. The proposed method contains activities to address each one of the considered requirements, including elements from the analyzed methods and following the way each method covers the requirements.

The research work is currently in progress with the accomplishment of tests and pilot projects intended to validate the method and to evaluate its applicability in different projects and contexts.

7 References
Requirement Preparation before awarding the Work to Vendor

Puja Sahuja  
Software Division, RMSI Pvt Ltd, Noida, Uttar Pradesh, India

Abstract - For a project to be successful it is of utmost importance that the stakeholder share common understanding regarding the objectives and goals of the project. A thorough brainstorming by the client before the start of the requirements phase with the vendor would eventually result in more clarity on the client side. This would help the client prepare themselves to finally interact with the hard core technical guys from the software industry.

Keywords: Preparing client for pre-requirements phase

1 Introduction

Being a client inside or outside the IT industry there is a need to understand how the projects are executed in a standard SDLC Software Development Life Cycle. To make the vendor understand the requirements it is important for that the clients be fully prepared with what do they want to get developed. To deal with the Requirements in the standard SDLC Software Development Life Cycle the most critical phase is the Requirements Definition/Analysis. This phase lays the foundation of any project. This paper focuses on what activities should be performed at the client side before awarding the work to the vendor.

1.1 Challenges in Industry

The software providers and their clients have separate processes with different rationales of Requirements Specification. This results in mismatch between the client and the vendor in terms of what software is expected to be designed and developed by the vendor. Technology Capturing and analyzing the requirements correctly in limited time frame is one of the major challenges faced by all Software Development teams. On the vendor side the developers struggle with the issues of not so clear requirements, changing or evolving requirements which pose a threat to the entire project’s delivery. It may result in effort overrun, rework & productivity loss.

2 Case Study

A project from the Health Care Domain required the vendor to create an advanced Event Management System. This system had to be designed to handle:

- Critical information from each outbreak event starting from initial response report through response
- Include databases for outbreak events & partner institutions
- Maintain database for consultants available for international response, supplies, inventory and contractual details

The project being on a large scale required clear sense of direction from client side on requirements and good domain understanding by the vendor.

3 Challenges Faced by Clients

There could be different challenges as faced by various clients under varied circumstances. Taking the case study example the challenges that were faced by the client were as listed:

- Requirements Mismatch
  This can be one of the most common of all challenges being faced by many others as well. The requirements mismatch can result in the phase going in iterative mode wherein both the client and the vendor struggle with the not so clear goals & objectives. This situation can be explained in the figure below:
The requirements features as required by the client have not been fully catered to in the vendor list whereas additional requirements have been. This is called as “Gold Plating” concept. This would result in additional effort from the vendor side and not so satisfied client.

- **Evolvement of concept/requirements:**
  It was observed that the requirements from the module owners were evolving during the brainstorming sessions with the vendor. This was proving a major bottleneck as none of the requirements could be finalized and this would result in more iterations and decrease in quality.

- **Involvement of multiple owners for a single module:**
  As all the projects are divided in various modules depending on the size & criticality of the project, there is a need to have separate owner for each module. Having multiple people explaining a concept or a problem statement brings in ambiguity.

- **Schedule slippage:**
  This could be a result of the above mentioned issues or due to unavailability or resources. In order to minimize this risk it is necessary to understand the need quickly & deliver quality product.

- **Non-Technical stakeholders from client side:**
  Technical capabilities of the client have to be analyzed before the interaction on the requirements. This is crucial for the project as the vendor should not get into a technical discussion with someone who is non-technical. Technical design or Architecture should not be combined with the Requirements phase

- **Effort Overrun:**
  Effort overrun is again a result of all the above mentioned cases. This can be reduced if the deliverable as planned is as per the problem statement identified.

- **Delay of Requirements Phase: Effort Overrun, Schedule slippage & evolving requirements ultimately hampers the requirements phase. Mitigation strategy should be thought in order to put a check on this.**

- **Lack of domain knowledge on vendor side:**
  If the vendor has not understood the client’s domain then it can pose a major problem for the success of the project. It was observed that if the domain knowledge sessions are mixed with the requirements understanding session then it would become chaotic and the quality of the project would suffer.

- **Scope Creep:**
  It is important that the scope of the project is defined in the beginning. As the project moves ahead in the requirements phase it would be difficult if all the requirements are not aligned in one direction.

- **Misinterpretation of the use cases by the vendor:**
  It so happens that the use cases prepared made by the client is misinterpreted by the vendor.

- **Attrition/unavailability of resources due to a delayed requirements phase:**
  Due to the huge time gap between the requirements phase & the development phase there was risk of attrition of resources. This may badly hit the schedule of the project pertaining to the loss of domain knowledge.

- **Time Zone difference between the client & vendor:**

---

Figure – 1: Gold Plating concept

![Diagram of Gold Plating concept](image-url)
Due to the time zone difference the communication is hampered. Proper channel of communication & reporting should be there.

All the above challenges posed major threat to the success of the project. It was realized that there is a need for the client to prepare before starting the kickoff of the requirements phase with the vendor. This pre-Requirements phase would help the clients understand the criticalities involved in such a project and thus target the project management issues separately from the requirements; there would be more satisfaction from the client side.

4 Solution to overcome the challenges

As per the standard Software Development Life Cycle the Requirements phase directly starts by workshops & questionnaires with client followed by the requirements analysis & definition.

Before giving deadlines to the Vendor it is important to prepare ourselves and plan steps that need to be taken so as to make the team understand the concepts & the requirements module wise.

Steps to be taken before the start of the requirements phase can be depicted as below in the form of a phase/cycle.

4.1 Mapping Steps to overcome challenges

Mapping steps to overcome the challenges faced by clients:

After having performed the above steps the requirements workshops should be initiated. This would result in a smooth requirements phase and help in delivering quality product.

5 Démarcation of Project & Requirements Management

There is a very thin line in between Project Management and Requirements Management. Based on hierarchy of a project a Project Manager is the top boss responsible for the successful deliverable of the project to the client. However the Requirements Manager is the one who interacts with the client directly and has the onus of the requirements phase.

It is extremely important that a project manager and a requirements manager be in sync to avoid any discrepancies in the project. Though the ownership of the requirements is with the requirements manager but a project manager should also know what all has been committed to the client and if there are any gaps or not. Similarly a requirements manager should also be aware of the project risks and the dependencies.
This would act as a risk mitigation strategy wherein that one can replace the other whenever required. Requirements manager must ensure that the requirements are duly aligned with the projects vision.

![Project Implementation Hierarchy](image)

**Figure – 3: Hierarchies during Project Implementation**

6 Conclusion

It is important for the client to develop a clear vision that would define project need and a problem statement. This would result in aligning the requirements of the client in one direction which would help in a smooth transaction of the requirements phase from both the client and the vendor.

7 References


[3] Journal on “A field study of the requirements engineering practice in Australian software industry”


[5] Requirement Gathering and Management in Globally Dispersed Teams (By Anosh Wadia)

[6] Improving Requirements Engineering from the Client's Perspective in the Health Care Domain (By I. Minkkinen and A. Eerola (Finland)
An Approach for Incremental Certification of Software Components

Eyton A. Ferguson and Ezra K. Mugisa
Department of Mathematics and Computer Science, University of the West Indies, Kingston 7, Jamaica, W.I.

Abstract - One necessary condition for the success of Component Based Software Development (CBSD) is the assurance that a component will work as specified when it is integrated with other components to build an application. Certification of software components, though a reliable solution to this problem, adds costs that can make certified reusable components unaffordable especially since these costs are mostly incurred early in the market life of the component (typically when deposited in the repository). We propose to develop a framework within which certification of software components can be accomplished by emerging methods such as self-certification, and good-enough testing so as to facilitate certification of a component on a phased (incremental) basis. The aim is to distribute the cost of certification over the market life of the component, thereby making certification of components, more affordable for component builders, and component users.

Keywords: Certification of Reusable Software Components, Component Based Software Development.

1 Introduction

The issue of “Trusted Components” [1] is a challenge to the successful practice of Component Based Software Development. An application developer who has concerns regarding the quality of available reusable software components might opt to build the required component in-house instead of developing the application with reuse. This is clearly not a practice that encourages CBSD. Certification seeks to overcome this challenge by providing a warranty that a reusable software component will live up to the claims of the component builder. Consequently, certification of reusable software components is seen as critical to the practice of both, development for reuse, and development with reuse.

A definition for Certification of Reusable Software Components (CRC) as given in [9] states: “For CRC, certification refers to a process in which inspection, analysis, and testing techniques are used to achieve assurance of the quality of reusable components.” The term “quality” refers to characteristics of the component that can be observed as what the literature on software certification refers to as the properties of the component. We draw on a definition of a property given in [7] as “a true statement about some aspect of a reusable part. A property might be an assumption that a part makes about its operating environment or a specific quality that the part can have.”

Discussions of the different approaches that have been practiced over the years are presented in [5]. Meyer also discusses a classification of certification approaches in [1]. Generally, techniques for certification of reusable software components can be classified as process based or product based. A process based approach certifies the quality of a component by warranting the process by which the component is produced in much the same way that the quality of a watch that is built by a master clockmaker is warranted to be of high quality. Product based techniques primarily subject the finished product to a set of tests, inspections, and analyses each of which aims at evaluating a particular property of the component. Irrespective of which approach is used certification should result in a statement (preferably a written one) that is issued by the certifying party to say which of the properties that the component is claimed to possess have been tested and verified as being true.

It is not always clear who has the responsibility for certification. On one hand, advocates of third party software certification argue that verification by an independent authority is the only reliable means of ensuring that trust in a component is warranted. Jeffery Voas [2] in particular, proposes the use of Software Certification Laboratories which test software components and operate in conjunction with Software Certification Authorities (SCAs) that issue certificates based on test results. Advocates of other certification approaches, though not totally opposed to third-party certification, argue that the use of Software Certification Laboratories can make the cost of a component too high for small developers and that third-party certification is economically feasible only for safety critical systems where the cost of failure far exceeds the cost of guaranteeing system reliability.[4] These researchers argue for approaches such as Developer Self Certification [4] which propose distributing the task of certification among the various CBSD participants, and the use of model checking [10] for software verification and certification. What is evident from the debates is that there is no “one size fits all” approach to certification but rather that a variety of techniques are needed depending on which class of properties is being certified and how much certification is needed to assure the user of the component. It is also important to note that, regardless of who does it or how it is done, certification increases the value of a component and will affect the price at which the component is sold. It is therefore desirable to identify reliable approaches that can reduce the cost of certifying a reusable software component. We present a proposal for certification of the properties of a
reusable software component on an incremental basis, as needed for each instance of reuse, as a solution to this problem. Our proposal aims to spread the certification effort over the market life of a reusable software component thereby reducing the cost of certification that is brought to bear on the price at which a certified reusable software component is sold to a developer.

2 Scope of Analysis

We will consider that price is the major factor affecting a developer’s decision to buy a reusable software component instead of building a comparable component in-house. This is certainly true for an application developer who has the skill to build a component that is comparable in quality to a certified reusable component that is available from a repository. The price of a component to a developer who is building an application is determined by:

1. The cost of building the component, being the costs that are incurred by the component builder;
2. The total cost of certification that is required to warrant the component for use in the application;
3. How the component is priced in order to recover these costs.

We note that there are several pricing models now emerging for the unique considerations that must be made in pricing software. Several of these are discussed in [11] including value based approaches, per use models, and transaction based pricing. Generally the trends are toward revenue models that seek to create revenue streams that reduce up front charges and focus instead on the downstream revenues from maintenance. It is our opinion that while pricing models can be used to reduce the cost of a reusable certified component the extent to which this can be done is constrained by the level of costs that are incurred in building and certifying the component. The cost of building a component is determined by the size of the component, its complexity, and the efficiencies of the software processes by which the component is built. We will not consider the issues that relate these factors to certification costs. Instead, our discussion will focus on the certification of a reusable software component after it has been built and deposited in a repository – product based certification.

3 Related Work

Our research is motivated by the work of several authors on software component certification. The concepts of Developer Self Certification presented in [4], and Good-Enough Testing [3] are fundamental ideas on which we base our proposal for incremental certification. We also acknowledge the work of [6,9,11] which present models for representation of the quality properties of a component. Early work on usage based testing [9] has also influenced our proposal for the construction of Context Models from which the quality requirements of a component in a specific reuse context can be derived. Finally, there are several authors who propose frameworks for certification of reusable software components. Our contribution to research in this area complements their work with our focus being on the economics of certification.

4 The Cost of Certification

Our discussion of certification will be based on the existence of formally specified properties for a reusable software component using the property-value pair approach that is described in [6]. Each quality characteristic of a component is represented by a set of sub-characteristics each of which is broken down to yield an attribute that can be measured by inspection, testing, or analysis to obtain a “score” that indicates whether the component has the property or not. The score can be either a Boolean value, or a value within range of values. A certificate for a reusable software component can be regarded as a pair of ordered lists:

\[ P = [p_1, p_2, ..., p_n] \]

being the list of all quality properties that are measureable for a component, and

\[ V = [v_1, v_2, ..., v_n] \]

being the list of values corresponding to the properties so that each \((p_i, v_i)\) is the property-value pair that represents the level of certification attained for the quality property \(p_i\).

For each application in which the component is to be integrated there can be defined a set of property values \(V = \{v_0, v_1, ..., v_n\}\), such that each \((p_i, v_i)\) defines a required certification level \(v_i\) for property \(p_i\). If a particular property is not relevant in the reuse context then \(v_i\) can be set to null. A component is said to be certified for reuse in context \(i\) if, and only if, all properties that the component must have to be warranted for reuse in the context are certified at, above, the level that is required, i.e.

\[ \forall v_i \in V, v_i \neq \text{null} \rightarrow v_i \geq v_i, v_i \in V \quad (1) \]

The cost of certifying a component for reuse in context \(i\) is the number of properties that need to be certified, times some measure \(h\) that represents the average cost of certifying each property. In the worst case all the required properties must be certified regardless of whether they were previously certified in some other context and the cost of certification in

Note: 1 In practice, the effort that is required to certify a property can include the cost of rework in the cases where testing reveals that the property does not hold. We will make considerations for rework in future discussions on incremental certification. For the present analysis we will assume that another component is sought to replace one that fails certification.
a specific reuse context is determined for cardinality of the set of all required properties as:

$$|V'|, v \in V, v \neq null \times h$$ (2)

If we assume instead that certification results can be reused and define \(X'\) to be the list of those properties for which reuse of prior certification results is not possible then we can define the set of properties which must be tested for certification in context \(i\) (those not already certified at sufficient levels) as:

$$X = \{(p_i, v_i)\}, v_i \in V, p_i \in P, v_i \neq null, v_i < v$$ (3)

Under this assumption we can establish a simple formula for the cost of certification in context \(i\) as:

$$c_i = |X'| \times h$$ (4)

If \(d_i\) is a measure of the extent to which the reuse of prior certification is possible in reuse context \(i\), and if we represent initial certification cost for the component as \(c_0\) then the quantity \(c_i\) can be expressed as:

$$c_i = (|V| \times h) - (d_i \times c_0)$$ (5)

where \(d_i = (|V| - |X'|) / |V|\)

The ideal certification approach is one in which the reuse factor \(d_i\) is 1 for every instance in which the component is certified for reuse. This ideal is conceivable for certification of a component that has a small number of properties, or for a component that is built for very limited reuse. What prevails in practice is that each context in which a component is reused will have its own requirements for the properties that the component can have in that context. Also, a particular property might only be required in a specific context while being irrelevant in others. We therefore define a desirable approach to the certification of a reusable software component as an approach that includes an initial certification of the component followed by certification (if necessary) when the component is selected for reuse by a developer. This approach should seek to:

1. Minimize the cost of initial certification, \(c_0\), by seeking to initially certify only generic properties, or at least those properties that are deemed to be important in a large number of environments in which the component might be reused;

2. Certify as many properties as can be certified in each instance of certification for reuse in a specific context thereby maximizing the potential for subsequent reuse of certification results;

3. Certify each property only to the level that is required for the use of the component in the specific reuse context.

Our proposal for incremental certification of reusable software components is based on these considerations.

5 Incremental Certification

Figure 1 depicts what we refer to as incremental certification of reusable software components. The following is a description of the processes that are involved:

1. The Component Builder conducts testing of the component as part of the development process. The test cases and results are documented and packaged with the component that is shipped to the repository.

2. The Repository conducts initial certification of the generic properties of the component. The component builder’s test results are used in the initial certification in order to lessen the cost that is incurred at this stage. The documentation that accompanies the component is then complemented by specifications and results of all tests that have been used in initial certification of the component. This will allow a developer who wishes to select the component to review these tests in what is referred to as developer self-certification [4].

3. An application developer who seeks a reusable software component constructs a Context Model which describes the interactions that the component will experience when integrated into the application (reuse context). The context model details the intended usage of the component including probabilities for each set of interactions. Certification levels that the component can have in the reuse context are determined by Certification Requirements Analysis of the Context Model to give the list \(V'\) of the required certification level that each property must have in the reuse context.

4. The properties that will have to be certified for reuse in Context \(i\) are determined by Good-Enough Analysis. The analysis compares the required certification level for each property in the list \(V'\) against the certification level already attained for the property in the list \(V\). A property will need to be certified in the target environment if has not yet been certified at, or above, the level of certification that is required for reuse of the component in the target environment. Good-Enough Analysis generates the list \(X\) being the list of those properties that are already certified at an acceptable level, and the list \(X'\) being the list of the properties that will need to be.
certified before the component can be warranted for reuse in context $i$.

5. A decision on whether to proceed to certification can be made at this point based on the number of properties in $X$ and the amount of effort that will be required to test the component for these properties. If certification is chosen then the component is tested for the properties in $X$.

6. If the component fails certification (i.e. one or more properties fail testing) then the developer either seeks another component, or requests that the component be reworked to address the fault that resulted in failure.

6 Conclusions and Future Work

We expect that reduction in the amount of effort (cost) that is incurred in certifying a component for reuse will encourage greater use of certified software components by application developers. To this end we propose incremental certification of software components, based on the principles of good enough testing and reuse of certification results, as an approach that can be used to reduce the cost of certification of a component for its use in a specific reuse context. We intend to continue work in this area by developing a framework for incremental certification of software components. The framework shall specify several models for managing certification data. These include data models for representing the quality properties of a component, and data models for the representation of the usage profile of a reuse context. The framework shall also define techniques for extracting certification requirements from these data models. Finally, and perhaps most importantly, the framework for the incremental certification of reusable software components shall specify decision models that can be used to determine which properties of a component should be certified at the point of initial certification, which properties are to be certified in each reuse context, and how much certification is considered sufficient to warrant the use of the component in each context.
7 References


[7] Michael F. Dunn, John C. Knight, *Certification of Reusable Software Parts*, University of Virginia, 1992


Binary Methods and Parameterized Types in Java: A Conflicting point between Theory and Practice

Cong-Cong Xing
Department of Mathematics and Computer Science
Nicholls State University
Thibodaux, LA 70310
cmps-cx@nicholls.edu

Abstract

The issue of the relationship between inheritance and subtyping has been extensively studied in OOP theory in the past two decades. Unfortunately, it seems that what we found in theory has not been taken into consideration in the mainstream OOP language Java. As a result, strange and anti-principle programming behaviors can occur in Java when inheritance and subtyping are considered in the context of binary methods and late binding, with the presence of parameterized types. In this short paper, we (1) review the theoretical study results pertaining to the relationship between inheritance and subtyping, (2) investigate how this issue is handled in Java with the presence of parameterized types, (3) show the confliction between OOP theory (principle) and OOP practice (Java), and (4) discuss possible ways towards resolving this confliction.

Categories and Subject Descriptors: D.3.1 [Programming Languages]: Formal Definitions and Theory; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs; D.2.5 [Software Engineering]: Testing and Debugging

General Terms: Languages

Keywords: Object-oriented programming, Java, \( \varsigma \)-calculus, binary methods

1 Introduction

Typing is a central issue in object-oriented programming (OOP). Without concerns for typing, objects could be satisfactorily interpreted as the self-application of records (e.g. [7]). It is typing, particularly subtyping which is a reflexive and transitive binary relation over the set of types, that creates a great deal of difficulties and bewilderment in the process of understanding and interpreting the OOP paradigm. Using the notations in [2], we write \( a : A \) to signify that a term \( a \) has type \( A \); \( A \to B \) to signify a function type with argument type being \( A \) and result type being \( B \); \([l_1:A_1, \ldots, l_n:A_n]\) to signify an object type where \( l_1, \ldots, l_n \) are the methods labels of the object and \( l_i \) has type \( A_i \) for each \( i \); \( A <: B \) to signify that type \( A \) is a subtype of type \( B \). The intention of (having a) subtyping (relation) is reflected in the following rule of subsumption:

\[
\frac{a : A \quad A <: B}{a : B}
\]

That is, if the term \( a \) is of type \( A \) and \( A \) is a subtype of type \( B \), then \( a \) is of type \( B \). Therefore, by this rule, for any program context where a term of a certain type is required, we can always fill this context by a term which is actually of a subtype of the required type.

Subtyping for primitive types is simple and fixed, and serves as the basis for other complex subtyping situations. For example, we can stipulate that \( int <: real \). Function subtyping is covariant at the result position and contravariant at the argument position. Namely, given two function types \( A \to B \) and \( A' \to B' \), \( A' \to B' <: A \to B \) if and only if \( A <: A' \) and \( B' <: B \). Object subtyping is more complicated and is still an ongoing research topic. Existing object subtyping
scenarios are, among others, covariant, invariant, and variant-by-need (e.g. [3, 2, 8]). Suppose $l: A \to B$ is a method of an object of type $Self$, if $B = Self$, that is, $l$ returns the hosting object or a modified hosting object, then we say that $l$ is a covariant method; if $A = Self$, that is, $l$ takes as argument the hosting object or a modified hosting object, then we say that $l$ is a contravariant method; if $A = B = Self$, that is, $l$ takes and returns the hosting object or a modified hosting object, then we say that $l$ is a binary method.

A critical issue that is associated with object subtyping is inheritance. On the one hand, they are two different entities. Inheritance refers to a mechanism of code writing and reuse (new objects/classes can be written by utilizing existing components of objects/classes and adding fresh components) whereas object subtyping is a binary relation over the set of object types. On the other hand, we certainly wish that inheritance can always imply object subtyping. Specifically, if an object $b$ is built through inheritance from an object $a$, and $b: B$, $a : A$, then we would like to have $B <: A$ so that $b$ can be used in any program context where an object of type $A$ is expected. By doing this, new data structures constructed through inheritance will be “compatible” with old programs, and large software programs can be effectively built up through inheritance. In fact, this is what we have been doing in practice with major OOP languages such as Java and C++. Unfortunately, it has been shown, in theory, that inheritance does not induce subtyping in general (e.g. [6, 4]). The problem is that when contravariant (or binary) methods are present in objects, allowing inheritance-induced subtyping can make programs invoke a method that does not exist and thus cause a “method-not-understood” run-time error.

In this short paper, we first review the binary/contravariant method problem under the framework of Abadi-Cardelli’s $\varsigma$-calculus [2]. Then we examine how this problem is practically handled in Java with the presence of parameterized types. By doing this, we demonstrate the confusing treatment of binary methods in Java which leads to a confliction between theory and practice, and suggest possible ways towards resolving this confliction.

2 The Binary Method Problem

Points with additional attributes, such as color points and movable points, have been used as an interesting study-case in the in the context of binary problems in the fundamental research of OOP. In this section, we recast the essence of the binary method problem in the literature by using point objects and $\varsigma$-calculus notation.

The fundamental element in the binary method problem is the contravariant position of the argument of a method. So in the following description of the binary problem, to keep things simple, we will use a contravariant method instead of a binary method. Suppose we want to define a one-dimensional point which contains a field $x$ indicating the point’s current position and a method $eq$ which can take another one-dimensional point and compare it with the hosting object itself to see whether they are at the same position, the type $PT$ of the point and the point itself $pt$ can be coded, in $\varsigma$-calculus, as follows:

$$PT \overset{\text{def}}{=} \mu(T)[x : int \quad eq : T \to bool],$$

$$pt \overset{\text{def}}{=} \varsigma(s : PT)[x = 1 \quad eq = \lambda(p : PT)(p.x == s.x)],$$

where $PT$ is recursively specified, $\varsigma$ is the self binder in $\varsigma$-calculus, $s$ is the self variable, and the intentions of field $x$ and method $eq$ are obvious.
Now we would like to define a color point \( cpt \) through inheritance from \( pt \) by inheriting field \( x \), adding a new color field \( c \), and overriding the method \( eq \). Let \( a \leftarrow b \) denote an object constructed through inheritance, where \( a \) is the superobject and \( b \) contains the new components and overriding components, we can easily write \( cpt \) as follows:

\[
\begin{align*}
\text{cpt} & \overset{\text{def}}{=} \text{pt} \leftarrow \varsigma(s: \text{CPT}) \\
& \quad \left[ \\
& \quad \quad c = \text{blue} \\
& \quad \quad eq = \lambda(p: \text{CPT})(p.x = s.x \land p.c = s.c) \\
& \quad \right],
\end{align*}
\]

with its type

\[
\text{CPT} \overset{\text{def}}{=} \mu(T) \left[ \\
& \quad x : \text{int} \\
& \quad c : \text{color} \\
& \quad eq : T \rightarrow \text{bool} \\
\right].
\]

Given the fact that \( cpt \) is inherited from \( pt \), and \( \text{CPT} \) and \( \text{PT} \) are the types of \( cpt \) and \( pt \) respectively, the question we would like to ask here is: Is \( \text{CPT} <: \text{PT} \)? Alternatively, does inheritance imply subtyping in this case? (Note that the method \( eq \) in both \( pt \) and \( cpt \) is a contravariant method.) The answer to this question is no. To see this, we define a one-dimensional point \( p_0 \) and a function \( f \) as follows:

\[
\begin{align*}
\text{p}_0 & \overset{\text{def}}{=} \varsigma(s: \text{PT}) \left[ \\
& \quad x = 0 \\
& \quad eq = \lambda(p: \text{PT})(p.x = s.x) \\
\right],
\end{align*}
\]

\[
\begin{align*}
\text{f} & \overset{\text{def}}{=} \lambda(p: \text{PT})(p.eq(p_0)).
\end{align*}
\]

The functionality of \( f \) is clear: it takes a point of type \( \text{PT} \) and compares it with the point \( p_0 \). Note that due to subsumption, \( f \) can take points which are of a subtype of type \( \text{PT} \). Note also that due to late binding (also known as dynamic binding), which is one of the defining features of OOP (see e.g. \([2, 5]\)), the method \( eq \) in \( f \) which is executed when \( f \) is called, will be the \( eq \) method of the run-time argument to \( f \), not the \( eq \) method decided at compile-time.\(^1\) Compilers have no way of knowing statically which \( eq \) method will be executed. If \( \text{CPT} <: \text{PT} \), then \( \text{cpt}: \text{PT} \) due to subsumption and \( f(\text{cpt}) \) will type-check. However, note that

\[
\begin{align*}
f(\text{cpt}) &= \text{cpt}.eq(p_0) \\
& = (p_0.x = \text{cpt}.x \land p_0.c = \text{cpt}.c)
\end{align*}
\]

and that \( p_0 \) does not have a color field, so the execution \( p_0.c \) crashes. Therefore, allowing inheritance-induced subtyping is unsound when contravariant methods are present in computations. This is one of the fundamental difficulties found in OOP theory research and is generally referred to as “method-not-understood” error.

How is this problem handled in Java? In the next section, we recode this problem in Java with presence of parameterized types and examine the responses from Java.

### 3 Java Code of the Problem with Parameterized Types

We now examine how the binary problem is handled in Java together with the use of parameterized (also known as generic) types.

\(^1\)This looks a bit natural in the setting of \( \varsigma \)-calculus. But, as we will see shortly that situations are different in Java.
```java
// class PT
public class PT<T>{
    T x;

    public PT(T p) { x = p; }

    public boolean eq(PT<T> p){
        System.out.println("eq in PT is being used");
        return (this.x == p.x);
    }
}

// class CPT, inherits from PT
public class CPT<V> extends PT<Integer>{
    V x;

    // a "color" field
    double c;

    public CPT(V p, double q){
        super(1);
        x=p;
        c=q;
    }

    public boolean eq(CPT<V> p){
        System.out.println("eq in CPT is being used");
        return (this.x == p.x) && (this.c == p.c);
    }
}

// Class SeeIt
public class SeeIt{
    public static void test(CPT<Integer> p){
        CPT<Double> p0 = new CPT<Double>(1.3,2.3);
        System.out.println("p's x is: " +p.x);
        System.out.println("p0's x is: "+p0.x);
        System.out.println("the result of comparison is: "+p.eq(p0));
    }

    public static void main(String[] args){
        CPT<Integer> b = new CPT<Integer>(7,2.1);
        test(b);
    }
}
```

Figure 1: Java Code of the Binary Method Problem with Parameterized Types
Note that the binary problem described in the previous section can be “generalized” a bit in the following sense: There are three types which are of our interests: the type of the parameter $p$ of the function $f$, the type of the point $p_0$ used in function $f$, and the type of the argument to function $f$ when it is called. Considering that each of the three types could potentially be $PT$ or $CPT$, we have, in combination, a total of eight cases to consider. In order to effectively point out the problem that Java has in handling the binary problem, we will not go through each of the eight cases. Instead, we just single out the case where Java behaves strangely.

Figure 1 lists the classes $PT<T>$, $CPT<V>$, $SeeIt$, and some execution scripts. Class $PT<T>$ defines one-dimensional points. The field $x$ and method $eq$ play the same role as specified as in the $\zeta$-calculus case. $T$ is the type parameter (variable) which is used to specify the type of the field $x$. In method $eq$, an additional message is printed showing the location of this $eq$ method to help us to trace which $eq$ is being called. Class $CPT<V>$ is defined through inheritance from class $PT<Integer>$. A “color” field $c$ of type $double$ is added into $CPT<V>$, and the $eq$ method is redefined by comparing both the $x$ field and the $c$ field of the hosting object with that of the argument object. Note that the field $x$ is redefined to be of type $V$ in $CPT<V>$. Class $SeeIt$ is used to test some computations with classes $PT<T>$ and $CPT<V>$. Its local method $test$ is the counterpart of the function $f$ in the $\zeta$-calculus description of the problem. In method $test$, a point $p_0$ of type $CPT<Double>$ is created and is submitted as argument to the $eq$ method of the parameter $p$ of type $CPT<Integer>$. In the $main$ method, a colored-point $b$ of type $CPT<Integer>$ is created and is passed to the method $test$.

From the execution script, we can see that it is the $eq$ method in class $PT<T>$ that is being executed, and the result of comparison is $true$. This raises the following two questions:

- Why $true$? The parameter $p$ in $p.eq(p0)$ in method $test$ will be replaced by the point $b$ at run-time. So what gets executed is $b.eq(p0)$. Due to the inheritance mechanism of Java, $b$ has two $eq$ methods with the same name but different signatures: One is inherited from class $PT<Integer>$ which takes an argument of type $PT<Integer>$ and returns a boolean value; the other one is newly added into $b$ which takes an argument of type $CPT<Integer>$ and returns a boolean value. The execution script shows that the inherited $eq$ method in $b$ is used. By its definition in class $PT<T>$, this $eq$ method compares, at run-time, the $x$ field of the hosting object ($b.x$ in this case) with the $x$ field of the actual argument ($p0.x$ in this case). Considering that $b.x$ and $p0.x$ have different values (as shown in the execution script, $b.x$ has value 7 and $p0.x$ has value 1.3), it is really strange as to why Java produces a $true$ as the result of this comparison.

- Is $CPT<Double>$ a subtype of $PT<Integer>$? As stated above, $b$ has two $eq$ methods with the same name but different signatures. In the process of choosing which one to be applied to the argument $p0$ of type $CPT<Double>$, Java apparently selected the inherited $eq$ method which requires an argument of type $PT<Integer>$. This indicates that Java treats the type $CPT<Double>$ as a subtype of the type $PT<Integer>$, which is dubious. To see why $CPT<Double>$ may not be a subtype of $PT<Integer>$, note that the field $x$ is redefined to be of type $Double$ in $CPT<Double>$ while it is of type $Integer$ in $PT<Integer>$. As such, if $CPT<Double>$ is a subtype of $PT<Integer>$, then any object of $CPT<Double>$ can be regarded as of type $PT<Integer>$, and consequently can be used to substitute for any parameter of type $PT<Integer>$ in any program context. This, however, can cause potential typing and programming errors. For example, given the following fragment of code

```java
public void f(PT<Integer> p){
    ...suc(c.p.x)... 
}
```
where succ is the successor function defined on integers only, and p is a parameter of type PT<Integer>; if p is replaced by an object q of type CPT<Double>, then q.x will yield a real number and will cause succ(q.x) to crash. Therefore, regarding CPT<Double> as a subtype of PT<Integer> is unsound, and it is strange as to why Java does so.

In short, the point we would like to make here is: although CPT<Double> is inherited from PT<Integer>, CPT<Double> may not be a subtype of PT<Integer>.

The above observations show that the issue of the binary method is not satisfactorily resolved in Java when parameterized types are involved in programming, and point out a place where Java needs to be improved.

4 Final Remarks

Inheritance and late binding are two of the defining features of the OOP paradigm. We have reviewed, in theory, that a fundamental difficulty, known as “method-not-understood” error, can occur as the result of the combination of inheritance, subtyping, and late binding when binary methods are present in computations. The corresponding Java code of the problem with parameterized types is subsequently presented and studied to examine how this problem is handled in practice. As we have seen, Java’s behavior is strange and unsound.

Towards a rectification of this problem, we would like to make the following remarks:

• Separate the notion of classes from the notion of types (perhaps, as in Ocaml [1]) in Java. A class is a template from which an object can be created, and contains concrete codes (definitions) for methods, whereas a type is an abstract specification of code behaviors. In Java, classes are types, which is one of the sources that confuse subclassing (i.e. inheritance) with subtyping.

• After classes and types are separated, separate the concept of inheritance from the concept of subtyping. Inheritance refers to code construction technique by reusing existing codes, whereas subtyping refers to a binary relation over the set of types of codes, bearing in minds that types are abstract specifications of codes. In Java, inheritance automatically engages in subtyping under all circumstances, which contradicts what has been found in OOP theory and may inflict problems (e.g., CPT<Double> vs. PT<Integer> as discussed in the previous section).

• After inheritance and subtyping are separated, we need to find out under exactly what conditions that inheritance can (or cannot) induce subtyping, especially for the case of parameterized types. More theoretical work needs to be done to clarify this issue.

• Corrections and improvements in Java, under the guidance of OOP theory, need to be done continuously and appropriately.

Theory and practice of programming are two complementary activities. While theory may be built upon practice, practice can be advanced by the results from theory. It is indeed the case in Java programming.
References


Experiences on requirement management in offshore outsourcing

Yuqin Li1, David Helgesson2

1 Computer Science Department, Lund University, Box 118, SE-22100 Lund, Sweden
2 HBC AB, PO Box 83, Sösdala, SE28010, Sweden

Abstract - Offshore outsourcing development has certain characteristics which are different from inshore development. As the most important part of software engineering, requirements management in offshore outsourcing development is more difficult than that in inshore development. Due to geographic separation between outsourcer and vendor, both partners in offshore outsourcing projects face challenges. This paper addresses characteristics of offshore outsourcing development, and discusses challenges that requirement management in offshore outsourcing development faces. Based on these challenges, some practical strategies are provided on how to deal with them. From our experience of using these strategies in real outsourcing projects, they are practical and immediately usable in offshore outsourcing projects.

Keywords: offshore outsourcing, requirement management, challenges, strategies

1 Introduction

Requirement management is an important part of software engineering. Many software projects have failed because of faulty requirements. Vague, incomplete requirements and unmanaged requirement change are the main causes [6]. Requirements are very important in inshore software development and are even more difficult in offshore outsourcing development. A key problem in outsourcing development is the evolution of requirements: no matter how thorough the requirement specification has been set up, the requirements for any non-trivial system will change, not only after the system has been built, but also during the process of implementing the system. This evolution of requirements is due to many reasons, including changing business needs or market and technology development. In addition to that, the process of designing, implementing, and writing test cases for requirements will increase the insight in the problem domain, which may very well lead to modifications on the initial set of requirements.[1]

Much research has been conducted in requirements management in order to make software development successful. It has mainly focused on inshore development. One company may develop the system using its own employees, or just contract the project to another company in the same country. These requirements management methods and tools are useful for both situations. Because the labor costs in different part of the world differ considerably, offshore outsourcing is getting more and more popular. For example, many Japanese companies outsource their projects to China, Japan, USA and Europe are main outsourcers, and at the same time, China, India, Russia and so on, are main vendors. In this paper, we define the outsourcer as the one who gives out a contract, and the vendor as the one who gets the contract and provides the service. In order to help outsourcing projects succeed, requirements management experts have started to conduct studies in this area. Requirements management tools for inshore development need to be changed to satisfy the new situation. Because outsourcing projects have their own characteristics, requirement management tools need to adapt to these characteristics as well.

Some research [13] was conducted in the outsourcing domain, most of which report experiences or suggestions. Most deal with general software management problems. A few studies focused on requirements management challenges. Among these, most focus on factors influencing the outsourcing development. More studies need to be conducted to adapt existing requirement management tools to outsourcing projects. This paper focuses on challenges of requirements management for offshore outsourcing development, and strategies to help outsourcing projects to reach their original objectives by managing requirements in an efficient way. A discussion about these strategies used in two projects was stated.

The paper is organized as follows: Section 2 discusses related work on requirements management in offshore outsourcing projects. Section 3 presents offshore outsourcing development, requirement management and its challenges, also provides strategies to handle the challenges. In section 4 a discussion is presented based on two projects conducted. Section 5 draws a conclusion and gives some suggestions for future work.

2 Related work

Requirement management is an important research area in software engineering. Most of the research done in outsourcing development was reports based on interviews, and the following are some of them:

Marco Lormans, Hylke van Dijk, and Arie van Deursen[1] discussed several difficulties managing evolving requirements by means of an industrial case study conducted at LogicaCMG. They reported on setting up a requirement management system in an outsourcing context and its application in real-life. They proposed a conceptual
framework of a system tailored for outsourcing environments, which captures the experience results. From the experienced outcome, several lessons were taken on how to manage evolving requirements, and solution directions.

Jyoti M. Bhat and Mayank Gupta, et al [2] focused on challenges arising from a client-vendor offshore-outsourcing relationship. Using case studies from an Indian IT-services firm, they addressed causes of RE phase conflicts in client-vendor offshore-outsourcing relationships. After reviewing the case studies, key strategic factors that are essential to RE’s success in a client-vendor relationship were revealed.

Alan Padula [3] examined attributes of the RE approach for two HP projects. The RE approaches were different and based on the differing project’s business drivers and project attributes. Countless variations in projects discourage widespread standardization and adoption of any single, rigid RE process. Rather there is: a fundamental meta-process that incorporates close teamwork; standardized requirement elicitation and documentation locally among individual project teams; an efficient, fast-turn-around change management process. On this basis, the RE process is tailored to best fit the business needs and project attributes.

Daniela Damian[4] pointed out that in order to overcome the significant cultural, time zone, and organizational challenges in global RE, stakeholders need effective knowledge acquisition and sharing as well as relationship building practices. Challenges in stakeholders’ global RE interaction are knowledge-acquisition and knowledge sharing, aligning RE processes and tools, and effective communication and coordination. Despite these challenges, it was found that practitioner reports of successful GSE practices give some hope to managing stakeholders’ distributed interaction. Those practices relevant to stakeholders’ RE interaction relate to supporting inter-organizational structures and supporting communication structures. These enable interactive ways of communication and coordination during the project life cycle, for the purpose of successful relationship building and expectation management. Furthermore, the “Collaborative Tools” sidebar outlines some resources that GSE projects have found useful.

Brian Berenbach, Mark Gall[5] proposed an unified approach that allows the integration of features, use cases, functional and non-functional requirements. By adding new symbols and relationships to UML it is possible to create a unified business model that allows inherent tracing of requirements and hazards.

Felix Rodriguez[13] compared several market-leading tools that aim at assisting software development within an offshore scenario. It [13] shows that there is no definite solution for selecting a tool for offshore development. The platform selection process always involves additional tradeoffs along with the technology and functional capabilities. Considerations such as platforms costs, estimated learning curve or project team expertise would definitely have to be taken into account. Tool selection depends on whether the emphasis is on one particular functional area or a balance across all areas desired. Tool strengths reside in one particular functional area and two at the most. As a result, any outsourcing project will require integration capabilities of the chosen tools to compensate for any flaws with a particular tool over a functional area.

Based on research experience [7,8], this paper focuses on addressing strategies to deal with requirement management in outsourcing and analysis results applied in two projects.

3 Requirement management for offshore outsourcing

3.1 Offshore outsourcing development and its challenges

Software development has changed a lot in recent years. The size of software is getting bigger and bigger, and number of stakeholders steadily increases. Based on different labor costs in different part of the world and companies’ strategy, offshore outsourcing development is getting more and more popular. From [9], [10] and [11], the outsourcing situation in China and Japan is revealed. Because China and Japan are neighbors and have similar culture, communication between China and Japan is much easier than with countries in other parts of the earth. China has many software engineers, and more and more software engineers enter the market from universities each year, thus China is a great source of software personnel. Meanwhile, Japan lack software engineers, so China is a natural outsourcing target of Japanese companies. China is also an offshore outsourcing target of both European and American companies. Because of English language skills of Indian people, India has been a big outsourcing market for European and American companies. As a consequence, it is obvious that outsourcing is a common trend in software development area. Offshore outsourcing is very significant since outsourcers can save labor costs by outsourcing projects to low-cost labor countries. For example, Japanese companies outsource to Chinese companies, and American companies outsource to Indian companies, etc.

On the other hand, some leading companies in computer area have built or plan to build their development center or support center in low-cost labor areas. For example, Microsoft built its engineering institute in Shanghai. IBM moved its support center to China not too long ago. The biggest motivation is likely readily access to low-cost labor. These kinds of companies have more offshore development than ever, even within the same company. The advantage of these companies to those outsourcing to other companies is that they usually employ
same tools and platforms even in different parts of the world. Some research [12,13,14] studied the key challenges which offshore outsourcing development partners need to deal with. They are: different platforms and tools, decreased project visibility, client business security, document maintenance and synchronization, factors leading to communication gaps, such as different time zone, different culture, different language, and so forth. Because the outsourcer and the vendor are in different parts of the world, and usually in different countries with a different time zone, communication between partners is limited. It is easier for those countries without big time difference to communicate, such as Japan and China, where there is only one hour time difference, and so members from different countries can communicate during work time. For countries with big time difference, such as USA and China, it is hard to communicate synchronously. If members from different geographic areas need synchronized communication, one part must work outside office hours, even during mid night, which is not convenient, and which will influence outcome of work in the following day, because if someone has worked overtime last night, one can not expect him/her to work efficiently the next day. This will also influence the group which he/she works with. Because of different time zone, most of the communication between different geographic areas is asynchronous. Asynchronous communication, such as email which is commonly used, can not provide direct feedback, so misunderstanding and questions take longer time to settle than in inshore projects. This means that the communication-response process is delayed by different time zone. Coordination costs grow to counteract the outsourcer’s original goal of decreasing cost.

Different culture and different languages easily lead to communication gaps. In offshore outsourcing development, misunderstandings are not guaranteed to be settled by asynchronous channels, and may be carried over into later phases. Offshore peers might be much more accustomed to work in a multi-culture and multi-language environment, but it still takes them time to follow up in the beginning. So if the outsourcer and the vendor are partners past the “getting familiar” period already, it is beneficial for them to keep a good relationship and continue work together, as outcome from previous cooperation can benefit later projects also. Usually at least one part has to use a language which is not his/her mother tongue to communicate with the other part. This constitutes a big challenge as well. If spoken language is not as good as written language, they had better use more written language when communicating, such as email, or a chat tool. The language skills of all members in the same part are not the same, so it is also wise that a team with good language skills is chosen to be representatives to communicate with the peer part. This can decrease misunderstandings, but it is not always easy to have a team as this, because people in such a team need not only have language skills, but also need to have good understanding about requirements and the phases following it.

Usually the outsourcer and the vendor are different companies, so it is common that they use different platforms and tools. For example, the outsourcer’s requirement management tool may be different from the vendor’s requirement management tool. Each side wants to keep its old tools and platforms when cooperating on a project, because it is easier for its own team members to use familiar tools in new project. If one side needs to adopt a new platform, it takes time for its members to get trained and accustomed to the new platform. This obviously constitutes a risk for the party which has to adjust. If important platforms and tools are not standardized, it may be hard and time consuming to fix even small problems. Sometimes one side has to sacrifice itself for the project.

Decreased project visibility is another challenge in offshore outsourcing development. Because of geographic separation, the outsourcer and the vendor can’t share project artifacts visually as easy as in inshore development. Many misunderstanding are removed easily by sharing a project visually when doing inshore development, but it is very difficult to have this done in offshore outsourcing development. When the outsourcer plans to contract out a project, this needs to be considered.

Some outsourced projects have some special business rules, and clients (outsourcers) don’t want to reveal all business information to vendors. If the vendor can’t understand thoroughly the client’s business rules, even some implied rules, then it leads to the vendor later providing the client with something the latter does not want. The client is normally very cautious in giving out test data as well. For example, financial companies have a responsibility to keep customers’ data confidential. But if there is no real test data, some problems can’t be found during the test period. There are even some clients that ask vendors and all participating employees to sign a confidentiality paper, but this still mainly relies on the individual’s ethical quality.

When two parts of the same project lies in different places, or even belonging to different companies, document maintenance and synchronization is yet another challenge. For example, the outsourcer is responsible for requirement specification, and the vendor is responsible for implementing requirements in artifacts. Along with requirement evolvement, the outsourcer might change some requirements when finding problems, but the change is not transferred to the vendor side. Because requirement evolvement is common and happens often, document maintenance and synchronization is consequently also very important.
3.2 Requirement management in offshore outsourcing

The purpose of requirement management is to manage the requirements of a project and to identify inconsistencies between those requirements and the project's plans and work products. Requirement management practices include change management and traceability. [12]

Because the amount of requirements is growing and change happens often, requirement management systems have been developed and used in many projects. Requirement management is more challenging in offshore outsourcing development than inshore development. Main challenges of requirement management lie in communication with stakeholders, comprehension of requirements and keeping consistency and traceability of requirements.

A simple way of outsourcing is what people used most at the beginning. The outsourcer is responsible for requirement specification, while the vendor is responsible for fulfilling requirements. Requirements are rarely changed during development cycle. When the vendor finishes the development, the vendor hands over artifacts of the system to the outsourcer. The outsourcer does an acceptable test of the product, and after the vendor respond to test results and fix bugs, the contract is finished and the vendor gets paid. Not much work is related to change management, because the requirements are stable during development. This is an idealized model of an outsourcing relationship, but in reality, the interface between the outsourcer and the vendor is not that simple and clear. The requirements usually are not stable after the vendor starts to develop the system. Without communication and participation in the requirement process, the members of the vendor can’t understand the requirements very well and thus can’t guarantee to fulfill the requirements in a correct manner. When they reach test phase, the ideal situation is that the software has no major flaws, otherwise it leads to redoing parts of the project, or even the project’s failure.

This kind of ideal situation has many advantages. A lot of communication is not needed between the outsourcer and the vendor, the outsourcer doesn’t need to explain a lot of business rules, and both sides are not influenced much by different time zone, culture etc. In current outsourcing market, outsourcing projects are not as simple as this. We thus need to identify challenges of common offshore outsourcing development, and find ways to deal with these challenges, in order to make outsourcing development successful and really reach the outsourcer’s main objective -lower cost.

Challenges of requirements management in offshore outsourcing consist of the following.

1. Different goals between the outsourcer and the vendor.

The outsourcer wants to simplify the management and decrease labor cost of development, and so it contracts the project to the vendor. Usually the outsourcer prepares the requirement specification and encloses it as a part of the contract, presuming that the rest of the work is the vendor’s. After the vendor gets the contract and requirement specification, it starts to develop the system. How to understand the requirements is a very important issue. Actually the vendor needs to know the priorities of the requirements and the outsourcer’s goal, otherwise how can the vendor satisfy the outsourcer? For many projects, the requirements are not fixed when the contract is signed. Users could for example come from different departments of the outsourcer. Requirements from different departments need to be integrated into one system, so negotiation between similar requirements from different departments should be made. Members from the vendor side can hardly know which department’s business is more important, and they can’t be required to ask any department to yield or sacrifice their requirements. Every user thinks his requirements are more important, and wants his own requirements to be fulfilled. In many projects, the IT department of the outsourcer has the role of communicating with business departments and gets final requirements and forwards those requirements to the vendor, as well as collects responses from the vendor and negotiates with inside users. The IT department is also responsible for having users from different business departments sign the requirement specification when the requirements are confirmed. It is common that requirement confirmation is delayed after the documents have been transferred from one department to another, and no one is willing to sign, because they are afraid to take responsibility. Usually the IT department is not powerful enough to negotiate with all these different parties of the project. Members in an IT department can’t know all the requirements and can’t discuss with the vendor side in behalf of the different business departments. When the vendor has questions regarding some requirements, they need to discuss with the business department in question directly in order to diminish misunderstandings. So the vendor has to deal with two groups: the IT department and business departments. Likewise, the outsourcer usually has to deal with two groups: a remote development team and a local team of the vendor. The local team of the vendor responsible for keeping good relationship with the outsourcer is mostly a sales section. It knows product information and outline merits, but actually can neither accept nor deny requirements in behalf of the development team.

2. Tools used by the outsourcer and the vendor are not standardized.

Usually the outsourcer and the vendor are different companies. It often happens that they are good at and using different platforms and tools. When the outsourcer chooses the candidate vendor, this is a fact that should be
considered, but it is still common that the outsourcer can’t always find a partner who is using same platforms and tools and also has good experience concerning the relevant kind of project. If the outsourcer and the vendor use different requirement management tools, it is still possible for them to share and comprehend requirement change later during development, but they have an increased risk of having more misunderstandings than if using the same tools. How to do? Some useful guidelines on how to deal with this challenge will be discussed in next chapter.

3. Communication channels and response time are limited.

If the outsourcer and the vendor are in different countries, they usually have different time zones. Different culture and language also increase difficulties of communication. Teleconference and instant chat programs are the most commonly used synchronous communicated methods. Email is the most commonly used asynchronous communicated method. Synchronous channels can provide direct response, but because of different time zones, it is really difficult to arrange teleconferences for all things to be discussed. At least one side has to work over time and outside office hours in order to use this kind of synchronous channels. Only in emergency situations will these methods be used. In most cases, asynchronous channels are used, such as email. No direct response is guaranteed, so it may take several days until a change goes through all related stakeholders. This risk of delay will especially be the case when the vendor is working under pressure of its own schedule, and so response to change will be even more delayed.

4. Bi-directional change management is difficult to maintain and synchronize.

Requirements are supposed to be fixed when the vendor starts to design and develop the system. This is only the ideal situation however. As the outsourcer digs down to business rules, some requirements need to be refined, changed, or even withdrawn. Requirements engineers of the outsourcer need to forward requirement change to the vendor side, and keep all requirements consistent and traceable.

On the other hand, when members of the vendor design and develop the system, they may find some requirements that conflict with others, and then need to make some changes. It can also be that when they design non functional requirements, some functional requirements are influenced, so that some requirements need to be changed during development. If the vendor just develops the system according to the changed requirements, without noticing the outsourcer, they will have problem to negotiate an agreement when testing the system. Whatever the vendor changes, the outsourcer should always be noticed. Documents on both sides also need to be synchronized.

All these aspects are related to building a mutually good relationship between the outsourcer and vendor. Actually offshore outsourcing development is an activity of building and maintaining a relationship between the outsourcer and the vendor.

3.3 Strategies on how to deal with challenges from requirement management in offshore outsourcing

Based on the challenges we discussed in previous chapter, some strategies will be helpful when dealing with these challenges. In the next chapter, we will show how these are used in our two projects and the results.

1. Set a shared goal between the outsourcer and the vendor.

If the outsourcer and the vendor have different goals, both sides need to sit down and communicate thoroughly. They need to understand that a common goal is vital for both of them. For instance, the outsourcer delays the process of requirement elicitation and analysis, and maybe thinks that the vendor can start working on the first part of requirements already, but the vendor however wants to have all requirements confirmed before starting to develop the system. In this case, the outsourcer needs to know that delayed requirement confirmation not only influences the vendor’s development schedule and cost, but also influences quality of the final system. Uncompleted requirements may lead to inconsistency of requirements in the whole system.

What if different departments inside the outsourcer organization have different goals? It is really difficult for the vendor to settle this problem. Actually one team with the outsourcer has to take this responsibility. Usually members in the IT department don’t have authority to settle this matter, but they should be given authority to ask other business departments to cooperate. A representative of all these departments can be given this role, such as a vice president of business.

The common goal should be set when the project starts, and both sides need to meet together to clarify each parts responsibility. After they understand the importance of sharing a common goal, the cooperation between the outsourcer and the vendor, as well as between different business departments will be much easier and smoother.

2. Standardize platforms and tools

When the outsourcer and the vendor use different platforms and tools, the risk of misunderstanding during requirement change management increases. One way to deal with this challenge is that one of the parties adopts the tools of the other party in order to have common standard tools. For example, the vendor is asked to adopt the new platform or tool which the outsourcer is using. It would be preferable that the outsourcer can provide some training so as the vendor project members can have confidence in using the new tool, and still keep the expected schedule. Usually the vendor does not want to adopt new platforms
and tools, because it’s obvious that more time and cost is needed for new tools. Sometimes the outsourcer needs to make sacrifices also in order to decrease risk. For example, the outsourcer needs to cover part of the cost for the vendor in order to adopt new tools or platforms.

This is not absolute however; the author had an experience when the outsourcer was willing to adopt new tools which the vendor was using. At the beginning, the outsourcer refused to change, and asked the vendor to change tools. After engineers from two both sides gathered together and the vendor side demonstrated the tool they were using, the outsourcer was convinced to change, after finding that this tool was better than their own one.

3. Choose right communication channels and set response time limitations

Asynchronous communication channels can’t provide direct response, while synchronous channels are more expensive and difficult to arrange due to different time zones. There is one way that can be used to solve this problem and increase efficiency of communication. One representative team on each side is selected to communicate with the other side. Both sides then knows whom to contact, and can get response from the right peer. The team will collect all things that need to be communicated and discussed before transmitting the message to the other side. After they send questions to the other side, they are supposed to get response within a certain time frame. At the beginning of the project, emergency levels are set up, and a corresponding response time is set for each level. When one side gets a question, that team will forward and discuss with related members and get a result, then respond to the other side within a certain time period. This makes it possible for them to predict how long they are supposed to wait until response. This can then be used to adjust their planning and scheduling.

4. Bi-directional change management being maintained

How to synchronize documents between the vendor and the outsourcer, so that both parties have identical documents? When the outsourcer changes some requirements, the vendor needs to know as soon as possible, otherwise this will influence the development of the system negatively, or even lead to conflicting requirements. If the vendor finds some problems during design and development, they make some change to the requirements. For the same reason, the outsourcer should also be noticed. The two sides need to make agreements about change, and how to synchronize documents and other artifacts of the system, otherwise when the system is developed, requirements on both sides will not be consistent. Maybe the outsourcer even doesn’t agree to accept the system.

If both sides use the same requirement management platform, it is easy for them to get changes from the other side, and easy to import the data into their own tool. Even better, if both sides can use the same tool online, in an internet version, in a certain period the change made by one side can automatically be exported to the other side, so that the other side can detect the change as soon as possible. After they examine the change, they will respond whether they accept the change or not. If not, they may continue to discuss, until an agreement is decided upon.

4 Discussion based on case studies

We studied these strategies and used in our two outsourcing projects. One project was cooperated with Japanese Company, the other one was with a company from Canada. When these projects were conducted, the main part of the team were in China, so we had less time difference with Japanese side, but more with Canadian side. In both projects, we were the vendor, and provided service for our customers.

First we chose a common platform to manage requirements. When the projects started, we had an agreed requirement document already, but as time went on, some changes were added in. Using the same tool to manage requirements and other artifacts was very helpful, and saved our time.

Different communication methods were used. With Japanese side, we used more email and phone call, even met together. The one who was responsible for communicating with us came to meet us at beginning and when we reached important milestones. We sent email about every thing needed to discuss. Two sides discussed over phone usually every week, or when something was needed to be discussed. But with Canadian side, we mostly communicated by email and sometimes by msn or skype conversation, both text and voice. From these two different projects, we can see that communication between China and Japan was much easier than that between China and Canada.

Also we used bi-directional change management method. Because we used the same tool to manage requirements, it was not difficult to synchronize between two parts. We usually did it once every week if no special needed, otherwise we did it when one side requested. For example, when our customer wanted to change requirements, they usually told us and we reacted fast to check if we could accept the change.

For setting the same goal between the outsourcer and the vendor, we discussed when we signed the contracts to cooperate, our goal was to fulfill the contract and tried to build a relation for more cooperation. The team in China was a consulting company providing outsourcing service. We put customers goal as our main goal, provided that we had enough resource. During the period we had no other resourcers, this was easy.

Actually offshore outsourcing development is very much related to relationship building and maintenance. If both sides are motivated to maintain a good relationship
with the other side and even prepared to initiate a step forward when needed, most of the problems coming from offshore outsourcing can be eliminated, such as cooperating with right communication channels, choosing standard platforms and tools, setting up mutual goals, etc. This leads to success for the whole team.

5 Conclusion and future work

This paper addresses characteristics of offshore outsourcing development, and discusses challenges that offshore outsourcing development face, which are different goals between the outsourcer and the vendor; non-standardized tools used by the outsourcer and the vendor; inefficient communication channels and response time; bi-directional change management is difficult to maintain and synchronize. Based on these challenges, some practical strategies are provided to deal with them. These strategies take offshore outsourcing’s characteristics into consideration, so the strategies are very practical. These strategies were practised in two projects and the result was good.

In future we will do more research on requirements management improvement in offshore outsourcing product lines development, and how to strengthen requirement management tools of inshore development to be used in offshore outsourcing development.

6 References


Experience of a Ruby-Syntax Representation for Structured Data

Kazuaki Maeda
Department of Business Administration and Information Science, Chubu University
1200 Matsumoto, Kasugai, Aichi 487-8501, Japan
E-Mail: kaz@acm.org

Abstract

This paper describes Ribbon (Ruby Instructions Becoming Basic Object Notation), a new representation written in a text-based data format using Ruby syntax. The design principles of Ribbon are high readability and simplicity of structured data representation. An important feature of Ribbon is an executable representation. Once Ribbon-related definitions are loaded into a Ruby interpreter, the representation can be executed corresponding to the definitions. Java programs are expected to read/write Java objects from/to persistent storage-media, or to traverse the structured data. A program generator was developed to create Ruby and Java programs from Ribbon definitions. In the author’s experience, productivity was improved in the design and implementation of programs that manipulate structured data.

Keywords: Data Representation, Structured Data, Domain Specific Languages, Ruby, Java

1. Introduction

A variety of representations for structured data have been developed to date. Currently, most of the representations are based on XML. If source code is analyzed and the result is written in XML to represent the syntactic information, the development of software tools is possible using various programming languages and libraries to manipulate the XML representation. JavaML[6] and XMLizer[7] are typical XML-based source code representations for abstract syntax trees (ASTs). XSDML[8] and srcML[9] support the representation of formatting information including white spaces and comments in addition to representing ASTs.

XML has many advantages. It can be used across different platforms, i.e., different computers, operating systems, and programming languages. Another advantage is a variety of libraries for many programming languages to manipulate XML documents.

In contrast, there are some disadvantages of XML. XML documents are written in human readable text format, but they are composed of many redundant tags, those are start tags and end tags, so that it is difficult to read and understand them. There are many standard specifications related about XML. But it is hard to learn most of them because there are a large amount of the specifications. Namespaces in XML are powerful to combine XML documents. It is, however, difficult to read XML documents containing the namespaces. The DOM-based approach is useful for programmers in manipulating the XML representation, according to the hierarchical tree structure. Programs using DOM traverse the structure; however, conversion from the document-oriented DOM to application-specific and problem-oriented data is necessary. If there are many kinds of objects to be converted, the work is tedious and error prone.

This paper describes Ribbon¹, a new representation of structured data in a text-based data format using Ruby syntax. The author believes that good readability and simplicity of structured data representation are important for all developers. Ruby is a simple but pow-

¹Ribbon stands for Ruby Instructions Becoming Basic Object Notation.
erful programming language, and its syntax is suitable for representation; therefore, Ribbon takes on advantage of using the Ruby syntax.

Ribbon and its related tools support all object-oriented programming languages (OOPLs). Although Java was used in our study, Ribbon does not impose restrictions on any of OOPLs. C# or Visual Basic could also have been used.

An important feature of Ribbon is that the representation can be interpreted and executed. For XML application development, an XML parser is needed. In the case of Ribbon, we do not need the parser for the representation because it is embedded in the Ruby interpreter. We therefore do not need to parse the representation in Ribbon. If definitions according to Ribbon are loaded into the Ruby interpreter, the representation can be executed corresponding to their definitions. This is useful for Java programs in reconstructing Java objects or for traversing the structured data.

Section 2 explains Ribbon and its related tools, Section 3 explains about current implementations, and Section 4 summarizes this paper.

2. Representation and Manipulation of Structured Data

2.1 DSL and Ruby

Domain specific languages (DSLs) are programming languages tailored to specific application domains[10] and designed precisely to describe problems in specific application fields[11]. They are special-purpose, and not general-purpose, programming languages.

Ruby is one of many general-purpose, object-oriented dynamic programming languages, combining scripting syntax with Smalltalk-like object-oriented features. The original implementation is written in C, but there are a variety of implementations of Ruby interpreters including Rubinius[14], JRuby[15], and IronRuby[16].

Ruby plays an important role as a host language for the DSLs. One of Ruby’s appealing features is the fact that parentheses for arguments of methods are optional; therefore, statements are easier to read and understand than ones in other programming languages. Another feature in Ruby is “block,” a group of program statements; it can be an argument of a method in Ruby. The block is powerful in its representation of structured data. The Ruby’s features give advantages to Ribbon in representing structured data. The next section explains the details of Ribbon.

2.2 Ribbon as an Object Notation

The representation of structured data in Ribbon is composed of several elements. Each element has a name and a value. For example, Figure 1 shows the name of an element as var and its value as “x.” The element var is not just a data representation, but internally, it is also an executable method invocation without parentheses in Ruby. The method name is var and the argument of the method is “x.”

```
var "x"
```

Figure 1. An element with the name var and the value as "x"

In Ruby, an element can have child elements using blocks. For example, Figure 2 shows that the var element has two child elements: line and column. The line element’s value is “3” and the column element’s value is “1.” Basically, one line can only have one element, according to Ruby syntax. If we write a semicolon at the end of an element, we can use multiple elements in one line. This is not recommended, however, since the author believes that simplicity is very important in representing structured data in Ribbon.

```
var "x" do
  line "3"
  column "1"
end
```

Figure 2. An element with two child elements

Ribbon supports four primitive data types: int, real, bool, and string. For example, Figure 3 shows that the bill element’s value is “by yen today,” and it has four child elements: price, tax, togo, and date. The price element has an integer value 340, the tax element has a real value 0.05, the togo element has a boolean value true, and the date element has a string value “April 1, 2008.”

An element in Ribbon basically does not have more than one child element with the same name. Recall
that a Java class has no more than one field with the same name. An element in Ribbon has a construct similar to a Java class. In comparison, an element to represent a collection can have more than one element with the same name. In Figure 3, the variables element has three child elements with the name var. The first var element’s value is “x,” the second var element’s value is “y,” and that of the last is “z.” This represents a sequence of elements.

If we need to represent an element linked to another element across the structure, a unique identifier is given to the element, and another element refers to the element using the identifier. Figure 4 shows that a uuid element with the identifier dbe0c2e6_5a63_4159_a76f_6e44 is given to the bill element with “by yen today.” The identifier in the figure is calculated using java.util.UUID, which is represented using a symbol in Ruby; another bill element in the var element has a reference to the unique identifier. This means that Ribbon supports graph-structured data.

2.3 Definition of Structured Data

Structured data is defined using symbols in Ruby and the four keywords has, is_a, is_seq_of and is_type, as shown in Table 1.

Figure 5 shows the definitions of the representations in Figure 3 and Figure 4. The meanings of the definitions are as follows:

- paymentAccount element has two child elements: bills and variables
- bills element is a collection of bill elements

2.4 Program Generation for Ribbon

A program generator (called “ribgen”) was implemented to generate Java programs from the Ribbon
:paymentAccount.has :bills,:variables
:bills.is_seq_of :bill
:bill.has :price,:tax,:togo,:date
:price.is_type :int
:tax.is_type :real
:togo.is_type :bool
:date.is_type :string
:variables.is_seq_of :var
:var.has :bill
:cash.is_a :bill
:credit.is_a :bill
:credit.has :cardtype,:cardno

Figure 5. An example of Ribbon definitions

java_package 'las_vegas'
java_prefix 'Serp'
generate_java
generate_ribsetup 'serp2init.rb'

Figure 6. Specification to generate Ruby and Java programs

definitions. In my experience of defining Java classes for ASTs, there are many cases in which a type name is made using the field name. For example, a type name “SerpCardno” is made using the field name “cardno,” so that the field declaration in Java is as follows:

    SerpCardno cardno;

Once the cardno element is defined in Ribbon, the above field declaration and the getter/setter method for the field are generated. Java classes generated by ribgen basically complies with this idea.

Ribgen generates Java programs and a class diagram for the generated Java classes. Figure 6 shows the specification as follows:

java_package specifies the path “las_vegas” for the Java package
java_prefix specifies addition of a prefix “Serp” at the beginning of the Java class name
generate_java specifies generation of Java classes
generate_ribsetup specifies generation of Ribbon-related programs to the file name “serp2init.rb”

Ribbon is designed to map the representation to Java objects. The program generator ribgen reads the definitions shown in Figure 5 and Figure 6, and generates thirteen Java classes as shown in Figure 7:

    SerpPaymentAccount.java
    SerpBills.java
    SerpBill.java
    SerpPrice.java
    SerpTax.java
    SerpTogo.java
    SerpDate.java
    SerpVariables.java
    SerpVar.java
    SerpCash.java
    SerpCredit.java
    SerpCardno.java
    SerpCardtype.java

Figure 7. Java classes generated by ribgen

The RbnApi class provides some basic methods for developers to build Java programs using the Ribbon representation. Typical methods are:

RbnApi() is a constructor that initializes Ribbon APIs
RbnBase.readRbnFile(String fileName) reads a Ribbon representation from the specified file and executes it on the Ruby interpreter.
void writeRbnFile(RbnBase obj, String fileName) writes the specified object (and objects reachable from it) in the specified file

All classes generated by ribgen are subclasses of RbnBase class. The typical methods of the classes are:

String getName() is an accessor to get the name of the element.
String getValue() is an accessor to get the value of the element.
void setValue(String value) is an accessor to set the value of the element.
void addChildFront(RbnBase obj) is a method to add the element to the front of the child elements.
void addChildRear(RbnBase obj) is a method to add the element to the end of the child elements.
java.util.List<RbnBase> getChild() is a method to get the list of child elements.

Figure 8 shows a snippet of a Java program using Ribbon APIs. It reads a representation from a file “test-in.rb” and instantiates Java objects, after which the program

gets a SerpPaymentAccount object,
gets a SerpBills object and a list from the object,
gets a SerpBill object from the list,
gets a SerpPrice object,
gets an Integer object,
sets 2000 to the SetPrice object,
writes all objects to another file “test-out.rb.”

Figure 9 shows two files, “test-in.rb” and “test-out.rb.”

```
Object obj;
RbnBase base;
SerpPaymentAccount pacc;
SerpBills bills;
SerpBill bill;
Integer iobj;

RbnApi api = new RbnApi("serp2init.rb");
obj = api.readRbnFile("test-in.rb");
pacc = (SerpPaymentAccount)obj;
pacc.setValue("Reno");
bills = pacc.getBills();
List<RbnBase> list = bills.getChild();
bill = (SerpBill)(list.get(0));
iobj = bill.getPrice();
iobj = 2000;
bill.setPrice(i);
api.writeRbnFile((RbnBase)obj, "test-out.rb");
```

Figure 8. An example of usage of Ribbon APIs

3. Current Implementation

The implementation work was performed on an Apple MacBook with Mac OS X 10.5.4, JRuby 1.1.2, and Java 1.6.0.05. The program generator ribgen is written in Ruby. It reads a Ribbon definition file, generates a Java program to set up, and generates Java classes corresponding to all elements. Moreover, it generates a class diagram to understand all generated classes and the inheritance hierarchy.

The Ribbon representation changes to graph-structured data using unique identifiers and references. When a Java program, using Ribbon APIs, writes graph-structured objects to a file, the objects with cyclic paths should be written only once. Ribbon APIs correctly serialize them using the algorithm mentioned in the paper[17].

NetBeans supports building, testing, and debugging Ruby applications[18]. Ribbon uses Ruby syntax to represent data so that it takes on advantage of NetBeans. For example, as shown in Figure 10, we can use auto indentation and syntax highlighting without modifying NetBeans or creating the plug-ins.

4. Summary

This paper described Ribbon, a new representation of structured data in Ruby syntax. One of its important features is that the representation can be interpreted and executed. Ribbon is useful for Java programs to read/write Java objects from/to persistent storage media, or to traverse the structured data.

A program generator was developed to create Ruby and Java programs from Ribbon definitions. In the author’s experience, productivity was improved in the design and implementation of programs that manipulate structured data.

Ribbon and its related tools are now being used in the development of commercial products, such as a diagram editor and a compiler front-end. The development and results will be published in a future paper.
Figure 10. Data definition and representation on NetBeans

References


Estimation of Software Testing Costs and Risks using Fuzzy Techniques

HARISH MITTAL
Department of Computer Science and Engineering
Vaish College of Engineering, Rohtak-124201(INDIA)
E-mail: harish.mittal@vcenggrtk.com

PRADEEP BHATIA
Department of Computer Science
Guru Jambheshwar University of Science & Technology, Hisar-125001(INDIA)
E-mail: pk_bhatia2002@yahoo.com

Abstract
Estimation of Effective cost of testing and risks of given software is the most challenging activity for an application. The overall life cycle cost of software associated with its failures exceed 10% of yearly corporations’ turnover. A major factor contributing to this loss is ineffective performance of software and systems verification validation and testing (VVT). A model, based on fuzzy logic technique, is presented in this paper. Cost estimation is not an exact science. Rather than using a single number for the estimated VVT cost, the VVT cost is regarded as a triangular fuzzy number which yields results comparable to estimations based on models using the probabilistic methods (less than 1% difference in VVT costs), while the results of Engel [1] differ by 4% from probabilistic methods. We can optimize the estimated cost for any application by varying one or more of the four arbitrary constants of this model.

Keywords
Verification, Validation, Testing, Fuzzy Technique, Triangular Fuzzy Number.

1. Introduction
A major factor contributing to software failures is ineffective performance of software and systems verification, validation and testing (VVT), throughout the lifecycle. The major problem associated with such cost estimates is that input data like costs, risk levels, VVT performance levels are imprecise in nature. Hence we use fuzzy logic technique to deal with such problems. The technique for fuzzification and defuzzification used in this paper is the same as used by us in “Optimization Criteria for Effort Estimation using Fuzzy Technique” [11]. To access economic impact of inadequate testing thirty Swedish companies were surveyed over a period of three years [21]. The findings indicated average losses of 9-16% from company annual turnover due to poor quality. The study reveals that the economic impact of inadequate infrastructure for software testing is small relative to the overall impact cost associated with Systems inadequate VVT. As a total VVT process covering all aspects of system behavior is not attainable due to cost and time considerations, so there is a need to quantify costs and risks of alternative VVT strategies and to develop an approach for minimizing the system’ life cycle cost. This paper presents a model for this purpose.

More and more software studies attempt to develop theoretical models for measuring and quantifying the testing process in the software arena. Easterbrook and Callahan [6] provide a case study describing a formal method for verification and validation of software when only partial specifications are available. This problem is particularly acute in the case of large and safety critical systems. The approach used entails translation of the requirements to formal specifications and construction of a formal system model. The models are analyzed for internal consistency and tested in a simulated form. Malaiya and Denton [15] dealt with the question of when to stop testing. The paper dealt with estimating the number of residual defects. The approach is based on constructing a model, which relates to the number of faults found in the past relative to the amount of money spent in the testing process. Fortunately, measuring the VVT process is becoming a central issue for many software-intensive organizations.

The paper is divided into 5 sections. Section 2 introduces related terms used to develop the proposed model. In section 3 the proposed model is presented, section 4 belongs to experimental study and the last section 5 is referred to conclusion and future scope.

2. Related Terms
2.1. Fuzzy Logic Concepts
2.1.1 Fuzzy Number
A fuzzy number is a quantity whose value is imprecise, rather than exact as in the case of ordinary single valued numbers [22]. Any fuzzy number can be thought of as a function, called membership function, whose domain is specified, usually the set of real numbers, and whose range is the span of positive numbers in the closed interval [0, 1]. Each numerical value of the domain is assigned a specific value and 0 represents the smallest possible value of the membership function, while the largest possible value is 1.

In many respects fuzzy numbers depict the physical world more realistically than single valued numbers. Suppose that we are driving along a highway where the speed limit is 80km/hr, we try to hold the speed at exactly 80km/hr, but our car lacks cruise control, so the speed varies from moment to moment. If we note the instantaneous speed over a period of several minutes and then plot the result in rectangular coordinates, we may get a curve that looks like one of the curves shown below; however there is no restriction on the shape of the curve. The curve in figure 1 is a triangular fuzzy number, the curve in figure 2 is a trapezoidal fuzzy number, and the curve in figure 3 is bell shaped fuzzy number. A triangular fuzzy number (TFN) is described by a triplet (α, m, β), where m is the model value, α and β are the right and left boundary respectively.
2.1.2 Fuzzy Logic

Fuzzy logic is a methodology, to solve problems which are too complex to be understood quantitatively, based on fuzzy set theory [4]. Use of fuzzy sets in logical expression is known as fuzzy logic. A fuzzy set is characterized by a membership function, which associates with each point in the fuzzy set a real number in the interval [0,1], called degree or grade of membership. The membership function may be triangular, trapezoidal, parabolic etc.

2.1.3 Fuzziness

Fuzziness of a TFN $(\alpha, m, \beta)$ is defined as:

\[
\text{Fuzziness of TFN (F)} = \frac{\beta - \alpha}{2m}, \quad 0 < F < 1. \tag{1}
\]

The higher the value of fuzziness, the more fuzzy is TFN. The value of fuzziness to be taken depends upon the confidence of the estimator. A confident estimator can take smaller value of $F$.

In order to model various types of level data, for example level of VVT performance, we define seven linguistic variables using hedges [14], as Very Low (VL) , Low (L), Medium Low (ML), Medium (M), Medium High (MH), High(H) and Very High (VH).

$$
\begin{align*}
\text{Table 2: TFN of various Levels} \\
\text{Performance level} & \quad \text{VL} & \quad \text{L} & \quad \text{ML} & \quad \text{M} & \quad \text{MH} & \quad \text{H} & \quad \text{VH} \\
\text{TFN of Performance level (} & \alpha, m, \beta \text{)} & \quad 0,0,0.167 & \quad 0.0167,0.333 & \quad 0.167,0.333,0.5 & \quad 0.333,0.5,0.667 & \quad 0.5,0.667,0.833 & \quad 0.667,0.833,1 & \quad 0.833,1,1
\end{align*}
$$
2.2 Basic VVT Concepts
Due to ineffective performance of software and systems the overall lifecycle cost associated with product failures exceeds 10% of yearly corporations’ turnover. The vast majority of industrial organizations spend considerable funds to promote product quality using sub-optimal VVT processes. However, a total VVT process, covering all aspects of systems behavior is not attainable due to cost and time considerations. There is a need to quantify costs and risks of alternative VVT strategies and to develop an appropriate approach for minimizing the systems lifecycle cost. VVT is a branch of software and systems engineering which focuses on ensuring that systems are delivered as error free as possible, are functionally sound and meet the users’ needs. Verification is “confirmation that products at end of a phase satisfy conditions at start of that phase”. Validation is “confirmation by stakeholders that specific intended use of a product is fulfilled”. Testing is “executing a program under conditions at start of that phase”. Observations suggest that the system definition phase of software and system lifecycle is the most critical in terms of VVT.

2.2.1 Phases of Life Cycle of a software
The life cycle of software development may be divided into following ten phases [11], and each of these phases has its own VVT activities.

P1: Definition,
P2: Design,
P3: Prototypes,
P4: Integration,
P5: Testing,
P6: Production,
P7: Usage,
P8: Maintenance,
P9: Upgrade,
P10: Disposal

2.2.2 VVT Activities, during the System Definition Phase (P1)
A set of VVT activities are specified for each phase. The eleven VVT activities [1], during the system definition phase are:
A1: Assess the project management plans.
A2: Assess customer functional requirements for consistency, feasibility and testability.
A3: Assess customer interface requirements for consistency, feasibility and testability.
A4: Develop general VVT plan and define VVT strategy.
A5: Determine specific VVT methods.
A6: Verify consistency between customer and system functional requirements.
A7: Verify consistency between customer and system interfaces requirements.
A8: Check that system requirements meet standards, laws and environmental impact.
A9: Check that system requirements meet company regulations and ethics.
A10: Conduct system requirement review (SRR).
A11: Obtain stakeholders input regarding definition (validate definition).

2.2.3 Ideal Cost
Ideal Cost of a phase is the aggregate of costs of all the activities of the phase, when each activity is fully performed.

If the ideal cost of an activity j of phase i is \( m_{ij} \), then the fuzzy ideal cost of the activity is taken as TFN \((\alpha_{ijk}, m_{ijk}, \beta_{ijk})\), where \( i=1,2,3,\ldots,10 \) and \( j=1,2,\ldots,11 \). Ideal cost of a system is the sum of ideal costs of all the phases.

2.2.4 Performance level of a VVT activity
When VVT activities are performed under some strategy then some VVT activities are performed partially, some are performed fully and some are not performed at all. Level of performance is decided by the strategy formed which is selected in accordance with the combined business objectives and vision of the systems’ stakeholders. Performance level is assigned one of the linguistic terms \{VL, L, ML, M, MH, H, VH\}. TFN of performance level \( k \) of activity j of phase i is represented by \((\alpha_{ijk}, m_{ijk}, \beta_{ijk})\).

2.2.5 Activity costs of a VVT activity
Activity cost of a VVT activity j of phase i is given by TFN \((\alpha_{ij}, m_{ij}, \beta_{ij})\).

2.2.6 Appraisal Risk
Some products exhibit defects due to inherent imperfection in the development, manufacturing or usage. The risk of detecting such deficiencies during the VVT process is labeled “Appraisal risk”. In such a case, the product will be returned for a corrective procedure and then it will be retested. This process of evaluating the product and, if defective, rectifying it, will continue until the evaluation does not indicate a flawed artifact.

2.2.7 Full Cost of Appraisal Risk
If the full cost of appraisal risk of any VVT activity j of phase i is \( m_{aij} \), then fuzzy full cost of appraisal risk is TFN \((\alpha_{aij}, m_{aij}, \beta_{aij})\), \( i=1,2,3,\ldots,10 \), \( j=1,2,\ldots,11 \).

2.2.8 Appraisal Risk Level of a VVT activity
Appraisal Risk Level of each VVT activity may accept any value from the linguistic terms \{VL, L, ML, M, MH, H, VH\}. TFN of appraisal risk level of activity j of phase i is taken as \((\alpha_{aij}, m_{aij}, \beta_{aij})\).

2.2.9 Appraisal Risk Cost of a VVT activity
Appraisal risk cost of a VVT activity j of phase i is computed by multiplying the full cost of appraisal risk of the VVT activity, the appraisal risk level and the performance level of the corresponding activity

\( (\alpha_{aij} \cdot \alpha_{ij} \cdot m_{aij} \cdot m_{ij} \cdot m_{ijk} \cdot \beta_{aij} \cdot \beta_{ij} \cdot \beta_{ijk}) \).

2.2.10 Impact risk
Any partially performed VVT activity or any VVT activity not performed at all constitutes an impact risk, these risks have stochastic effects on the system and, of course, they constitute undesirable expenditures. A single such risk may generate multiple impacts, affecting the system at different lifecycle phases.
2.2.11 Impact Risk Level
Impact risk level of each VVT activity may accept any value from the linguistic terms \{VL, L, ML, M, MH, H, VH\}, TFN of appraisal risk level of activity j of phase i is taken as \((\alpha_{ij}, m_{ij}, \beta_{ij})\).

2.2.12 Full cost of impact risk of a VVT activity
If the full cost of impact risk of any VVT activity j of phase i is \(m_{ij}\), then fuzzy full cost of appraisal risk is TFN \((\alpha_{bij}, m_{bij}, \beta_{bij})\), \(i=1,2,3,...,10, j=1,2,...11\).

2.2.13 Impact Risk Cost
As partially performed activity or not performed activity in any life cycle phase can cause impact risk in any subsequent phase, we calculate Fuzzy impact risk cost is computed by multiplying the full cost of impact risk of the VVT activity, the impact risk level and the reciprocal of the VVT performance level and is given by \((\alpha_{bij}*\alpha_{ijn}*m_{ij}*m_{ijn} * \beta_{bij} * \beta_{ijn} * \beta_{ijp})\).

3. Proposed Model
Historical results show that cost of verification, validation and testing of software is approximately 30% of total life cycle cost. So we take VVT cost in our model to be 30% of total life cycle cost. Full Cost of VVT activities during the system definition phase is taken as 2.56% of total life cycle cost. The ratio of estimated costs of VVT activities are taken as same as by Engel[1]. The estimated costs of the eleven VVT activities described in 2.2.2, during the system definition phase are in the ratio 2.5:4:3:9:8:5:4:7:12. Cost of appraisal VVT risks is taken as 1.32% of total life cycle cost. Full Cost of appraisal VVT risks of \(A_1\), \(A_2\), \(A_3\), \(A_4\), \(A_5\), \(A_6\), \(A_7\), \(A_8\), \(A_9\), \(A_{10}\), \(A_{11}\), supposing fuzzy component level of each VVT activity as very high, are in the ratio 1:3:2:2:1:4:5:2:3:4:6.


In addition the following assumptions are taken for our model
a) Each VVT activity is completely independent from each other
b) VVT performance level corresponds to a linear cost of VVT activity.

Single crisp estimate of a TFN \((\alpha, m, \beta)\) is taken as \(w_1(\alpha) + w_2(m) + w_3(\beta)\)

where, \(w_1, w_2\) and \(w_3\) are arbitrary constants. \(w_1, w_2\) and \(w_3\) are weights of the optimistic, most likely and pessimistic estimate respectively. Maximum weight should be given to the most expected estimate, F is fuzziness.

3.1 Appraisal Risk Level of various activities of Phase 1
The appraisal risk level of the various activities of phase 1 are given in table 3

3.2 Impact Risk Cost
The description of impact risk with corresponding impact risk level is given in table 4.

3.3 Reciprocal of VVT activity performance level
Reciprocal of VVT activity performance level of activity j of phase i is TFN \((\alpha_{ij}, m_{ij}, \beta_{ij})\), defined in table 5

---

Table 3: Appraisal risk level of various activities

<table>
<thead>
<tr>
<th>Activity</th>
<th>(A_1)</th>
<th>(A_2)</th>
<th>(A_3)</th>
<th>(A_4)</th>
<th>(A_5)</th>
<th>(A_6)</th>
<th>(A_7)</th>
<th>(A_8)</th>
<th>(A_9)</th>
<th>(A_{10})</th>
<th>(A_{11})</th>
</tr>
</thead>
<tbody>
<tr>
<td>VVT Risk level</td>
<td>VH</td>
<td>ML</td>
<td>L</td>
<td>VL</td>
<td>VL</td>
<td>L</td>
<td>ML</td>
<td>L</td>
<td>ML</td>
<td>ML</td>
<td>ML</td>
</tr>
</tbody>
</table>

Table 4: Impact Risks and Impact Risk Levels

<table>
<thead>
<tr>
<th>Impact Risk no.</th>
<th>Activity</th>
<th>Phase in which Impact Risk occurs</th>
<th>Impact Risk</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(A_1)</td>
<td>2</td>
<td>Project development delays during the design phase</td>
</tr>
<tr>
<td>2</td>
<td>(A_2)</td>
<td>3</td>
<td>Project development delays during the subsystems prototyping phase</td>
</tr>
<tr>
<td>3</td>
<td>(A_3)</td>
<td>4</td>
<td>Project development delays during the system prototyping phase</td>
</tr>
<tr>
<td>4</td>
<td>(A_4)</td>
<td>5</td>
<td>Project development delays during the testing phase</td>
</tr>
<tr>
<td>5</td>
<td>(A_5)</td>
<td>6</td>
<td>Project production delays during the production phase</td>
</tr>
<tr>
<td>6</td>
<td>(A_6)</td>
<td>4</td>
<td>Inferior and inadequate subsystem prototypes</td>
</tr>
<tr>
<td>7</td>
<td>(A_7)</td>
<td>5</td>
<td>System do not meet customer requirements</td>
</tr>
<tr>
<td>8</td>
<td>(A_8)</td>
<td>7</td>
<td>Customers dissatisfied with system</td>
</tr>
<tr>
<td>9</td>
<td>(A_9)</td>
<td>4</td>
<td>Inferior and inadequate subsystem prototypes</td>
</tr>
<tr>
<td>10</td>
<td>(A_{10})</td>
<td>5</td>
<td>System do not meet customer requirements</td>
</tr>
<tr>
<td>11</td>
<td>(A_{11})</td>
<td>8</td>
<td>System do not support on-going maintainability</td>
</tr>
<tr>
<td>12</td>
<td>(A_{12})</td>
<td>2</td>
<td>VVT is not planned in a consistent manner</td>
</tr>
<tr>
<td>13</td>
<td>(A_{13})</td>
<td>3</td>
<td>The VVT infrastructure is not adequate to perform the VVT process</td>
</tr>
<tr>
<td>14</td>
<td>(A_{14})</td>
<td>4</td>
<td>The subsystem VVT is not performed in a consistent manner</td>
</tr>
</tbody>
</table>
T, Appraisal Risks

Suitable values of \( w \), which is valued at approximately $250

Let 

\[
\alpha = 0.7m \quad \text{and} \quad \beta = 1.3
\]

4. Experimental Study

We take the case of a software upgrade of a planned avionics suite for a fighter aircraft [1]. The total life cycle cost of which is valued at approximately $250000K. Let \( F = 0.3 \), \( k = 1 \), then from table 1

\[
\alpha = 0.7m \quad \text{and} \quad \beta = 1.3
\]

Suitable values of \( w_1, w_2, w_3 \) can be obtained by optimizing cost for any application. We have taken for our calculation, 

\[
w_1 = 1, \quad w_2 = 4 \quad \text{and} \quad w_3 = 1.
\]

Cost of various VVT activities are given in table 6

<table>
<thead>
<tr>
<th>Activity</th>
<th>TFN of activity cost in K$</th>
<th>TFN of appraisal risk cost in K$</th>
<th>Performance Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A_1 )</td>
<td>(140,200,260)</td>
<td>(70,100,130)</td>
<td>VL</td>
</tr>
<tr>
<td>( A_2 )</td>
<td>(350,500,650)</td>
<td>(210,300,390)</td>
<td>M</td>
</tr>
<tr>
<td>( A_3 )</td>
<td>(350,500,650)</td>
<td>(140,200,260)</td>
<td>H</td>
</tr>
<tr>
<td>( A_4 )</td>
<td>(280,400,520)</td>
<td>(140,200,260)</td>
<td>MH</td>
</tr>
<tr>
<td>( A_5 )</td>
<td>(210,300,390)</td>
<td>(70,100,130)</td>
<td>VL</td>
</tr>
<tr>
<td>( A_6 )</td>
<td>(630,900,1170)</td>
<td>(280,400,520)</td>
<td>M</td>
</tr>
<tr>
<td>( A_7 )</td>
<td>(560,800,1040)</td>
<td>(350,500,650)</td>
<td>H</td>
</tr>
<tr>
<td>( A_8 )</td>
<td>(350,500,650)</td>
<td>(140,200,260)</td>
<td>MH</td>
</tr>
<tr>
<td>( A_9 )</td>
<td>(280,400,520)</td>
<td>(210,300,390)</td>
<td>L</td>
</tr>
<tr>
<td>( A_{10} )</td>
<td>(490,700,910)</td>
<td>(280,400,520)</td>
<td>M</td>
</tr>
<tr>
<td>( A_{11} )</td>
<td>(840,1200,1560)</td>
<td>(420,600,780)</td>
<td>VH</td>
</tr>
</tbody>
</table>
Table 7: VVT costs for System Definition Phase

<table>
<thead>
<tr>
<th>S.No.</th>
<th>Phase</th>
<th>VVT Activities</th>
<th>Impact Activities</th>
<th>Risk Activities</th>
<th>VVT Cost in (K$)</th>
<th>Activity Impact Risk Cost in (K$)</th>
<th>Appraisal Risk Cost in (K$)</th>
<th>Impact Risk Cost in (K$)</th>
<th>Total in (K$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>P1</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>3940</td>
<td>613</td>
<td>0</td>
<td>0</td>
<td>4553</td>
</tr>
<tr>
<td>2</td>
<td>P2</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>20</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>P3</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>47</td>
<td>0</td>
<td>47</td>
</tr>
<tr>
<td>4</td>
<td>P4</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>150</td>
<td>0</td>
<td>150</td>
</tr>
<tr>
<td>5</td>
<td>P5</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>176</td>
<td>0</td>
<td>176</td>
</tr>
<tr>
<td>6</td>
<td>P6</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>66</td>
<td>0</td>
<td>66</td>
</tr>
<tr>
<td>7</td>
<td>P7</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>454</td>
<td>0</td>
<td>454</td>
</tr>
<tr>
<td>8</td>
<td>P8</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>29</td>
<td>0</td>
<td>29</td>
</tr>
<tr>
<td>9</td>
<td>P9</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>57</td>
<td>0</td>
<td>57</td>
</tr>
<tr>
<td>10</td>
<td>P10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>44</td>
<td>0</td>
<td>44</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>11</td>
<td>40</td>
<td>0</td>
<td>3940</td>
<td>613</td>
<td>1044</td>
<td>5596</td>
<td></td>
</tr>
</tbody>
</table>

Table 8: Comparison of VVT costs for Systems Definition Phase using various models

<table>
<thead>
<tr>
<th>Cost Types</th>
<th>Crisp Cost in K$</th>
<th>Fuzzy (Engel [1]) Cost</th>
<th>Fuzzy Cost (Model Presented in this paper)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Stochastic</td>
<td>Probability</td>
<td></td>
</tr>
<tr>
<td>Actual cost</td>
<td>3934</td>
<td>100</td>
<td>3940</td>
</tr>
<tr>
<td>Appraisal cost</td>
<td>671</td>
<td>100.2</td>
<td>687</td>
</tr>
<tr>
<td>Impact cost</td>
<td>1023</td>
<td>101.2</td>
<td>1058</td>
</tr>
<tr>
<td>Total cost</td>
<td>5629</td>
<td>100.3</td>
<td>5832</td>
</tr>
<tr>
<td>CVM cost</td>
<td>6400</td>
<td>100.0</td>
<td>6202</td>
</tr>
</tbody>
</table>

Table 9: Estimated Total Cost as per present model versus Fuzziness

<table>
<thead>
<tr>
<th>Fuzziness</th>
<th>0.1</th>
<th>0.2</th>
<th>0.3</th>
<th>0.4</th>
<th>0.5</th>
<th>0.6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual Cost in K$</td>
<td>5629</td>
<td>5629</td>
<td>5629</td>
<td>5629</td>
<td>5629</td>
<td>5629</td>
</tr>
<tr>
<td>Estimated Cost as per present model in K$</td>
<td>5516</td>
<td>5596</td>
<td>5678</td>
<td>5757</td>
<td>5840</td>
<td></td>
</tr>
</tbody>
</table>

Comparison of estimated effort with actual effort at fuzziness 0.3 is given in fig 5

Fig. 5: Comparison of Estimated cost of proposed model (model 1) with other models at fuzziness 0.3

5. Conclusion and Future Research

The proposed study provides a fuzzy logic based precise approach to quantify Cost of software testing and risks. Most VVT cost data and relevant parameters are not available in precise form. Therefore, fuzzy modeling has the distinct advantage of deriving realistic information based on imprecise knowledge. The proposed study gives better results as compared to some earlier models. Furthermore, the calculation process is simpler than the process of earlier models. The methodology of fuzzy logic used for, in the proposed study, is sufficiently general and can be applied to other areas of quantitative software engineering.

References

1. Avner Engel, Mark Last 2007, Modeling software testing costs and risks using fuzzy logic paradigm, Journal of Systems and Software, v.80 n.6, p.817-835
Utilizing Concept Drift for Pervasive Supervision

Matthias Baumgarten, Kieran Greer, Maurice Mulvenna, Kevin Curran, Chris Nugent
Faculty of Computing and Engineering, University of Ulster,
Shore Road, Newtownabbey BT37 0QB, UK

Abstract - With an ever-growing complexity of current and future computational systems both users as well as developers are faced with an increasing overhead in order to manage, configure and most importantly, supervise them efficiently. In an ideal scenario, all relevant tasks would be performed by the system itself with no, or very little, user intervention. As a consequence a system needs to be able to adapt itself based on its own performance as well as on changes in its environment. Thus flexible control mechanisms are required that pervasively facilitate specific supervision tasks which are controlled, executed and also monitored from within the system that is under supervision. This research proposes a dedicated supervision mechanism that utilizes a phenomenon known as concept drift, which analyses short- as well as long-term behavior of a system in order to detect deviation therein. In theory, this could, ultimately, lead to systems capable of freely evolving within pre-defined boundaries that describe the functional correctness of the system under supervision.

Keywords: Pervasive Supervision, Autonomous Systems, Self-Adaptation

1. Introduction

There is strong motivation for new perspectives on generic supervision methodologies in order to provide more resilience in the face of ever more complex systems. In particular, future autonomic systems that ideally operate with no or only a limited user input require advanced supervision to control and adapt different variables of existing systems but more importantly to supervise the dynamic aggregation of individual autonomous working components as found in, e.g., service-oriented architectures or large scale sensor networks. In fact within such architectures the discovery and composition of services is only the first step towards fully ubiquitous and pervasive computing. While effective real-time coordination and management forms another obstacle, the pervasive supervision of such services as well as the underlying environments forms an important aspect for the success future systems. Furthermore, the real world models of individual services, or the underlying data they are based upon, are naturally of a volatile nature and as such likely to change constantly over time; thus continuously opening a gap between the actual model and the real world concept for which they were designed. This problem, referred to as concept drift, implies the constant adaptation of intelligent services and their underlying models in order to achieve a stable state within the pre-defined boundaries in which a system is allowed to operate. This could prevent a system becoming unstable by drifting outside of its operational parameters. In order to adapt to such changes effectively, a supervision mechanism needs to incorporate a computational model of the real world problem, which it intends to supervise.

This research seeks to explore the requirements for such a supervision mechanism, which needs to be capable of observing and analyzing complex and dynamically constructed models as reflected by a real world service or computational system. Furthermore, the method proposed will be able to operate at different levels of granularity with respect to the model supervised and as such supports the methodological framework for pervasive supervision as discussed in e.g. [1]. Subsequent sections explore the requirements for different observation methodologies for distributed and network like knowledge structures as proposed e.g. in [2]. Particular focus will be given to how such knowledge can be monitored and utilized to allow a system under supervision to evolve freely within the pre-defined boundaries that describe its functional correctness. In particular, the use of a lower and upper bound as well as the ideal state of individual variables will be explored.

The remainder of this paper is organized as follows. Section 2 provides insight into a generic supervision architecture and outlines aspects that are relevant to drift behavior while Section 3 introduces the problem of concept drift itself and maps relevant aspects to the supervision architecture. Section 4 lays down the foundations of a flexible service based architecture and sketches a prototype implementation before relevant
target architectures are discussed in Section 5. Finally, Section 6 concludes this paper and outlines future research directions.

2. Generic Supervision Architecture

In general, state of the art supervision methodologies and systems implement a closed control loop approach, as introduced in [8], which implement at least the following three concepts:

- **Monitoring**: Being the first step, this task is concerned with the gathering of information from the system that is under supervision. Additional tasks may include correlation and translation activities in order to pre-process incoming information and as such improving the quality of the monitored data and to reduce information overhead. Independent of the technique used to monitor an individual source, the goal can be summarized as to collect \( \omega = (\alpha, \tau) \), where \( \alpha \) is the observed value and \( \tau \) a timestamp referring to the time the value or event has been observed. A dedicated history service may also be used to build up a history of the observed source such as \( \theta = \omega_1, \omega_2, \omega_3, \ldots \). As for distributed systems, three distinct monitoring mechanisms are relevant. These can be summarized as:

  - **Event Based Monitoring**: The observed source posts relevant information at pre-defined intervals or at certain events to a centralized monitoring unit.

  - **Request Based Monitoring**: A centralized monitoring unit requests at pre-defined intervals or at certain events (e.g. an outside alert) relevant values from observed sources.

  - **Embedded Monitoring**: Individual components provide the functionality to monitor themselves. That is, a specialized component is linked to the component under supervision, which facilitates relevant monitoring tasks.

While specific circumstances may require the utilization of the first two mechanisms, the latter is probably preferred in most cases due to the fact that monitoring can be facilitated in a truly distributed fashion and because relevant values can be stored wherever they were produced thus reducing network traffic and response time.

- **Analysis**: This step incorporates methods that test for certain conditions, violations etc. that are of interest to the supervision process. Current analytical methods often implement a static rule or policy-based methodology where information are “hard coded” for each system and as such can not be adapted to changing conditions. While such methods may be sufficient for some applications, future autonomic systems will require more dynamic, highly intelligent and fully automated services that are able to operate in distributed context aware environments and as such are able to not only adapt the system under supervision but also the supervising system itself.

- **Reaction**: The reactive part of a supervision system closes the loop to actually achieve supervision. As such it is concerned with the identification, configuration and execution of relevant measures to counteract incorrect behavior, to invoke a specific recovery mechanism or to enforce selected rules, policies or other procedures. The challenge for this part is not to realize or control individual actuators, which execute individual corrective measures. On the contrary the correlation of a given problem that has been detected with the correct countermeasures at different levels of granularity (that is at different abstraction levels of the underlying system) can be seen as the biggest obstacle for pervasive supervision. The following reaction mechanisms are regarded to be of particular interest as they allow for supervision at different granular levels.

  - **Direct Reaction**: Corrective measures are invoked whenever an illegal state or violation is detected. This mechanism is relevant for autonomous micro-supervision systems that are fully aware of what to supervise, how to supervise it and finally how to react if something goes wrong.

  - **Descriptive Reporting**: If a system is not able to react on an illegal state or violation or if a system is forced to invoke countermeasures on a more global aspect of a system then individual components may choose to report their current ‘health’ to conceptually higher oriented supervision components. Obviously, such a reporting mechanism should be as complete as possible containing information about the sender, the fault, possible reasons (if the fault already has been analyzed locally) and, if known, relevant corrective measures.

Both mechanisms may be realized in a centralized way where corrective measures are identified and executed via a centralized system or, preferably, in a decentralized fashion, where individual components have full control to execute corrective measures. The latter obviously requires that each component is aware of the reactive measures it can invoke.

The same three concepts as discussed above are also relevant for a more long-term orientated and evolutionary-based supervision principle as envisioned.
here. However, in order to allow a system to evolve over time but at the same time assure the correctness of the underlying logic, additional methods are required which allow a system to (a) forecast changes in a supervised system; (b) enable predicting individual attributes based on past behavior or on other attributes and (c) detect critical states before they actually occur in order to execute countermeasures on time.

3. Concept Drift

Simplified, a concept of interest reflects the underlying model of a given service or application in a machine readable format. Stemming from the area of predictive analytics, a concept of interest can be described as a phenomenon that reflects a real world model or system, which is defined by its underlying contextual information or raw data. By nature, any such concept of interest is likely to change over time which is referred to as concept drift, which may occur in the underlying concept of interest if it is (a) not static, e.g., for dynamic models, (b) it can not be described in its entirety, e.g., incomplete models, or (c) if its values are subject to change, e.g., changing contexts [3]. Relevant systems that deal with the problem of concept drift and hidden contexts include STAGGER [7], IB3 [6] and FLORA [3]. For a more detailed review of these systems the authors refer to [5] where the problem of concept drift is addressed outside the scope of supervision and relevant related work is presented.

Considering that a concept of interest may depend on a hidden or very complex context it is often extremely difficult to design and implement them, let alone the modeling of the system that is intended to supervise it. The fact that the type and structure of the system under supervision is normally not known beforehand complicates the modeling thereof even more. In addition to the above, as the system under supervision may change, the underlying supervision model needs to change accordingly. This introduces an additional level of complexity within the supervision process that is difficult to deal with. Another problem is the handling of potentially very large numbers of properties that are required to reflect even relatively small concepts of interests and, furthermore, the implementation, adaptation and constant supervision of relationships among individual properties can be particularly difficult. Moreover, as such a system has to react to any type of data (relevant or not), the handling of noisy or even irrelevant data forms another obstacle for supervision. This problem can be particular hazardous as it may cause oversensitivity or insensitivity for the supervision systems with respect to their adaptability for changing conditions by erroneously interpreting noise as a type of false behavior. The levels of complexity of a concept of interest as well as the different types of data therein form another obstacle.

Unforeseen changes within a concept of interest, independent of the direction, are referred to as concept drift. For simplicity, a concept of interest is denoted by $\Phi$, whereas the concept of boundaries a given $\Phi$ can evolve in is denoted by $\Theta$ with $\Theta = (\beta^-, \beta^\ast, \beta^+)$, where $\beta^-$ reflects the lower bound, $\beta^\ast$ the ideal state and $\beta^+$ the upper bound. Within the literature, three distinct analytical types of drift behavior have been identified, these are:

- **Sudden drift behavior:** If a given $\Phi$ is unexpectedly and all of a sudden outside its boundaries and as such enters an illegal state then so called sudden drift has occurred.
- **Visible, continuous drift behavior:** such behavior is referred to if a given $\Phi$ is continuously shifting towards one of its boundaries. Moreover, such drift is called visible if a change can be observed in attributes that are directly attached to $\Phi$.
- **Hidden, continuous drift behavior:** similar to the above, except that drift is not directly observable, as it is not reflected by the attributes that are modeled into $\Phi$. Instead, the observed drift behavior is based on other aspects that are outside the scope of $\Phi$. Such behavior is particularly hazardous because it is the one that is most difficult to detect.

Based on the three types of drift behavior three analytical methods can be derived and summarized as follows:

- **Current state:** Analyzing the current state of $\Phi$ with respect to its own value / state and / or with respect to pre-defined boundaries as specified by $\Theta$ enables the detection of sudden drift behavior.
- **Forecasting:** Based on its own past context, analyzing if and at what time $t$, $\Phi$ reaches e.g., a critical state based on specified boundaries allows for the detection of continuous drift that is related directly to $\Phi$ (visible drift).
- **Prediction:** Predicting $\Phi$ for time $t^{\text{in}}$ based on other variables that are correlated to it, would enable the detection of hidden drift behavior.

Note that the term forecasting is used here to forecast attributes based on their own history, whereas the term prediction is used to predict attributes based on the context of other attributes.

3.1 Boundaries

Continuous observation of individual properties of a system under supervision would allow detecting if a
system slowly but continually moves into a specific direction or towards an unwanted state. Obviously, a supervision system is indifferent as to whether such a state represents only an annoyance or a more seriously and maybe even critical situation. It needs to (a) identify if a system is in an illegal state; (b) forecast the time it takes to reach an illegal state; and (c) to self−organize it around an ideal state which is either pre-defined or dynamically computed and constantly adjusted.

3.1.1 Lower and Upper Bound

As depicted in Figure 1, the lower and upper bound (β−, β+) define the borders that a system can safely evolve or operate in. As such the current state of Φ is constantly monitored and validated against these boundaries. Overlaid trends would then also allow the forecast of long term directions so that possible out of bound violations could be identified before they actually occur. Such forecasting capabilities may be used not only to visualize the current direction of drift but also to decide when a system may enter a critical state or vice versa which state a system might have in the future. Individual lower and upper bounds do not need to be static. Depending on changing contexts, individual boundaries may change as well. Furthermore, depending on the system under supervision, boundaries may also be of different primitive or complex types.

3.1.2 Ideal State Situation

More interesting in the context of autonomic computing is the organization of a system around an ideal state, in the sequel denoted by β′. Such a state reflects the status or situation a system needs to maintain to operate at its optimum behavior. Consequently, a supervision process should attempt to organize a relevant concept of interest as close to this state as possible. For instance, an autonomic system regulating the temperature of a building has to react on a multitude of factors, e.g., outside temperature, number of people in the building, open windows, etc. Nevertheless, based on its configuration its ultimate goal could be as simple as “keeping the temperature at 22 degree Celsius”. In this case the ideal state of the system under supervision is reflected directly by its goal. As depicted in Figure 2, such a service may detect the difference between the current state and the desired state and, if configured, executes various countermeasures.

3.2 Numeric vs. Symbolic Properties

The methods discussed so far specifically target the supervision of individual properties or to be more precise the correctness of the value a property can have or may drift towards over time. Naturally, a system under supervision contains data that is either numerical or symbolic in nature. While numerical data can normally be evaluated with statistical processes that are similar to time series evaluation, it may often prove to be more difficult to detect drift in symbolic values. This is based on the simple fact that no standardized ordering or distance related measurement is provided for to compare any two or more symbolic values. Another obstacle is formed by the fact that the system under supervision as well as individual variables thereof may exist in a distributed rather than in a localized environment which calls for more flexible mechanisms to synchronize the artificial ordering / distance of symbolic data sets. For instance, two data sets each containing the same symbolic values may require a different mechanism to order their values, which could be based on e.g. different context, usage, etc. Synchronizing such ordering among different objects is difficult as the underlying concepts the ordering is based upon may not be accessible by the supervision system.

One possibility is to build up a statistical record of the symbols recently monitored and then detect when different sets of symbols are being returned. For example, one could split up a history buffer of symbolic values into two parts and compare the values
in the second part with the values in the first part. If they are different then maybe drift has occurred. In addition, if the difference can be quantified this could be directly used to reason about the impact of the drift detected. However, specific evaluation criteria are normally required when dealing with symbolic and/or complex data. In general, symbolic values or other object-based data can be processed in the same way as numeric values if at least one of the following conditions is provided for.

- The set of symbolic values is finite and known beforehand: In this case a hash function may be used to provide a consistent order between all objects.
- A sort of ordering can be induced over the symbolic values to be expected to occur: Mapping symbolic values into a numerical equivalence is the simplest form to provide such ordering but it requires an a priori understanding of the values to occur.
- A dedicated logic is used to evaluate the incoming stream of data: To be provided either by the user or the system at runtime, a dynamic evaluation logic can be employed to map symbolic values into a measurable concept suitable to provide a consistent order among them.

4. Concept Drift Component

The following sections describe in more detail the technical requirements for realizing (a) the overall supervision cycle and (b) the component envisioned to facilitate the task of dealing with the various kinds of drift behavior as outlined previously. In general, it is envisioned that the Concept Drift Component (CDC) itself, as well as its sub-components, are realized as independent services to allow for maximum flexibility with respect to the dynamic orchestration of specific supervision tasks. In addition, the resulting architecture should not be limited by any structural or conceptual boundaries, as it needs to be able to handle different types of variables, different relational concepts among them, as well as different supervision goals at different levels of abstraction.

The architecture of the CDC is depicted within the center of Figure 3, comprises a number of components. Based on this, the following has to be accommodated for, to allow for individual supervision mechanisms, but in particular for the detection of drift behavior.

Firstly, the CDC must connect to a generic monitoring interface (left side) capable of providing access to relevant components, properties and protocols, in order to monitor relevant attributes that build up the concept of interest to be supervised. In addition, the monitoring interface must also be able to establish a connection to the actuator interface (right side) of the supervision architecture. This is required to facilitate supervision at different levels of abstraction, which is discussed in more detail in Section 4.2.

Secondly, an actuator interface, depicted at the right side of Figure 3, has to be established to which individual services, users, or to be more general, dedicated notification and corrective mechanisms can realise specific countermeasures and notification procedures. Such specific realisations may be registered to the CDC’s actuator registry and triggered in the case that drift behaviour is detected. Thirdly, the CDC itself provides the following:

- A generic service based architecture where individual modules can be dynamically (un-)loaded, to allow for the dynamic orchestration of supervision tasks.
- A communication bus enabling flexible communication among individual sub-services.
- A reaction registry where applications, services and other CDC’s can register with so that multiple stakeholders can be notified in case drift has been detected.

![Figure 3: CDC – Architecture](image)

4.1 CDC Orchestration

Each CDC is composed of specific sub-components that provide the functionality of the CDC in a sense that specific supervision tasks are configured via individual modules and then orchestrated within the CDC, which acts as a dedicated container component. Such modules may come in the form of services that implement the required functionality, to monitor the type of concept drift they were designed for. The general idea is that a specific CDC instance can be configured by dynamically loading services in order to perform specialized supervision of specific $\Phi$, rather than generic observation.

The detection of out of bound violations, ideal state monitoring and forecasting mechanism outlined in the previous section are examples of such services and have been implemented in the CDC framework.

Another useful service is represented by the Input Buffer, which as visualized in Figure 4, aims to de-
sensitize the system under supervision by e.g. averaging the stream of input values. For instance, such a service could read a series of values over a specified time interval, calculate the average and then forward this value to other components as the actual monitored value. Assuming that an evaluation function can be dynamically loaded, such a service may not only be used to de-sensitize a system but also to deal with outliers, noisy or erroneous data or to condense the input stream in the case of highly volatile systems.

![Figure 4: Input Buffer](image)

### 4.2 Hierarchical CDC Organization

So far, the architecture provides only limited supervision functionality in a sense that each CDC is only capable of supervising a single concept of interest. Obviously, a concept of interest is not limited to just primitive types but indeed have the potential to reflect more complex models or structures. Thus, the supervision within a single hierarchical level will, in most cases, not be sufficient. To achieve a more powerful and eventually fully pervasive supervision potential, relevant mechanisms need to be available at different levels of abstraction on (a) the system under supervision and (b) the supervising system. In order to provide for the hierarchical, or in fact network like orchestration of CDC’s, the selected feedback mechanism (currently the actuator interface) also needs to serve as an input source for the monitoring unit.

![Figure 5: Hierarchical CDC Organization](image)

As depicted in Figure 5, individual CDC’s may register with lower and / or higher oriented CDC’s to be notified of certain events. If connected to a reasoning mechanism, incoming notifications may be properly correlated and analyzed, irrespective of whether they stem directly from monitored concepts or from other CDC’s. While this not only enables the modeling of more meaningful concepts of interests it also allows for supervision at different levels of abstraction, yet maintaining the independence of individual supervision components. Thus, once available on a micro level, supervision may be propagated to a macro level providing supervision on different levels of abstraction. That is, if a problem cannot be solved by one component then it may be propagated to other components were it would be analyzed further until it can be solved.

### 5. Target Architectures

Naturally, any sort of computational system could benefit from the proposed supervision mechanism. Nevertheless, systems that are highly distributed, heterogeneous in nature and loosely coupled are particular relevant for the following reasons: (a) such systems comprise a large number of distinct components which need to be supervised individually as well as in relation to other components; and (b) higher level services may be formed by orchestrating any number of other services together. Thus, supervision is required on different levels of abstraction, such as the supervision of each individual service as well as the overall business goal and its execution.

### 6. Research Directions and Conclusions

Future research directions come from a multitude of areas and include individual supervision components as well as the overall architectural framework. For instance, what are the technological pre-requisites to build, configure and to interact with supervision pervasions? That is, which detailed capabilities are required from components and component orchestrations to support (a) monitoring and interactions, and (b) to model and to implement a supervision instance in an autonomic fashion?

Self-organization is seen as one of the key properties of autonomic systems. Within the context of supervision this translates into the dynamic configurations and maintenance of supervision configurations and, following the pervasive supervision paradigm, supervision subsystems are thus a part of such configurations. As a consequence, they have to follow the same organizational rules as the supervised systems.

Finally, situation-awareness is not yet properly reflected within current supervision approaches. Connecting the overall supervision architecture to relevant contextual sources could provide a new generation of supervision pervasiveness that is not yet possible. For instance, for a supervision component to
monitor the temperature of a room it needs to be connected to a temperature sensor. However in order to supervise it, it also needs to be connected to an actuator unit that allows changing the room temperature in some way or the other. Now, for a system to know that e.g. the window has been opened in order to freshen up the air allows additional reasoning about if, how, when and what countermeasures should be employed. How such situation-awareness can be achieved and how it can be successfully exploited for supervision is yet another open issue to be investigated in more detail.

This work has dealt with various approaches related to pervasive supervision and as such provides the foundation for a supervision mechanism that is based on drift behavior. In the first part, the closed control loop architectural has been discussed, which is the basic architectural paradigm employed by all supervision systems. While the analysis of concept drift is only one possible mechanism to allow for long-term supervision, the concept of interest and the real world problem they reflect are seen as key principles to enable autonomic services to self-evolve in context aware environments.

The main problem in this area is that a system under supervision can, at different levels of abstraction, contain any type of information. It is therefore not possible to create a generic evaluation function that is capable of evaluating any type of information, or concept of interest, respectively. This is based on the fact that an evaluation value is likely to be meaningless if it is derived in a generic fashion rather than based on the specific context of the system under supervision. Thus, it may be better to allow a system to try and self-organize itself in a way that micro versions of the whole supervision system exist at different levels. If such micro supervision systems maintain a stable state, the overall system should be stable too. On the other hand, there is a risk that if this state changes in any way, a system may recognize this as odd behavior and may react on this. The main advantage of this approach is that individual concepts of interests are, on a micro level, more likely to be of primitive types rather than complex structures. If this is the case then standardized evaluation criteria may be employed to (a) assess them and (b) to configure the supervision system. Realizing a current state analysis combined with more advanced forecasting and prediction mechanisms will allow the detection of sudden as well as gradual drift behavior. If embedded in a virtual realization of a system under supervision, such drift behavior could be detected at early stages and effective countermeasures, or a fail back mechanism, could be invoked on specific components of a system rather than safeguarding the overall system.

Not much has been said about the different application domains and scenarios in which such supervision is of interest and where or how it can be applied. This is because it is considered that such supervision mechanisms will become an integral part of future systems as they are required to successfully self-adapt small as well as fully distributed systems towards their optimal behavior.

8. Acknowledgments

Work supported by the project CASCADAS (IST-027807) funded by the FET Program of the European Commission.

10. References

Formal Construction of Deterministic Finite Automata Recognizing Intersection of Regular Languages

Nazir Ahmad Zafar¹, Nabeel Sabir² and Amir Ali²
¹Pakistan Institute of Engineering Applied Sciences
Nilore, Islamabad, PAKISTAN
Tel: +92-51-9290273-4; Fax: +92-51-2208070
Email: nazafar@pieas.edu.pk
²Faculty of Information Technology, University of Central Punjab
31-A, Main Gulberg, Lahore, PAKISTAN
Tel: +92-42-5755314-7; Fax: +92-42-5710881
Emails: {nabeel.bloch, amiralishahid}@ucp.edu.pk

Abstract: Automata play an important role in modeling behavior while Z is an ideal notation used for describing state space of a system. Consequently, an integration of automata and Z is required increasing their modeling power. Further, deterministic finite automata (DFA) may have different implementations and therefore it is needed to verify the transformation from diagrams to a code. If we describe a formal specification of a DFA, then confidence over transformation can be increased. To achieve this objective, we have combined automata and Z and a linkage is established between both of these approaches. At this level, we have given formal specification of DFA and its complement accepting regular languages. Then DFA accepting union of two regular languages is constructed after defining their complements. Formal construction of intersection of the languages is described. Formal specification of the linkage between Z and automata is analyzed and validated using Z/EVES tool.

Keywords: Integration of approaches, Automata, Formal methods, Z notation, Validation.

1 Introduction

Application of formal methods is a well researched area in specification of software systems [1]. As the use of computers has been increased, concern for the issues of complex systems has also grown up. Formal methods are a promising way of giving increased confidence in such systems. Formal methods are mathematical notations which are based on discrete mathematics such as logic, set theory or graph theory. Z and VDM are examples of logic based whereas automata and Petri-nets are examples of graph based formalisms. The design of a complex system, not only requires the techniques for capturing functionality but it also needs to model control behavior. Functions over any of the systems can be decomposed in terms of operations and constraints. Since, Z can be used to define state of a system and the operations over it and hence it is an ideal application of capturing functionality. Control over a system can be viewed in terms of visual flows in between the system’s functions. Automata are very powerful in modeling control over the system’s behavior. Consequently, it requires an integration of automata and Z increasing modeling power for complex systems, which is one of the objectives of this research.

Requirements capturing and design specification has been a persistent problem since last couple of decades. Specification of software systems has a vital role in software engineering. A formal specification of a system is its mathematical description that may be used to develop an implementation. It only describes what a system does, but it does not show how it does. For a given specification, it is possible to demonstrate correctness of design of a
system using various formal verification techniques. It has an obvious advantage over traditional approaches that incorrect design can be revised before implementing it.

As we know deterministic finite automata (DFA) may have different implementations [2], [3] and its execution time must be different. Further, we require preserving semantics of transformation from diagrams in DFA to specification in Z. If we describe formal specification of the DFA then transformation can be guaranteed. Further, a robust, consistent and accurate code can be expected.

In previous work of the authors, some important structures in automata are formalized [4], [5]. At this level of integration, we have given formal specification of a DFA accepting a regular language and its complement is defined. Then a DFA accepting union of two regular languages is constructed after defining their complements. Finally, formal construction of intersection of the given regular languages is described using deMorgon’s law and their relationship is established in terms of predicates. Formal specification of the whole set of activities is analyzed and validated using Z/EVES tool [6]. The main objectives of this paper are: (i) applying formal methods for systems development, (ii) an integration of automata and Z by giving a syntactic and semantic relationship, and (iii) reducing implementation issues of DFA.

Although integration of approaches is a well-researched area [7], [8], [9], [10], [11], [12] but there does not exist much work on formalization of graphical based notations. The work [13], [14] of Dong et al. is close to ours in which they have integrated Object Z and Timed Automata for some aspects. We have assumed their work as a starting point for this research. Another piece of good work is listed in [15] in which R. L. Constable has given a constructive formalization of some important concepts of automata using Nuprl. Some work of interest is also reported in [15], [2], [6]. In [16], a combination of Z with statecharts is established. A relationship is investigated in between Z and Petri-nets in [17], [18]. An integration of UML and B is given in [19], [20]. It is to be mentioned here that most of the researchers listed above have either taken some examples in defining integration of approaches or have addressed only some aspects of these approaches. And, there is a lack of formal analysis which can be supported by computer tools. Our work is different from others because we have given a generic approach to link Z and automata with a computer tool support.

In section 2, an introduction to formal methods is given. In section 3, applications along with limitations of finite automata are addressed. Integration of automata and Z notation is given in section 4. Conclusion and future work are discussed in section 5.

2 Introduction to Formal Methods

Formal methods are approaches based on the use of mathematical techniques and notations for describing and analyzing properties of software and hardware systems [21]. That is, descriptions of a system are written using notations which are mathematical expressions rather than informal notations. These mathematical notations are based on discrete mathematics and algebraic concepts, such as logic, set theory and graph theory. There are several ways in which formal methods may be classified. One frequently-made distinction is between model oriented and property oriented methods [21]. Model oriented methods are used to construct a model of a system’s behavior [22]. State transition diagrams, for example, are used to model the behavior as a set of states and transitions between them. Property oriented methods are used to describe software in terms of a set of properties or constraints that must be satisfied. The Z notation [23] is a model oriented approach which is based on set theory and first order predicate logic. It is also used for specifying the behavior of abstract data types and sequential programs. Z notation is used in our research for specification because it describes a state space of a system and a set of operations that may be performed on it at an abstract level of specification.

3 Limitations of Finite Automata

Finite automata are abstract models of machines based on mathematical techniques and notations which can be represented using diagrams. These models can be used to perform computations on inputs and an output can be generated by moving through a sequence of configurations. If we are able to reach any of the accepting configuration by using a series of computation then the given input is accepted. The transition function computes the next state based on the current state and an alphabet at every step of computation. The starting point from where a DFA is initiated is called an initial state. A set of information required in defining a finite automata are: (i) a set of non-empty finite number of states, (ii) a finite set of alphabets, (iii) a transition function, (iv) an initial state and (v) a finite set of final states.

Finite automata have various applications and it plays an important role in many areas of computer science and engineering, particularly, in modeling control behavior and compiler constructions. Modeling of finite state systems and defining a regular set of finite words are two important and traditional applications of it in computer science. Automata have emerged with several modern applications, for example, optimization of logic based programs, specification and verification of protocols [24]. The human computer interaction is another interesting research area where abstract machines can be used.

But diagrams in deterministic finite automata have been difficult to be used except for the very trivial cases, which is one of the major issues in representation of it diagrammatically. It is a well known fact that a given DFA may have different implementation methodologies and consequently its time and space complexity may vary, which is another issue in modeling using DFA. After identifying weaknesses and limitations of DFA, it is argued that this single approach cannot be used for modeling and specification of a complete system and consequently its integration is required with other useful approaches. Based on the need and reasoning given above, a linkage between Z and automata is proposed in this
paper which will be useful in modeling using integrated approaches. We believe that this integration is not useful at an academic but at an industrial level as well. A formal linkage of automata and Z notation is given in the next section. For this purpose, DFA is constructed which accepts an intersection of two given regular languages.

4 Integration of Automata and Z

A formal construction of DFA accepting an intersection of two regular languages is demonstrated. We start with the definition of finite automata which is a 5-tuple (Q, Σ, δ, q0, F), where (i) Q is a finite non-empty set of states, (ii) Σ is a finite set of alphabets, (iii) δ is a transition function which takes a state and an alphabet as input and produces a state as an output, (iv) q0 is the initial state and (v) F is a finite set of final states.

The above 5-tuple is a deterministic finite automata (DFA) if for each state q1 and for every alphabet a, there is a unique state q2 such that δ(q1, a) = q2. The definitions used here are based on well known books on Automata and Computation Theory [25], [26]. Let us suppose that L is a language over a set of alphabets Σ and is accepted by a machine DFA = (Q, Σ, δ, q0, F). We define complement Lc of language L as the language of all the strings based on Σ that are not words in L. Mathematically we can define the complement of L as:

Lc = {s, s is a string based on set of alphabets Σ | s \∈ L}.

Let DFA1 and DFA2 be two deterministic finite automata accepting the regular languages L1 and L2 respectively. First of all, we construct the DFAs accepting the languages L1c and L2c which are complements of the languages L1 and L2 respectively. Then a new DFA for (L1 \∩ L2)c is constructed which accepts all the words of L1c and L2c. Now by deMorgan’s Law we know that: L1 \∩ L2 = (L1c \∪ L2c)c, and hence the resultant regular language L1 \∩ L2 is an intersection of two given regular languages for which a new DFA can be designed. Formal construction is given using Z notation. A sequence of activities to give a complete construction of a DFA that accepts all the words which are both in L1 and L2 is: (i) DFA1 and its complement, (ii) DFA2 and its complement, (iii) Union of complements of DFA1 and DFA2, and (iv) complement of the resultant of the above union.

4.1 Deterministic Finite Automata

Let DFA1 = (Q1, Σ1, δ1, q01, F1) accepts a regular language L1. The definitions of the components are same as given above. To formalize the DFA1, Q1 and Σ1 are represented as Q and Σ respectively.

[Q, Σ]

In modeling using sets in Z notation, we do not impose any restriction upon the number of elements and a high level of abstraction is supposed. Further, we do not insist upon any effective procedure for deciding whether an arbitrary element is a member of the given collection or not. As a consequent, Q and Σ are sets over which we cannot define any operation. For example, cardinality to know the number of elements in a set cannot be defined. Similarly, subset and complement operations are not defined as well.

To describe a set of states for DFA1, a new variable states1 is introduced. Since a given state q is of type Q (q: Q) therefore states1 variable must be a type of power set of Q (states1: P Q). To describe a set of alphabets a variable, alphabets1 is used. Since a given alphabet is of type Σ therefore alphabets1 must be of type of power set of Σ. As we know that δ1 relation is a function because for each input (q1, a), where q1 is a state and a is in set of alphabet1 there must be a unique output q2 of type Q, which is image of (q1, a) under the transition function δ1. Hence we can declare δ1 as δ1: Q \times Σ \to Q. It is obvious that initial state q01 is of type Q (q01: Q). Now our last one construct F1 is required to be defined. The set of final states for our DFA1 is represented by finals1 and is of type of power set of Q (finals1: P Q).

For a moment, we have used mathematical language of Z notation which is used to describe various objects. It is to be mentioned here that the same language can be used to define the relationships between these objects. This relationship will be used in terms of constraints after composing the objects. The schema structure is used for composition of the objects because it is very powerful at abstract level of specification and it helps in describing a good specification approach. All of the five components of DFA1 are encapsulated and put in the schema named as FiniteAutomata1. The formal description of the schema in Z notation is given below. After designing DFA1, we are required to check whether a given string is accepted by it. For this purpose, the set of strings based on set of alphabets is declared as below.

Strings == seq Sigma

FiniteAutomata1

| states1: P Q |
| alphabets1: P Σ |
| delta1: Q \times Σ \to Q |
| q01: Q |
| finals1: P Q |
| strings1: P Strings |

q01 \in states1
finals1 \subseteq states1
∀q1: Q \mid q1 \in states1 \land a \in alphabets1

• \exists q2: Q \mid q2 \in states1 \land delta1(q1, a) = q2

∀st: Strings \mid st \in strings1 \land ran st \subseteq alphabets1

Invariants:

1. The initial state q01 must be an element of set of total states states1.
2. The set of final states finals1 is a subset of set of total states states1.
3. For each \((q_1, a)\) where \(q_1\) is an element of \(\text{states}_1\) and \(a\) is a member of \(\text{alphabets}_1\), there must be a unique state \(q_2\) such that:
\[ \text{delta}_1(q_1, a) = q_2. \]

4. Any string given as input to finite automata must be based on the set of alphabets of it.

After defining DFA1, we need to take its complement. For this purpose a schema \(\text{ComplementOfFA1}\) is defined. It contains DFA1 and some components in addition to it, which are required in defining complement. A relation is defined between the given DFA1 and its complement.

\[
\begin{align*}
\text{ComplementOfFA1} & \\
\Xi & \text{FiniteAutomata1} \\
\text{states}_1c & : P Q \\
\text{alphabets}_1c & : P \Sigma \\
\text{delta}_1c & : Q \times \Sigma \rightarrow Q \\
q01c & : Q \\
\text{finals}_1c & : P Q \\
\text{strings}_1c & : P \text{Strings} \\
\end{align*}
\]

\[
\begin{align*}
\text{states}_1c &= \text{states}_1 \\
\text{alphabets}_1c &= \text{alphabets}_1 \\
\forall q_1, q_2 : Q, a : \Sigma \mid q_1 \in \text{states}_1 \land q_2 \in \text{states}_1 \land a \in \text{alphabets}_1 \\
& \quad \cdot ((q_1, a), q_2) \in \text{delta}_1 \iff ((q_1, a), q_2) \in \text{delta}_1c \\
q01c &= q01 \\
\text{finals}_1c &= \text{states}_1c \setminus \text{finals}_1 \\
\text{strings}_1c &= \text{strings}_1 \\
q01c &\in \text{states}_1c \\
\text{finals}_1c &\subseteq \text{states}_1c \\
\forall q_1 : Q, a : \Sigma \mid q_1 \in \text{states}_1c \land a \in \text{alphabets}_1c \\
& \quad \cdot \exists q_2 : Q \mid q_2 \in \text{states}_1c \land \text{delta}_1c(q_1, a) = q_2 \\
\forall st : \text{Strings} \mid st \in \text{strings}_1c \land \text{ran} st \subseteq \text{alphabets}_1c \\
\end{align*}
\]

**Invariants:**

1. The set of total states in DFA1 and the total states in its complement are same.
2. The set of alphabets in both of the DFA1 and its complement are same.
3. The action of transition function is defined in same way over both DFA1 and its complement.
4. The set of initial states in both DFA1 and its complement are same.
5. The set of final states in complement of DFA1 is equal to difference of sets \(\text{states}_1\) and \(\text{finals}_1\).
6. The sets of strings in both, DFA1 and its complement, are equal because their alphabets are same.
7. The initial state \(q01c\) of complement of DFA1 must be in set of states \(\text{states}_1c\) of the same automata.
8. The set of final states \(\text{finals}_1c\) in complement of DFA1 is a subset of set of total states \(\text{states}_1c\).
9. For each ordered pair \((q_1, a)\) where \(q_1\) is an element of states and \(a\) is a member of alphabets there must be a unique state \(q_2\) in set of states \(\text{states}_1c\) such that:
\[ \text{delta}_1c(q_1, a) = q_2. \]
10. Any string given as input to \(\text{ComplementOfFA1}\) must be based on set of alphabets of the same automata.

### 4.2 DFA for Language L1 and its Complement

Let \(\text{DFA}_2 = (Q_2, \Sigma_2, q_02, F_2)\) accepts a regular language \(L_2\). Here \(Q_2\) is a finite non-empty set of total states, \(\Sigma_2\) is a non-empty finite set of alphabets, \(q_02\) is the initial state and \(F_2\) is a finite set of final states of \(\text{DFA}_2\). Similar to \(\text{DFA}_1\), the \(\text{DFA}_2\) is represented by a schema \(\text{FiniteAutomata2}\) as given below. The components of this schema are put in first part and the invariants are defined in the second part of it. The informal description of its invariants is repetition of properties as defined in \(\text{DFA}_1\).

\[
\begin{align*}
\text{FiniteAutomata2} & \\
\text{states}_2 & : P Q \\
\text{alphabets}_2 & : P \Sigma \\
\text{delta}_2 & : Q \times \Sigma \rightarrow Q \\
q02 & : Q \\
\text{finals}_2 & : P Q \\
\text{strings}_2 & : P \text{Strings} \\
\forall q_2 : Q, a : \Sigma \mid q_2 \in \text{states}_2 \land a \in \text{alphabets}_2 \\
& \quad \cdot \exists q_2 : Q \mid q_2 \in \text{states}_2 \land \text{delta}_2(q_1, a) = q_2 \\
\forall st : \text{Strings} \mid st \in \text{strings}_2 \land \text{ran} st \subseteq \text{alphabets}_2 \\
\end{align*}
\]

**Invariants:**

1. The initial state \(q01\) must be an element of set of total states \(\text{states}_1\).
2. The set of final states \(\text{finals}_1\) is a subset of set of total states \(\text{states}_1\).
3. For each \((q_1, a)\) where \(q_1\) is an element of \(\text{states}_1\) and \(a\) is a member of \(\text{alphabets}_1\), there must be a unique state \(q_2\) such that:
\[ \text{delta}_1(q_1, a) = q_2. \]
4. Any string given as input to finite automata must be based on the set of alphabets of it.

After designing \(\text{DFA}_2\), its complement is defined as a schema and represented as \(\text{ComplementOfFA2}\). Its components are defined in first part of the schema and invariants over it are identified and defined as predicates in the second part of it.
Invariants:

1. The set of total states in DFA2 and the total states in its complement are same.
2. The set of alphabets in both of the DFA2 and its complement are same.
3. The action of transition function is defined same way over both DFA2 and its complement.
4. The set of initial states in both DFA2 and its complement are same.
5. The set of final states in complement of DFA2 is a subset of set of total states of the same automata.
6. The sets of strings in both DFA2 and its complement, are equal because their alphabets are same.
7. The initial state q02c of complement of DFA2 must be in set of states states2c of the same automaton.
8. The set of final states finals2c in complement of DFA2 is a subset of set of total states states2c.
9. For each ordered pair (q1, a) where q1 is an element of states and a is a member of alphabets2c there must be a unique state q2 in set of states states2c such that: delta2c(q1, a) = q2.
10. Any string given as input to ComplementOfFA2 must be based on set of alphabets of the same automata.

4.3 Union of Complemented Languages

After defining DFAs and their complements for regular languages L1 and L2, we can give a formal definition of union of the complemented languages. It will be a language consisting of all the strings which are either not in L1 or not in L2. It is to be noted that \( \Sigma_1 = \Sigma_2 \). Mathematically, we can define this language as:

\[
\Sigma_1 \cup \Sigma_2 = \{ s, a \text{ string based on } \Sigma_1 | s \notin L1 \lor s \notin L2 \}
\]

The finite automata accepting the above language is given below. The set of properties in the resultant automata, accepting the language which is union of \( L_1^c \) and \( L_2^c \), is also given in terms of invariants defining union relationship of \( L_1^c \) and \( L_2^c \).
element of it is a final state in the DFA2$^c$.

7. All possible strings based on the alphabets of resultant automata are equal to the alphabets of DFA1$^c$.

8. The initial state $q0c$ is an ordered pair whose first element is the initial state of DFA1 and the second element is the initial state of DFA2. Further, $q0c$ is an element of the set of states of the resultant automata.

9. The set of final states of resultant automata is a subset of set of total states of the same automata.

10. For each input $((q1, q2), a)$ where $q1, q2$ are states and $a$ is any alphabet of the DFA, there is a unique state $(q3, q4)$ such that: $delta ((q1, q2), a) = (q3, q4)$.

11. Any string given as input is based on set of alphabets of the resultant DFA.

4.4 Construction of Intersection

We know that $L1 \cap L2 = (L1^c \cup L2)^c$ where formal construction of $(L1^c \cup L2)$ is done in the sub-section 4.3. Now complement of $(L1^c \cup L2)^c$ is a language accepting an intersection of L1 and L2. The DFA accepting it is represented by $ComplementOfUnionOfCompOfFAs$ and is defined as a schema given below.

```
ComplementOfUnionOfCompOfFAs
statesc: P (Q × Q)
alphabetsc: P Sigma
deltac: (Q × Q) × Sigma → Q × Q
q0c: Q × Q
finalsc: P (Q × Q)
stringsc: P Strings
```


Invariants:
1. Set of states in DFA and its complement are same.
2. Set of alphabets in both of the DFAs are also same.
3. The action of transition function is defined in a same way for both of the automata.
4. The initial states in both the automata are same.
5. The set of final states in complement of DFA is equal to difference of sets of states and set of final states.
6. The sets of strings in both of the automata are equal because their alphabets are same.
7. The initial state $q0c$ must be in the set of total states.
8. The set of final states $finalsc$ in the complement of automata is a subset of set of total states $statesc$.
9. For each $((q1, q2), a)$ where $(q1, q2)$ is an element of $statesc$, $a$ is a member of alphabets there is a unique state $(q3, q4)$ such that:

   $$delta ((q1, q2), a) = (q3, q4).$$

10. Any string given as input to a complement DFA is based on alphabets of it.

5 Conclusion

The main objective of this research was proposing an integration of Z notation and automata. In this paper, we have described formal specification of an algorithm which can be used to construct deterministic finite automata accepting a language which is an intersection of two given regular languages. Although a part of integration between Z and automata is treated, in this paper, but we have observed that this approach can be extended to give a comprehensive formal linkage between these approaches.

We have identified an interesting relationship between some parts of these approaches. Our idea is original and important because we have observed after integrating that a natural relationship exists there. This work is important because formalizing graph based notation is not easy as there has been little tradition of formalization in it due to concreteness of graphs [21]. An extensive survey of existing work was done. Some interesting work [13], [14], [27], [28] was found but our work and approach are different because of abstract and conceptual level integration of Z and automata. Why and what kind of integration is required, were two basic questions in our mind before initiating this research. Automata is best suited for modeling behavior of a system while Z is an ideal notation to be used describing state space and then defining operations over it. This supporting behavior of Z forces us to integrate it with automata.

Formalization of some other concepts in automata is under progress and will appear soon. We have taken some assumptions in this integration, for example, in union of two DFAs, it was assumed that their sets of alphabets are same. Such assumptions were taken for simplicity of construction. In future work, a more generic integration will be proposed after relaxing such assumptions.

6 References


Process Maturity Model for Software Product Lines for SMEs

Sathya Ganeshan and Dr. Muthu Ramachandran
Innovation North: School of Computing, Leeds Metropolitan University, Leeds, UK

Abstract: Software product line based approach to software development has emerged for not only to speed up the development of new products and also to improve the quality of software systems explicitly by composing system artefacts. The process improvement frameworks have been successful for large scale systems but have failed to achieve in other smaller business (SMEs). In this paper we have proposed a framework (PPI model) for product line process improvement which combines software process improvement and software product lines.

Keywords: Software Product Lines, Process maturity, SME

Introduction

In the modern software world, every organisation strives towards meeting common standards that decide the quality of the product it generates and as a result, its survival. The emerging trends have laid more stress on not just the quality of the product but also on the processes and the management schemes employed in developing that product. Initially the event of winning a contract from a client requires that an organisation is able to produce its reliability through its past records of delivering quality products. In these days where a project is divided into several sub-projects, clients are critical about selecting the right service-provider for their project. One of the key factors that help in this decision making process, along with past records is the level of maturity of processes and expertise an organisation practices to achieve its target. In this context it becomes vital for any concern to be aware of its own maturity, not just for winning contracts but also to continuously review and improve its own performance. Process improvement models and metrics allow an organisation to measure its level of expertise and to strive towards further improving its maturity. CMM developed by SEI and funded by DOD is a well known approach for measuring process maturity in an organisation.

Such standards for measurement and improvement are not known to exist for an organisation that has switched to or in the process of switching to Software Product line approaches as its development and management process. We observed that it would be a great step forward if it was possible to measure the level of maturity an organisation has reached in the context of software product lines. In other words, we propose a model to measure the level of maturity of reuse an organisation has achieved through Software Product Lines. Parallel to this, the next task would be to derive a set of rules and guidelines similar to the representations in CMM that define each level and that are to be adhered to achieve a systematic shift from one level to the next higher one. We have proposed the standards and models that allow a concern to measure the level of maturity in reusability achieved through software product lines while presenting with a set of rules and guidelines that helps achieving the next phase of maturity.

Software product line based approach to software development has emerged for not only to speed up the development of new products but also to improve the quality of software systems explicitly by composing system artefacts. The process improvement frameworks have been successful for large scale systems but have failed to achieve in other smaller business (SMEs). In this paper we have proposed a framework for product line process improvement which combines software process improvement and software product lines.

Product-line based development aims to achieve higher level of reuse. There have been two major approaches that are known to support product line based development known as commonality & variability analysis and domain analysis. Most of the current approaches to software product lines [1]; [2]; [3] provides some interesting insights and they mostly are based on UML based components and notations (extending UML) and are insufficient in many ways when it comes to support large scale domain modelling for product line reuse. Commonality and variability analysis introduces analyzing systems for common feature and modelling variability parameters [4]. Kuusela and Savolainen [5] say one of the most promising reuse approaches is product families. Most of these methods and approaches are mainly focused on design which we believe is too late for product-line based development. Also we consider existing methods on requirements capturing do not go far beyond functional analysis.

Mannion and Kaindl [6] state that a product line model consists of numbered, atomic, natural language requirements, a domain dictionary and a set of discriminants, in which a discriminant is any requirement which differentiates one system from other. They also suggest that there are many open issues about reuse and requirements and in particular reuse of requirements. It is interesting but doesn’t go beyond our goal of analysing and extracting requirements for a product-line based software development.
Therefore current practices in requirements engineering do not support product families explicitly. Most of these works also have not been exploited systematically for managing product line development. Also, a well structured methodology for developing and maintaining core assets have not been widely discussed. Our approach considers product-line based development during domain analysis and product line requirements. We have derived well practiced guidelines and rules which aid in achieving these goals during the development phase of a product line. Our previous work considered commonality and variability analysis during use case modelling [7]. This paper identified a framework for managing product family and configuring a specific product or an application from the family. Our approach focuses mainly on:

- The use of scope management techniques and domain boundary
- A product configuration language
- Use of commonality and variability techniques at the feature and use case level.

Some of the existing works [8] does provide an approach to derive a systematic approach towards development. But these methods are manual and only the process of commonality analysis has been automated. They also provide approaches to compare models rather than define a set of rules for a concern to achieve a particular level of process maturity. Moreover these methods aim at finding the best practices within a process by comparing complex systems, whereas our approach defines a set of clear guidelines for a concern to reach a particular level of maturity in reuse thorough software product lines.

Software Process Improvement (SPI) and Software Product Lines (SPLs)

For any organisation with the intention of improving its process and management, the key indicators that denote a failure of their current approach are usually the following factors:

- Poor quality of product:
- Unpredictable cost
- Constant missing of targets
- Mismanagement
- Poor morale

Poor quality results from the failure of the process used to develop that product. It can be both product development process and the management team that was involved in developing the product. Cost is also a remainder of failure in the side of the management for assuming unreliable cost targets and failing to do so. Missing targets and the feeling of general mismanagement can be attributed again to the management side which results in poor morale. The above factors remind us of the importance of process and the efficiency of the management team that empowers this process. Imposing software product line approach automatically presents a way for a systematic process for development and management. But it is necessary to have a standard to measure and structured implementation of product line approach so as to make the adoption process smooth and cumulative. There are no existing methods or models to do so and it might highly aid any concern as a starting point with an intention of process improvement.
Figure 1 shows the constant refining process that underlies every aspect of maturity models. It shows the two key factors i.e. process and the team that utilizes the process in an endless cycle of review and refinement while also in communication with each other. In the CMMI models developed by SEI [9] in association with Carnegie Mellon University, Management has been decomposed into two factors as people and technology. We consider people as part of management and since technology is also the choice of the management, we represent management as a combination of these two. In our model as shown above we propose a continuous management review which should mainly focus on business, market and risk analysis for future products and in the process there should be a continuous focus on improvement effects on new products and cost-benefits analysis. Currently this is lacking in SEI product line framework.

Maturity models are used to address the issues mentioned in section two with the following goals.

- Shares knowledge gained by a community
- Establishes uniformity within which aids in better communication among different groups
- Identifies a framework and a path towards further refinement in process and management
- Identifies priority among tasks for improvement

Key to using a maturity model is sharing of knowledge through best practices gained by communities involved in development and management over a long period. This knowledge allows identifying areas of improvement and standardisation within processes. Without a structured technique, even simple process improvement steps are sometimes overlooked. In this case, a standardised model becomes more essential in identifying and implementing these changes to the process methods and the team that runs it. By accepting to follow certain standardised process recommended by these models, the growth of the concern can be achieved systematically monitored constantly to identify refinements.

For any concern that desires to switch to process improvement through SPL approach, the first major step is to identify the tasks to be performed and their priority. This allows a smooth transition from the present status to a product line based process development approach. Once this transition is complete, the sequel would then be to continue by establishing in the prior levels of maturity. Once refined in a particular level, depending upon the size, business goals and targets of the concern a shift from one level to the next becomes possible.

**Product Line Process Improvement (PPI) for SMEs**

This section will focus on our new framework for Product line Process Improvement for SMEs (PPI). Based on the
process maturity of the current process used in a concern, it can be categorised into five levels:

- **Initial**: The process used is mostly add-hoc. There might be some unconscious reuse in practice. But there is no intention of reuse to the advantage. Generally the process is chaotic and no particular system is in place.

- **Selective**: There is some kind of reuse is present. But it is practiced without any long term plan. For example segments of code, use cases etc are reused to save time. Where ever reuse saves cost, money and time without compromising quality, it is employed. This gives a brief understanding of the benefits to come if reuse is followed.

- **Biased**: There is an intention of reuse to the advantage. Components, use cases, code, test cases etc are made abstract so that they could be reused. Generally items that can be reused are preferred in places of one-off components.

- **Designed**: There is a long term plan and a firm belief towards reuse. The process and management are adapted to suit reuse. Core assets, common architecture, family oriented software development are all in use. Particularly, reuse is taken advantage by the organisational structure. Investment is made in the domain engineering process which is building up of the facility that generates products rapidly.

- **Refined**: Measures are collected from every aspect and there is a constant refinement of the process as the product line matures. At this stage a group is well versed with SPL based

reuse and the plans laid further are well defined, long-term and totally family based.

Each level is a well established phase with its own set of guidelines and rules to establish a group in a particular level. It should be noted that these levels are cumulative, which implies that a group at level three follows the accumulated rules combined from all prior levels. It should also be observed that bypassing a particular level to move to the next higher level might be counter productive since it is essential to be well versed in lower levels to move on to the next one. Now let’s look closely into each level and the associated guidelines.

**Product Line Key Process Areas (P-KPAs)**

Each level or phase has guidelines that are geared towards improving the process by shifting slowly towards family based software production. The tasks included in the selective phase include:

- Requirements Management.
- Identify common aspects
- Identify reusable opportunities
- Quality assurance

Figure 2 shows the five phases of Process Maturity Model for Software Product Lines that are Initial (add hoc and opportunistic), introduced and adapted Commonality and Variability Analysis, Biased (well established for components composition and system integration techniques), Designed (well managed to control variability and new requirements), and Refined (well controlled and measured).
Figure 2: PPI Model

**Initial**
- Product focused
- Established Configuration Management

**Biased**
- Multi project and product development cooperation
- Organisational process focus and training
- Project management
- Identify priority of reusable entities
- Create asset storage facility

**Designed**
- Commonality & variability analysis

**Refined**
- Core asset development and management
- Organisational structuring
- Training packages
- Coordination and process engineering
- Review teams
- Quality assurance

**Refined**
- Quality management
- Monitor and refinement process
- Matured long-term predictions

---

Domain and application engineering followed, core assets, common architecture, commonality and variability management are well employed

Adopted C &A

Use case driven RE or any popular methods established
Starting adopt CA

Planned for reuse, components, architecture etc.

Initial
- Not planned for SPL
- Product Focused
- Established CM (At least baseline in version control)
- Reasonable requirements capturing technique
- Customer driven

Initial
- Product focused
- Established Configuration Management

Biased
- Multi project and product development cooperation
- Organisational process focus and training
- Project management
- Identify priority of reusable entities
- Create asset storage facility

Designed
- Commonality & variability analysis

Refined
- Quality management
- Monitor and refinement process
- Matured long-term predictions

Refined
- Measures collected, analysed and refined to increase maturity

Biased
- Planned for reuse, components, architecture etc.

Initial
- Not planned for SPL
- Product Focused
- Established CM (At least baseline in version control)
- Reasonable requirements capturing technique
- Customer driven

Initial
- Product focused
- Established Configuration Management

Biased
- Multi project and product development cooperation
- Organisational process focus and training
- Project management
- Identify priority of reusable entities
- Create asset storage facility

Designed
- Commonality & variability analysis

Refined
- Quality management
- Monitor and refinement process
- Matured long-term predictions

Refined
- Measures collected, analysed and refined to increase maturity

Biased
- Planned for reuse, components, architecture etc.

Initial
- Not planned for SPL
- Product Focused
- Established CM (At least baseline in version control)
- Reasonable requirements capturing technique
- Customer driven

Initial
- Product focused
- Established Configuration Management

Biased
- Multi project and product development cooperation
- Organisational process focus and training
- Project management
- Identify priority of reusable entities
- Create asset storage facility

Designed
- Commonality & variability analysis

Refined
- Quality management
- Monitor and refinement process
- Matured long-term predictions

Refined
- Measures collected, analysed and refined to increase maturity

Biased
- Planned for reuse, components, architecture etc.

Initial
- Not planned for SPL
- Product Focused
- Established CM (At least baseline in version control)
- Reasonable requirements capturing technique
- Customer driven

Initial
- Product focused
- Established Configuration Management

Biased
- Multi project and product development cooperation
- Organisational process focus and training
- Project management
- Identify priority of reusable entities
- Create asset storage facility

Designed
- Commonality & variability analysis

Refined
- Quality management
- Monitor and refinement process
- Matured long-term predictions

Refined
- Measures collected, analysed and refined to increase maturity

Biased
- Planned for reuse, components, architecture etc.

Initial
- Not planned for SPL
- Product Focused
- Established CM (At least baseline in version control)
- Reasonable requirements capturing technique
- Customer driven

Initial
- Product focused
- Established Configuration Management

Biased
- Multi project and product development cooperation
- Organisational process focus and training
- Project management
- Identify priority of reusable entities
- Create asset storage facility

Designed
- Commonality & variability analysis

Refined
- Quality management
- Monitor and refinement process
- Matured long-term predictions

Refined
- Measures collected, analysed and refined to increase maturity

Biased
- Planned for reuse, components, architecture etc.

Initial
- Not planned for SPL
- Product Focused
- Established CM (At least baseline in version control)
- Reasonable requirements capturing technique
- Customer driven

Initial
- Product focused
- Established Configuration Management

Biased
- Multi project and product development cooperation
- Organisational process focus and training
- Project management
- Identify priority of reusable entities
- Create asset storage facility

Designed
- Commonality & variability analysis

Refined
- Quality management
- Monitor and refinement process
- Matured long-term predictions

Refined
- Measures collected, analysed and refined to increase maturity

Biased
- Planned for reuse, components, architecture etc.

Initial
- Not planned for SPL
- Product Focused
- Established CM (At least baseline in version control)
- Reasonable requirements capturing technique
- Customer driven

Initial
- Product focused
- Established Configuration Management

Biased
- Multi project and product development cooperation
- Organisational process focus and training
- Project management
- Identify priority of reusable entities
- Create asset storage facility

Designed
- Commonality & variability analysis

Refined
- Quality management
- Monitor and refinement process
- Matured long-term predictions

Refined
- Measures collected, analysed and refined to increase maturity

Biased
- Planned for reuse, components, architecture etc.

Initial
- Not planned for SPL
- Product Focused
- Established CM (At least baseline in version control)
- Reasonable requirements capturing technique
- Customer driven

Initial
- Product focused
- Established Configuration Management

Biased
- Multi project and product development cooperation
- Organisational process focus and training
- Project management
- Identify priority of reusable entities
- Create asset storage facility

Designed
- Commonality & variability analysis

Refined
- Quality management
- Monitor and refinement process
- Matured long-term predictions

Refined
- Measures collected, analysed and refined to increase maturity

Biased
- Planned for reuse, components, architecture etc.

Initial
- Not planned for SPL
- Product Focused
- Established CM (At least baseline in version control)
- Reasonable requirements capturing technique
- Customer driven

Initial
- Product focused
- Established Configuration Management

Biased
- Multi project and product development cooperation
- Organisational process focus and training
- Project management
- Identify priority of reusable entities
- Create asset storage facility

Designed
- Commonality & variability analysis

Refined
- Quality management
- Monitor and refinement process
- Matured long-term predictions

Refined
- Measures collected, analysed and refined to increase maturity

Biased
- Planned for reuse, components, architecture etc.

Initial
- Not planned for SPL
- Product Focused
- Established CM (At least baseline in version control)
- Reasonable requirements capturing technique
- Customer driven

Initial
- Product focused
- Established Configuration Management

Biased
- Multi project and product development cooperation
- Organisational process focus and training
- Project management
- Identify priority of reusable entities
- Create asset storage facility

Designed
- Commonality & variability analysis

Refined
- Quality management
- Monitor and refinement process
- Matured long-term predictions

Refined
- Measures collected, analysed and refined to increase maturity

Biased
- Planned for reuse, components, architecture etc.
We have developed a list of assessment questionnaires which are categorised as follows:

- Application domain experience
- Organisation Support: People and Management
- Business, Performance, SPI
- Requirements Engineering
- Commonality and Variability management
- Architecture, Design, and Technology

**Web Based Assessment Tool and Evaluation**

Currently a web based tool has been designed and distributed to few SMEs and the results are encouraging.

The initial assessment was conducted with three SMEs (who have less than 50 engineers) such as A, B and C and they are all satisfied a percent of assessment checklists for each category such as domain experience, organisational support, core business, RE, C&V management and their design and technology. As we can see from the above table C&V was the least satisfied which was common across all SMEs.

**Conclusion**

Software product line can achieve its promises if adopted systematically from commonality and variability analysis standpoint which must be a part of the requirement engineering activity. This is evident from our model and assessment results from SMEs.

**References**


Abstract: Software Product line based development has emerged to address reuse and shorten development lifecycle. However, not much has been learned about the earlier part of the product-line lifecycle. We have proposed a method, approach, and techniques for adopting product line based software development. Our approach is based on domain engineering method which focuses mainly during product line requirements identification.

Keywords: Software product line, product family, reuse, components, assets

Introduction

Domain Engineering of product lines is one of the least explored areas in the point of view of software product lines. Domain Engineering has been identified to synthesis software system supporting reuse explicitly [1], [2]. The goal of Domain Engineering is to establish a family production facility, common design and a process for building family members based on the design. And application engineering uses this facility to produce the family members [3]. It has been well recognized, that core assets form a vital part in the successful production, evolution and maintenance of product lines [4]. So far there isn’t a clear guideline on how to achieve this in-practice. For example how do identify core assets during the requirements for a product development so that we can engineer it carefully with all the utilities of Software Engineering. There is still a huge demand for well engineered components and assets that possess all the –ilities of software engineering attributes.

Product-line based development aims to achieve higher level of reuse. There have been two major approaches that are known to support product line based development known as commonality & variability analysis and domain analysis. Most of the current approaches to software product lines [3], [5], [6], [7] provides some interesting insights and they mostly are based on UML based components and notations (extending UML) and are insufficient in many ways when it comes to support large scale domain modeling for product line reuse. Commonality and variability analysis introduces analyzing systems for common feature and modeling variability parameters [8]. Kuusela and Savolainen [9] say one of the most promising reuse approaches is product families. Most of these methods and approaches are mainly focused on design which we believe is too late for product-line based development. Also we consider existing methods on requirements capturing do not go far beyond functional analysis. Mannion and Kaindl [10] state that a product line model consists of numbered, atomic, natural language requirements, a domain dictionary and a set of discriminants, in which a discriminant is any requirement which differentiates one system from other. They also suggest that there are many open issues about reuse and requirements and in particular reuse of requirements. It is interesting but doesn’t go beyond our goal of analysing and extracting requirements for a product-line based software development.

Therefore current practices in requirements engineering do not support product families explicitly. Most of these works also have not been exploited systematically for managing product line development. Also, a well structured methodology for developing and maintaining core assets have not been widely discussed. Our approach considers product-line based development during domain analysis and product line requirements. We have derived well practiced guidelines and rules which aid in achieving these goals during the development phase of a product line. Our previous work considered commonality and variability analysis during use case modeling [11]. This paper identified a domain engineering method for managing product line development. Our approach focuses mainly on:

- The use of domain engineering methods.
- Mapping product line requirements to product assets such as use cases and features.
- Knowledge based approach to managing product line development.
- Use of commonality and variability techniques at the features and use cases level.

In the following sections, we will discuss our approach, methods, techniques, and on going development on tool support.

**Knowledge Based Domain Engineering Method**

Knowledge engineering and domain engineering combined together can offer much to automate product line based software development and can help to achieve production assembly. We have successfully applied guidelines on commonality and variability during use case driven modeling to extract common features and variable parameters during earlier part of the product line driven development process. All these must be supported by a systematic framework and process.

**Figure 1: A Framework for Managing Product Line Development**

The Figure 1 illustrates a framework for integrating domain modelling and product line approaches. The framework provides support for analysing product line requirements and matching those onto existing assets (such as requirements [use cases, features], design, testing, components, guidelines). The domain and design knowledge can be encoded as rules or allow system integration engineers to choose manually by browsing through assets classification hierarchy/lattice. Note that here we present a software engineer as a product line or system integration engineers. Boehm has predicted that the future of software engineering job market will be dominated by system integrators and infrastructural engineers. The integrator can retrieve domain knowledge whereas domain experts help to enrich domain knowledge over the period of product family.

The FARE method provides a list of common and variable features and assets for a specific product line development. Finally the system integration engineers can assemble systems much quickly by shortening software development lifecycle.

Our IDEM (Integrated Domain Engineering Method) starts at the point when the integration engineer receives a new requirement or a set of requirements. Software Engineers can then match these requirements onto the existing assets. This is followed by passing these requirements through a set of development & design guidelines which identify commonality, variability and the criticality of the requirements. Further domain experts and integration engineers Employ product-line management constraints, FARE analysis before delivering the
commonality and variability attributes, features and assets which in turn are used to enrich and develop components, use cases and product assembly line.

Some of our guidelines on domain engineering method are presented as follow:

- Identify product line family tree. Product1.1, Product 1.2, Product 1.3, etc.
- Identify commonality features and variable features in new product versions. Draw a ven diagram illustrating shared features which can help to identify common and core assets earlier in the lifecycle. These should then be reflected on to a family tree.
- Classify commonality features (this is also know as domain classification which is rather difficult to predict before hand if no family product exists)
- Identify and develop three classes of reusable components for each branch, generics, and building blocks with fixed interfaces. Develop at least one component for each branch in the commonality tree.
- Select object classes when both clients and entity operates in the same infrastructure/environment and the process; select components when the clients and the entity operates in the same process but different environment; select web service architectures when clients and the entity operates in different process and different environment.

These guidelines provide design heuristics for system composition and thus form as part of the domain knowledge. We can represent these design heuristics as rules and as neurons (as in neural networks). We are currently investigating a best possible approach towards knowledge representation. FARE or Family Oriented Analysis and Requirements Engineering [11] is one of them. FARE mainly concentrates on producing requirements for a family of products. A check list is used to ensure SCV (Scope Commonality and Variability) analysis and cost-benefit analysis has been carried out. We are considering a similar rule or guidelines based method for knowledge representation.

**Asset Library Management**

A core asset is any piece of information that could be helpful in the production, evolution and maintenance of products in the product line. [4]. According to Brien and Smith [12], core assets have to be built from scratch, bought or contracted for or have to be mined from existing assets. When a product line matures, so does the number of assets. It becomes necessary at this point to employ some kind of a structured method that manages all these assets. This might prove highly helpful for the team of domain analysts who use these assets and the associated information in the future. Before venturing further into the subject, it must be duly noted that, the library’s scope would be limited only to that organization or product line from which it was built. In other words, it is specific to that particular domain and cannot be used or contracted as COTS.

We propose the construction of a Library of Assets in which each asset that enters the library has to be appropriately tagged and catalogued. Assets that undergo change have to be recorded and documented. We considered the similarity between design patterns and assets. Sommerville [13] has suggested the method of pattern name, description of problem area, solution description and statement of consequence and trade-offs for distinguishing each asset. Despite the fact that the concept of design pattern is completely different, overall it specifies how to manage a large number of assets which might be used in managing assets with some minor modifications.

Apart from numbering the asset and cataloging it, certain additional information for each asset be added for quick reference.

- The approximate time frame of when the asset was created.
- Asset name.
- If it is currently used by any family member and the description of the member.
- Any test results or test data available to test that asset (in case of high level language code).
- When this asset was last modified.
- Any tradeoffs that might have to be considered while using this asset.
- Any specific requirements that should be fulfilled in order to use the particular asset.
- Which requirements and use cases are linked to the asset.

Along with the asset specific information, the above questionnaire could also be used to gather
Each asset undergoes the process of encapsulation, where information about each asset accumulated. This package might include asset specific, standard information and an indexing system to identify a specific asset. Care is taken that these information are brief and clear. These capsules of core assets are then added to the collection or library of assets. The process of building a library is an evolving process and each and every new item that enters the library are tagged and any change made to old items is recorded.

**Services, Use Cases and Features**

One of our major aim is to link and map between various systems services to a specific product line features and then to a specific product use cases. Use cases view the process as actors and system. Actors initiate an activity inside the system. And the resulting activity takes several courses of paths until the operation is performed. Features on the other hand, represent the attributes that identify the end system. The feature that a system possesses is directly proportional to the requirements fed into the development process. We considered it worth while to study this pattern (Service-Feature-Use case pattern) based on view of product lines. For short we name this pattern as SFU pattern.

Though it is well known that requirements decide the features of the end product, there has not been any well known approach to clearly trace a particular requirement to a specific feature. We considered the idea of devising a mapping technique that clearly identifies the link between requirements and the resulting features.

We are currently involved in the process of building a browser, which takes features as input, and traces all possible requirements which are responsible for the existence of that particular feature. This tool would be highly specific for a particular domain in which the product line exists. It correlates with the knowledge base of the product line, and use case mapper to capture the link between a feature and a single/group of requirements.

During the development process of either conventional method based software or product line based, requirements gathering have always proved to be crucial, especially while developing family based systems. This method attempts to gather the requirements based on features even before the actual process of requirement gathering is established. But the knowledge base used in this tool, should be as detailed as possible, to generate requirements close to reality. But it should be noted that this can be initially used to verify the validity of the original requirements, since relying entirely on a tool for requirements might be risky until the knowledge base has matured to a certain level. In the following section we have illustrated a tool support to help to identify relationships amongst services, features, and use cases.

**Tool Support**

Tool is based on a set of functions that are integrated closely together as they support different aspects of our method. For example we are currently investigating XMI based development using meta-modeling technique to represent commonality and variability of use cases. Figure 3 shows the GUI of our
commonality analysis tool, which is used to compare the requirements of two products. The same tool can be used to analyze the requirements with the general product line.

Figure 3: IDEM Tool Support for creating, mapping, and retrieving SFU pattern

Figure 3 shows the working principle of IDEM. We have planned a web based tool, which is denoted by the intelligent browser, which gathers input in the form of name of the products or requirements or both. The analysis is performed based on the guidelines of FARE which is denoted by the product line knowledge base. This also represents the knowledge of the product, the list of all requirements within the product line. The use case mapper is our mechanism that does the actual process of mapping requirements of one product with another. The output is indicated by the generated list of requirements, use cases and charts.

Figure 4: Commonality Analysis with IDEM
Figure 4 shows the sample input screens for the tool where the products to be analyzed are selected. The analysis is based on a set of guidelines based on FARE. The aim of this approach is to deduce points to each requirement based on its commonality with the product line.

A group of checklists are used to calculate these points and these checklists themselves are a product of domain knowledge. Once derived, charts and graphs can be generated for comparison of products.

Table 1 shows the typical commonality comparison chart between two products. With a simple observation we can say that the product that covers more area has more commonality and is more suitable for SPL. And in our example the product Z3436 seems to exhibit more common features to the general product line as compared to Z3456. Our IDEM supports wide variety of outputs and charts for analysis and we are currently working on comparison of features and use cases as well.

**Conclusion**

This paper addressed the issue of product-line based development and has proposed a systematic method for achieving this. We have also proposed a tool that supports IDEM method. Our approach to product-line development is based on best practice guidelines where Software Engineers needs help to achieve better quality in short lead time. The proposed tool is currently under development.

**References**


ABSTRACT
State charts or machines are used to model the functionalities of software or its components. For a user interface, each control has several properties. A change in any control property causes a state change to the whole state machine. As a result, a simple application is represented by a large number of user interface states. This paper suggests an alternative definition or representation for a user interface state. The user interface model is transformed to a schema saved in an XML file. The schema structure represents the user interface state. This state considers only the structure of the user interface and ignores controls’ specific properties.

KEYWORDS
State machines, GUI states, user interface testing, and modeling.

1. INTRODUCTION
State machines are used to generate model-based tests. They describe state transitions of objects with states. Model checking is a technique used for verifying a system composed of concurrent finite-state machines. State machines should be finite as model checkers need to exhaustively explore the entire state space of the state machine.

In current applications, the ability to save and control the GUI state is very useful in several features such as: Undo, redo, animation, hide, show, enable, disable, etc. For example, a user wants to be able to undo an action or actions and remove their effect any where in the application.

Through using XML to store the GUI tree, we introduced a new definition for a GUI state. Rather than assuming that the GUI state depends on each property for each control in the whole application, we define the GUI state as the hierarchy that is embedded in the XML tree. A GUI state change here means a change in any parent-child relation, or any change in the specific properties that are parsed.

When using XML to store the GUI tree, we introduced a new definition for a GUI state. Rather than assuming that the GUI state depends on each property for each control in the whole application, we define the GUI state as the hierarchy that is embedded in the XML tree. A GUI state change here means a change in any parent-child relation, or any change in the specific properties that are parsed.

This new definition produces an effective reduction in GUI states. For example a small application like Notepad, can have more than 200 controls, each of which may have more than 40 properties, this produces 200 * 40 =8000 states. Any property or control from the 8000 controls and properties triggers the state change. In our assumption, 200 controls will have 200 parent-child relations (or less). Each control has less than 10 relevant parsed properties. The total possible GUI states
are reduced to 2000 or a 75% total reduction.

In testing, we usually use the changing of the user interface state as a reason to trigger or reapply several testing processes. For example, in GUI test automation, GUI scripts should be regenerated or revisited upon GUI state changes. As such activities can be expensive; we need to trigger them only when there is a good reason to do that.

Each GUI has many forms, each form has many controls, and each control has many properties. The general definition used in many literatures [1], [2], [3], [4], [5], [6], [7] that a change in any control property causes a GUI state change means that the number of states any GUI can have, is very large even for a small application.

2. RELATED WORK

An object state is the condition(s) of that object in a given time. A given state for an object defines the events that may affect it at this time, and the next possible states. A GUI state is described in terms of the specific objects or controls it currently contains, and the values of their properties [8, 9]. The information of a GUI state at any particular time is important for testing.

Saving the GUI to a GUI state file is investigated by several papers [10, 11]. The GUI state can be saved and retrieved from such files. This facilitates using some services such as undo, simulation, and testing.

In literature, usually, there is an ambiguity between the application and the user interface states. Although those two states are related, yet they are not identical. The application state is the state of all the application resources, including the user interface, at any given time.

Currently, there are many user interface description languages used to facilitate communicating with the interface, and the code behind it, from different software engineering tasks such as: requirement, design and testing. We will use XAML as an example.

XAML (eXtensible Application Markup Language; pronounced "Zammel") is a new Microsoft Longhorn declarative markup programming language for building applications’ user interfaces. Elements in the XAML file are correlated to the GUI objects at run time. XAML utilizes XML hierarchical logic to present the hierarchy of GUI objects. This makes our testing framework matches in principles the approach XAML is taking. It is possible to develop a solution with XAML without developing any code or develop partial XAML/code applications.

The ultimate goal of XAML is to have a standard syntax for describing user interface controls and eventually serializing all GUI the XAML files. This will be very useful in several ways. In one particular advantage, the UI implementation will be documented in a way that can be easily accessed, edited, and reconstructed. Some other advantages expected from having a standard syntax for UI design and implementation is; the ease of modifying the UI even at run time, the ability to separate UI components from the other layers of the application or separate the UI model from its view, and the ability to reuse the UI or some of its components.

XAML can be used as a UI modeling language to create UI elements that can be implemented in any platform and with any programming language (at least in principle). XAML simplified control properties into its type and text
only. The created button has default visual presentation through theme styles, and default behaviors through its class design. For testing, this reduces the large amount of possible UI states previously we were getting from the fact that each control has tens of properties.

A control can be presented in XAML using the line;

```
<Panel1> <Button Content="OK"/> </Panel1>
```

where the panel, panel1 and the button OK are two XAML elements.

XAML elements (i.e. UI controls) are mapped to .NET types that can be extracted from their assembly. Abstract classes are not mapped to XAML tags.

Avalon; Windows Presentation Foundation layer (WPF) is WinFX (i.e. .NET framework 3.0) user interface framework or graphics subsystem platform for Windows client’s applications. It is preinstalled in MS Windows Vista operating system. In WPF, control’s logic is separated from its appearance that adds flexibility to the way controls can be displayed. WPF content can be hosted in a Win32 window and visa-versa.

3. GOALS AND APPROACHES

The goal for studying the state of an object at any given time for testing is to know the “scope” of that object, i.e., to know the current actions that may affect the object and the results of those actions. This is important in particular, for transaction processing applications. For example, if a car is in an initial complete “off” state, some actions such as “switch on” will be available, while others, such as “accelerate” will not.

Should changing the color of one control in the whole GUI change the whole state of the GUI?! In other words, will such action disable some events and enable some others?! Maybe it is not accurate to say that in all cases, such change will not have any impact on the overall GUI state, but for the most cases, it will not. To deal with the problem of having large number of GUI states, we have to consider the major ones only.

The application can display the GUI hierarchy, its controls, and properties. Testers can then specify the properties that they want them to trigger a GUI state change as this can be different with the different scenarios.

The tool that we developed, as part of a GUI test automation framework [12, 13], checks for GUI state changes through comparing the XML tree file with the previous one (or any another selected one). We can allow certain processes or activities (e.g. regression testing) to be triggered once a GUI state change occurs.

The automatic comparison and verification of the overall GUI state (i.e. GUI structure) is not intended to be used for test case generation. It can be used to trigger the execution of regression testing in the same way a sensor triggers switching on, or off, an air condition system.

4. CONCLUSION AND FUTURE WORK

In this paper we suggested a solution to deal with the large number of the possible states for a specific application’s user interface. This number will affect the space of creating test cases that will cover the application (for test adequacy or coverage). We suggested using XML trees to represent the states rather than having all possible combinations of properties. We will apply the suggested track in several projects and compare the coverage of the created test cases in order to be
compared with those created from other techniques.

5. REFERENCES


Use of Data Recharging for Personal Information Environment

Atif Farid Mohammad
School of Computing, Queens University, Kingston, ON Canada
atif@cs.queensu.ca

Abstract- In this paper we provide a new way of defining personal information environment and its framework using data recharging. Using this new approach, human intervention can significantly be reduced while performing data creation and/or alternation update propagation. Furthermore, all these things can be done in a synchronized and secured way in user’s own domain called as personal information environment. Experimental results show that the provided algorithm works correctly and can be adapted/extended to other directions associated to an individual user’s personal information environment.

Keywords: Data Recharging, Personal Information Environment, Latency, Data Replication

1 Introduction

We live in a world full of electronic gadgets and many computing devices are associated directly with our lives. Mark Weiser, one of the first visionaries of pervasive computing said in 1991 that we are heading to a “state where computing devices are so pervasive and critical to our activities that they are taken for granted and effectively disappear into the background” [1]. This statement describes the notion of “Pervasive/Ubiquitous computing”, which is the next generation of computing where individuals have access to their data anywhere and anytime.

Pervasive computing embeds computing devices both large and small in our daily life. The challenge is to facilitate the integration of these devices into an environment that enhances a user’s daily life while making the integration easier to manage or even invisible to the user. Users commonly employ multiple devices in their daily lives, such as office desktop, laptop, PDA, cell phones etc. They have data objects on all these devices. The union of these objects is the user’s Personal Information Environment.

Personal Information Environment is a relatively new field in pervasive computing. This environment contains the pervasive information associated within a certain individual’s context. An important question arises here, “what is a context?” A context is an interrelated condition in which something exists. Context should be understood as a function of interaction between users/objects and environment, as well as a consequence of focus or attention [2]. The human mind uses context to manage a massive amount of information from a myriad of different situations, such as work, family, friends, and community. Using context, people quickly can interpret what information is relevant in a given situation. As people, we can recognize the contexts we are in, know what information is applicable to each context, and derive information from each context. The information derived is needed to be stored and further processed related to an individual in a Personal Information Environment, which is related to an individual’s data processing and its replica management [3]. A Personal Information Environment contains several domains of an individual’s life, such as:

- Personal info, such as phone, e-mail directory and physical addresses of contacts,
- Financial data: e.g., bi-weekly earning, daily spending record, medical spending, travel and leisure spending record, etc.

In this paper we examine the concept of Personal Information Environment (PIE) based upon the idea of having information available “anywhere, anytime and anyhow” [1]. Our experiment shows that the provided algorithm can efficiently help reduce human intervention to perform PIE activities in a synchronized and secured way

2 Personal Information Environment

In a PIE an individual user is to utilize all resources of PIE-associated devices. Coda file system [8] was the first after Andrew File System (AFS) [9] to introduce consistent availability of data in case of intermittent connectivity of devices due to any data communication service disruptions. Satyanarayanan and his peers name it Isolation-Only transactions
(IOTs) in Coda. A transaction is a unit of work that is treated in a consistent and reliable mode to operate independent of other transactions. Frangipani [10] is another distributed file system that manages a collection of disks on multiple machines as a single shared pool of storage.

Figure 1: Devices and data storage and its I/O operations relationship diagram.

It is a fact that the amount of information in our surroundings is continuously increasing. To make sure that an individual has all relevant information is getting more time consuming. To capture the required information quite a few computing devices are being utilized. The pictorial representation given below is another way to demonstrate the utilization of various devices in our daily lives.

Every device shown in the Figure 1 is a pervasive smart device in an individual’s Personal Information Environment. The availability of the mentioned devices needs charged batteries as an important factor to be used by a user. A user needs two major types of recharging of available resources, both power and data. First, power recharging is needed to make sure that the devices work. Second, data recharging is needed in today’s high-tech world. The devices use rechargeable batteries for the power and can be recharged when needed from a power grid. An analogous process can apply data and recharging from a specific data grid or the information grid (i.e. the Internet) [4]. Everyday data processing in a PIE requires data transmission within PIE-associated devices. In order to reduce data access latency [7], data replication is required [4]. The data replication process constructs an environment that permits several copies of the same data to be distributed to various devices in the same environment. Efficient use of data replication helps in improving reliability by creating multiple replicas of the data on related devices. A replication [15] scheme ensures consistency in the replicas. There are two basic types of replication: synchronous replication and distributed/asynchronous replication. Both of these synchronous and asynchronous strategies can be used depending upon the environment’s needs. The effective and efficient use of replication is the key to making Personal Information Environment data always available for the users regardless of their mobility and changes to their contextual domains.

Data recharging is facilitated by the use of data replication. Wolski et al [5] describe this process as the maintenance of data in an up-to-date fashion in all PIE-associated devices, when one of these runs out of fresh data, needs to be recharged, similar to battery power. They also stated about the complexities involved in data recharging, such as data replication, “timely synchronization and data reconciliation in the cases of conflicting operations.”

The proposed prototype solution in this research work achieves effective data recharging in a PIE using a FIFO cache scheme as well as in replica transmission process. Computers include caches at several levels of operation including cache memory and a disk cache. This cache [6] or content distribution management allows prompt availability of data that would otherwise take longer periods of time and network bandwidth to access from PIE-associated devices. It involves keeping a copy of objects that are expensive to construct after the immediate need for the object is over. The object may be expensive to construct for any number of reasons, such as requiring a lengthy computation or being fetched from a database.

3 Data Recharging

All data transmission/broadcasting and communication can be categorized as types of data recharging using data replication. As per [16] data replication can improve data accessibility and system performance as a promising technique. In an application environment data recharging plays a crucial role, any changes of the master data needs to be replicated to all nodes containing the replicas in exact. Master data holder device can recharge data from the application server that needs to have its data replicated to its associated sub nodes. Recharging utility can support all forms of snapshots [13] /replicas to be replicated [14], while maintaining data consistency.

Data recharging can allow very rapid restoration/updating of replicas using data replication to a relatively recent point in time upon the connection establishment between primary and sub nodes. This scenario can be understood as a power charging of rechargeable batteries connection to a power outlet. Ideas presented in this study are based on previous work done on data transmission and broadcasting.
4 Related works with issues and resolution

The most recent work done in data recharging is Project 54 [11, 12]. It is the pervasive system designed to create an organizational pervasive computing environment within the police cruiser where officers can get and update information and interact with their computers both within their vehicles and the servers at the office.

There is a class of commercial products available for data transmission with advantages and disadvantages; PIE prototype resolves these disadvantages on no cost to an individual user. Some of those are presented here for a brief understanding of middleware data communication/recharging solutions.

Avail Wide Area File Services (WAFS) [17] software allows branch office users to access and share files over the WAN at LAN speeds. Its real-time multi-directional acceleration and mirroring technology ensures that the same data exists on all servers regardless of where changes occur. An individual’s data is to be propagated using the same network can be a potential data security issue, which is resolved in PIE prototype. Its cost is USD 495.00.

PowerSync 5.0 WS [18] is the workstation version of the industry leading tool for backup and synchronization of files across local and wide area networks. This solution can operate in MS Windows only. PIE prototype has no restriction of operating system.

Avvenu [19] Access and Share: Nokia has introduced this software to share within their user domain to access and share your files within your service along with the phone service to get your files available on a Nokia cell phone as well. It contains two services, first one is free to remote access your files, second is a search facility of your files, third is to play music from any of your computers. You have some limitations as you cannot use more than 10 devices. Second service is to be bought by the users and cost is between USD 9.99 to USD 79.99. The issues with this facility are as follows:

- User has to open an account with Avvenu. There is no need to open any accounts anywhere in PIE.
- User has to have a connection with Nokia service provider to access files on his cell phone. There is no need of any such connection in PIE, as PIE prototype works on any WIFI.
- User has no control on his privacy, even it is been provided by the service provider, user still has no control, where his data is being routed from. PIE prototype is to encrypt data during transmission phase for data security.
- Avvenu can have some maintenance issues and if there servers are down due to any reason, user has no access to his/her data for that specific time frame. In a PIE if one server goes down second server takes over as all the desktops and laptops can work as a server upon requirement.

5 Data Recharging Algorithm

The designed solution is to achieve the rolling data recharging related to a Personal Information Environment. There are a few separate folders to be considered in our example; such as a “work” folder or a “home” folder. Some computing devices may be set to automatically pull or push data in all or some of these folders. Let these folders be denoted L1, L2, ..., Lm, where “m” is the total number of folders. As every folder contains several files with various data sizes, we define the function N(Li) is the “number of files in List i”. It is to supply the number of files in folder Li. Individual files are denoted by Li(x), where x ∈ {1,2,…,N(Li)), which denotes the xth file in the ith list. Or we can say that x denotes a particular file belonging to the set of files in Li (1<= x <= N(Li)) rather than just being a numerical index.

For each file, let us define the functions P(Li(x)) denoting the “Path name”, S(Li(x)) denoting the “Size” in bytes and TS(Li(x)) denoting the “Time stamp” [Time] (last time the file was modified).

There are two classes of computing devices in the system: machines (Desktop computers, Laptop notebooks) and portables (PDAs, iPaqs, Blackberries etc.). Machines are denoted m1,...,mL for the n machines and portables are denoted p1,...,pL for y portables. The internet server is denoted by “I”. Machines have different behavior from portables when it comes to files.

For example, if Machine 2 is set to download Folders 1 and 3, then this is denoted L(m2) = {1,3}. However, portables have limited space, so they will only download a few specific files that the user of the system has requested from the list display. From a portable, a user may login and browse from all the files in the system in all lists available at internet server “I”,...
which is the index file of all individual folders, and then download what they want. If Portable 1 is set to download the files “poetry_podcast.mp3” in List 1, “Resume.doc” in List 2 and “todo.txt” in List 3, then we use the notation given below:

\[ L(p_1) = \{L_1(“poetry_podcast.mp3”), \ L_2(“Resume.doc”), \ L_3(“todo.txt”)\}. \]

As this PIE solution is in its initial stages, the transmission speed is of great concern as per the performance perspective. Let us use \( B(u,v) \) to denote “transmission speed from \( u \) to \( v \)”, in bytes/second, where \( u \) and \( v \) are either machines, portables or the internet server. Note that \( B(u,v) \) is not necessarily equal to \( B(v,u) \). For example, \( B(m_2,I) \) denotes “transmission speed from Machine 2 to the internet server”. Total transfer time may be expressed in the light of predefined functions as

\[ T(L_i(x), u, v) = \frac{S(L_i(x))}{B(u,v)} \quad [\text{Seconds}], \]

where total transfer time is the transmission time for file \( x \) in list \( i \) from \( u \) to \( v \).

6 Data Exchange Operations

There are two data exchange operations pushing and pulling are described below in the context of the algorithm design phase.

- A “push” is where a machine is to encrypt data for security and uploads the user’s data onto the internet server.
- A “pull” is where data is downloaded from the internet server to a machine or portable and decrypted for any processing needs.

In case a machine associated with a Personal Information Environment is switched off for any reason, as soon as it restarts the PIE prototype solution will restart and will bring in all the changed files (if any) from the server so as to be equivalent with all related peers. The pseudocode for the data recharging solution is as follows:

A. At every interval on a machine (i)

1) Take a census of all shared files on machine i.
2) Download all file information from the internet server for the folders in \( L(m_i) \). The \( i^{th} \) downloaded list is denoted as \( DL_i \) so as to be distinct from \( L_i \).
3) Execute the following loops:
   For each file in both lists, compare

\[ \text{If } TS(L_i(x)) > TS(DL_i(x)) \Rightarrow \text{Start: Secure data: Push(L}_i(x)) \]
\[ \text{If } TS(L_i(x)) < TS(DL_i(x)) \Rightarrow \text{Start: Pull(DL}_i(x)) \text{: Decrypt data} \]

B. Every interval on a portable (i)

1) Download all file information from the internet server for the files in \( L(p_i) \).
2) Execute the following loop:
   For each file in \( L(p_i) \):
   \[ \text{If } TS(L_i(x)) > TS(DL_i(x)) \Rightarrow \text{Start: Secure data: Push(L}_i(x)) \]
The last step mentioned above will only occur on the user’s initiative, and is dependent on the availability of both enough memory and storage in the user’s handheld device. It is a fact that smaller devices, such as PDAs, Blackberries etc., are clearly a resource–constrained in memory and storage when compared to Laptop or desktop computers. This suggests a PIE prototype solution which minimizes the use of communication bandwidth and memory usage.

The PIE prototype solution for a handheld device allows a user to download needed data files for both ‘read’ and/or ‘write’ purposes. In a case when a ‘write’ transaction happens on the data, changing the time stamp this data is pushed back to other devices by the PIE prototype solution to make all replicas consistent. This process utilizes the communication bandwidth only in either a download or upload scenario and minimizes the use of bandwidth. It can have a drawback of the risk of leaving a stale replica on the PDA for potentially a long time until this is synchronized with a computer or laptop to get this replica transferred in the unavailability of a wireless network connection.

7 PIE Prototype Operations Analysis

Operations analysis helps a PIE prototype solution’s user to accomplish the objectives of a PIE
and its related data by bringing a systematic and restricted approach of data transmission in a closed PIE domain. This is a single-user Personal Information Environment prototype solution, where several simplifying factors can be expected:

1) The user will only be working on one machine at a time, and will usually be doing so for an extended period.
2) The user will only be working on few files at any given time.
3) The user is expected to close the files, when done working on them.

This means the use of the system will be much simpler than a multi-user, multi-file processing environment. An interesting question can be asked by a PIE user: what can I expect the usage of the system to look like? If a user is typing only text into a Word document, then we can expect or assume the size of the file to increase by roughly one kb per minute. To distribute this out to every computer in a PIE prototype system would not take up any significant bandwidth. The other extreme would be if a user is video-editing or music compiling. In a nutshell, it is assumed the master data update originating device transmits the updates to all other replicas to replace data with an older timestamp after the master data update is committed. The data index list is compared after a predefined time interval to assure that all the replicas converge to one data file.

The system was analyzed under some simple models, and the results are displayed graphically. For simulation purpose it is assumed that the system updates both sends and receives data once a second.

First, latency was modeled under different numbers of machines and average data upload/creation rates. Let’s define these terms:

- **Latency** – How far behind, in seconds, the system is from being perfectly up-to-date throughout. A latency of zero seconds means the system is perfectly up-to-date.
- **Data Upload/Creation Rate** – The user is creating or updating new files, which must be uploaded to the internet server and sent out to each machine. Obviously, this will never be consistent. Data upload/creation rate is here approximated as a simple “x kb/s”, which represents the expected value of the random variable which is the amount of new files uploaded/created on each second basis.

In the graph below shown in chart 1, latency is computed as in the following equation:

\[
\text{Latency} = R \left( \frac{1}{B(m_i, IS)} + \frac{n - 1}{\min \left( \frac{B(IS)}{n}, B(IS, m_i) \right)} \right)
\]

Where R is the expected value of the upload/creation rate

\(B(m_i, IS)\) and \(B(IS, m_i)\)

are assumed to be 400 kb/s, and the user is working at the \(k^{th}\) machine. \(B(IS)\) is the total bandwidth available to the Internet Server, which is split among all the machines it sends data to. This is assumed to be 1000 kb/s.

![Chart 1](image)

The most important interpretation of this graph is that when latency is greater than 1 second, it means that, every second, it takes the system more than 1 second to catch up, so the system is constantly falling behind. This should be rare in practice.

Second, pulling was modeled for an individual machine. A more realistic model was sought here for file creation/upload. Instead of an expected value, an exponential probability distribution function was used. The idea behind this is that small updates are very common, whereas large updates happen, but they can be rare. The probability density function for an exponential distribution is:

\[
f(x; \lambda, A) = \lambda e^{-(\lambda/A)x}, \quad x \geq 0
\]

And the expected value of the random variable \(X\) with an exponential distribution is \(A/\lambda\), where \(\lambda\) is called the rate parameter and A is a scaling constant. We assume
that $\lambda = 1$ is an acceptable rate parameter, and this requires $A = 50$ for an assumed expected value of 50 kb/s. Then, the probability density function looks like the following chart 2:

![Chart 2](image)

To generate random values with this distribution, the following equation was used:

$$X = -\ln(p) \frac{A}{\lambda}$$

Where $p$ is a uniform random variable on [0,1]. A sample of some random upload/creation of data rates in [kb/s] is shown below:

![Chart 3](chart3)

```
107.2811034
18.67197596
47.49730736
1.604099243
163.6371854
```

Let us denote these values as $R_t$ for time $t$ is the progress of downloading a file, and let us denote the latency as $L_t$ in kb at time $t$. So, in the simulation, our update step will look like:

$$L_{t+1} = \max(L_t - 1,0) + \frac{R_t}{B(IS,m_k)}$$

The purpose of the mathematical phrase $\max(L_t - 1,0)$ is to carry over latency from the previous iteration. Under simulation, the results are graphed below in chart 4:

![Chart 4](chart4)

The line below starting from 0 represents the latency experienced by single machine that is pulling. The upper line represents the latency that the same machine experienced if it was reconnected to the system after a long time off, and was 1.5 MB “behind in data downloading process”. Note that this is rectified fairly fast.

Third, a simulation can be performed for how the portable’s latency can affect data transmission. For a portable, the bandwidth is comparatively small, and also relatively erratic as WIFI connections are utilized for data transmission. Bandwidth needed to be modeled in a manner that very roughly mimics cellular network behavior. This is characterized by:

- Varying regions of signal strength
- If the signal is strong (weak) and thus, the bandwidth is high (low) it is likely to remain that way.

### 8 Future Work

This algorithm can work for any devices attached to our environment as pervasive devices. This research work can further be evaluated for health care industry as well to monitor several data associated with a patient in care facilities or at their home. Palliative care can be a good example. A patient in palliative care needs several pervasive devices to reduce or remove pain to survive in better condition. Data associated with these devices need to be transmitted via a certain computing device from patient’s care facility on ASAP basis to the medical authorities. These hospice staff is to make decisions about patient’s care in the least most time period to provide advice to the attending staff to take actions to provide relief to the patient. This data transmission is possible with our research work on pervasive data management and recharging.


9 Conclusion

Given the ubiquity of wired and wireless high speed connections, there are no significant barriers to the performance of this solution. Data recharging in a personal information environment is a phenomenon of seamlessly transmitting data between $n$ nodes. This research work brings in a novel approach of reducing human intervention to do data updates and propagation in a synchronized and secured way. The rolling data recharging solution for the PIE is based on lazy replication. Intermittent connectivity of the information grid (Internet) is a main reason to use a good mix of the approaches to transmit data asynchronously. The research objective of this rolling data recharging in a personal information environment is to improve a PIE users’ productivity by providing data access at anytime, anywhere and with consistent data transmission performance. Optimistic replication is an eminent method to achieve such objective but its practicality is strongly based on the fundamental consistency protocol to guarantee swift and dependable replica consistency. The contribution of this paper is n-fold. There can be an $n$ number of devices attached to a user’s PIE. A user can utilize data can any nature and data recharging in a PIE-settings will remove all obstacles of data transmission and replica updates for all required files of any structure. PIE data recharging structure facilitates efficient, snapshot-consistent data availability for any processing needs.

Acknowledgments

I like to thank Dr. Pat Martin of Queens University to show me the path to conduct research and produce this work on Personal Information Environment.

References

Software Measurement Needs Its Own Theory

Kehinde O. Jolayemi¹ and Austin Melton²
¹Department of Computer Science, Kent State University, Kent, Ohio 44242, USA
²Depts of Computer Science and Mathematical Sciences, Kent State University, Kent, Ohio 44242, USA

Abstract - Measurement is critical in scientific and engineering disciplines, including computer science. Though it is relatively easy to define algorithms which assign numbers to software artifacts and processes, it is not so easy to establish rules which allow us to determine whether or not a given algorithm should or should not be considered an acceptable software metric. For approximately twenty years, software measurement researchers have been using classical measurement theory as a basis for software measurement. However, to use classical measurement theory as a basis for software measurement, we should be able to establish that the assumptions upon which the classical measurement theory is based are satisfied by our software metrics. In this paper, we show under generally accepted properties for software metrics that software measurement is not measurement in the classical sense. Thus, there is a need to establish a theory for software measurement.

Keywords: Measurement theory, software engineering, software metrics.

1 Introduction

Software measurement probably began when people began counting lines of code. The field of software measurement, first called software science, began in the early 1970s with Maurice Halstead. For a summary of Halstead's work and some pre-software science related work, see Linda Ott's “The Early Days of Software Metrics: Looking Back After 20 Years” [6].

Though software measurement is a form of measurement, the software engineering community was slow to look to measurement theory for help or guidance; it was not until the early 1990s that researchers began to incorporate ideas of measurement theory into their work [3]. Today, however, measurement theory is commonly used and referenced in software measurement. See, for example, the often referenced and highly regarded text by Norman Fenton and Shari Lawrence Pfleeger [4]. It is, of course, good that software engineers are using results from measurement theory to try to improve the quality of software measurement. However, there is an important question which needs to be addressed. The question is whether measurement theory, which was developed for use in the physical sciences, is an appropriate theory of software measurement.

Of course, one might ask whether it makes any difference, i.e., is it important, to have a theory for software measurement. We will be able to answer this question with a clear “Yes” after we review the representation theory of measurement. (Please see Remark 2.3.) Also, we will show that software measurement is significantly different from classical measurement that there is a need to try to define a measurement theory for software measurement.

In the next section, we give a short introduction to classical measurement theory according to Campbell and to the representation theory of measurement. After these introductions, we look at Elaine Weyuker's paper “Evaluating Software Complexity Measures” [8], and based on her paper, we show that software measurement is not measurement in the classical sense. Further, we give another reason why software measurement is different from classical measurement. There are entity sets in software engineering with natural orders which are not linear.

Before moving into the main body of this paper, we note additional support for developing a measurement theory for software measurement. Briand, El Eman, and Morasca have an interesting take on the use of measurement theory in software measurement [1]. They claim that strict adherence to measurement theory principles should not be required in software measurement because strict adherence to measurement theory principles would limit the metrics which could be used. Thus, they are clearly saying that classical measurement theory is not appropriate for software measurement. Their advice might be right on target if it were not important to have a guiding measurement theory so that we know how to use our software metrics. (See Remark 2.3.) The recent paper by P. N. Gegela, P. Kosaraju, and A. Melton also makes a case for developing a measurement theory for software measurement.
2 Measurement theory

2.1 Measurement theory according to Campbell

We present a short summary of measurement theory as developed by Norman Campbell [2]. Our summary is based on the monograph by Thomas Reese [7]. It is appropriate to reference Reese's monograph because the purpose of the monograph was to examine whether measurement in psychology was measurement in the classical sense. Interestingly, the main point of contention in the discussion reported by Reese was the question of additivity in psychological measurement. Additivity will also be a major concern for us.

Campbell said measurement is an assignment of numbers or symbols to entities according to the `amount" of an attribute that the entities have. Campbell's first law of measurement says a metric should measure relative amounts of an attribute so that if two entities have different amounts of an attribute, then the metric should assign a number or symbol with greater value to the entity with the greater amount of the attribute. According to Campbell, a metric which satisfies this first law is an ordinal metric.

The second law requires additivity, and metrics which satisfy this law are called additive or extensive. The additive principle requires a binary operation to be defined on the set of entities. Further, the metric needs to "respect this binary operation" as described below. E denotes our set of entities; □ denotes the binary operation on E; V denotes the set of measurement values; and m denotes the metric. For us V is a subset of the non-negative real numbers.

In order to be additive, a metric must meet the following conditions where a, b, c, d, e, f, and g are elements in E.

If m(a) = m(b) and m(c) > 0, then m(a □ c) > m(b).

If m(a □ b) = m(g), then m(b □ a) = m(g).

If m(a) = m(b) and m(c) = m(d), then m(a □ c) = m(b □ d).

If m(a) = m(b), m(c) = m(d), and m(e) = m(f), then m((a □ c) □ e) = m(b □ (d □ f)).

2.2 Representational theory of measurement

We give a short introduction to the representation theory of measurement. For a more complete introduction, see [3]. Our short introduction is based on [3], and in fact, our shopping and joke examples and our description of the five basic representation scales are borrowed from [3].

According to representation theory, measurement is to parallel empirical observations in that the measurement or metric "answers" are usually numbers for which relationships among the numbers match empirical relationships among the entities being measured. This matching of relations on the entities and the numbers allows for a firm mathematical foundation for measurement. Our current emphasis is on a more intuitive understanding of the representation theory of measurement. We give a couple examples and definitions of the different types, called scales, of metrics according to representation theory.

Example 2.1 Assume we want to measure people by the type of store where they most often shop. This type of measurement is classification. We use three types of stores: grocery, hardware, and clothing. We encode our classifications with 1 for grocery, 2 for hardware, and 3 for clothing. In this example, there is no natural relationship between people who normally shop in one kind of store compared to another kind of store. Thus, there is no significance in clothing shoppers having a 3 compared to grocery shoppers having a 1.

Example 2.2 Assume we want to measure people based on their responses to a joke. We consider five responses: weeping, frowning, no observable response, smiling, and laughing. We encode these responses with 1 for weeping, 2 for frowning, 3 for no observable response, 4 for smiling, and 5 for laughing. For this example, there is a relationship among the responses in that frowning is more desirable than weeping, that no facial response is better than frowning, that smiling is better than no response, and that laughing is better than smiling in response to a joke. Thus, the larger the number is, the better the response is. However, it is worth noting that it would have been just as appropriate to encode our responses by 1 for weeping, 20 for frowning, 21 for no observable response, 300 for smiling, and 327 for laughing. The only requirement for the numbers is that the order should correspond to the "ordering" of the responses.

There are five well known scale types. They are nominal, ordinal, interval, ratio, and absolute. These types may be characterized by the allowable types of answer transformations.

For the nominal measures the admissible transformations are one-to-one functions. Examples of nominal measures are labels or classifications as in the shopping example.

For the ordinal measures the admissible transformations are strictly order-preserving functions. F: X → Y is a strictly order-preserving function if whenever x < x' then F(x) < F(x'). Examples of ordinal measures are preference and quality measures such as the joke example.
For interval measures the admissible transformations are linear functions of the form $F: X \rightarrow Y$ where $F(x) = ax + b$ and $a$ and $b$ are constants with $a > 0$. Examples of interval measures are temperatures such as Fahrenheit and Celsius.

For ratio measures the admissible transformations are linear functions of the form $F: X \rightarrow Y$ where $F(x) = ax$ and $a > 0$ is a constant. Examples of ratio measures are length and absolute temperature.

For absolute measures the only admissible transformations are the identity functions. Examples of absolute measures are counting measures.

Remark 2.3 How does knowing representation theory help us answer the question why is it important to have a measurement theory for software measurement? Consider the shopping example. Given the measurements of grocery shopping is 1, hardware shopping is 2, and clothing shopping is 3, one might be tempted to say that hardware shopping is the mean or average of grocery and clothing shopping. Of course, in this example, this “mean” is laughable. However, when we are trying to measure something which we do not understand well, we might be tempted to determine a “mean” under circumstances which would be equally laughable. In terms of representation theory, the problem with trying to take an average in the shopping example is not that the example is shopping; the problem is that our metric is only a classification or nominal metric, not an ordinal or ratio metric. We need to have a theory for software measurement so that we can know the significance of our metrics and so that we can use them meaningfully.

3 Software metrics

One of the problems with discussing a foundation for software measurement and metrics is finding a starting point. We use Elaine Weyuker's paper “Evaluating Software Complexity Measures” [8] as a starting point. We show if we accept Weyuker's properties as valid and accurate software complexity metric properties, then software measurement, at least as determined by software complexity metrics, is not measurement in the sense of classical measurement.

In [8], upper case Latin letters $P$, $Q$, and $R$ are used for programs. We will still use $E$ for the set of entities; thus, $P$, $Q$, $R$, $S$ are elements in $E$. Also, we continue to use $m$ for the metric. Weyuker uses “;” for the binary operation on programs, and we will use “□” in place of “;”. Using these conventions, Properties 6a and 6b in [8] are:

6a: There exist programs $P$, $Q$, and $R$ such that $m(P) = m(Q)$ but $m(P;R) \neq m(Q;R)$.

6b: There exist programs $P$, $Q$, and $R$ such that $m(P) = m(Q)$ but $m(R;P) \neq m(R;Q)$.

Software complexity metrics which satisfy Weyuker's Properties 6a and 6b can not be metrics in the classical sense of measurement because they are not additive. Weyuker writes: “Although a given program body $R$ has a fixed complexity in isolation, $R$ may not interact at all with a program body $P$ with which it is concatenated, while $R$ might interact with $Q$ in subtle and important ways which affect the complexity of the resulting program body.” (page 1361 of [8])

This “interacting” activity may be present in classical measurement theory, but there is a significant difference in what is allowed in classical measurement theory and what Weyuker is describing. It should be noted that the interacting as described by Weyuker is well accepted among software engineers.

Campbell's properties allow for interactions when entities $a$ and $c$, for example, are combined by the binary operation. The possibility of interacting is why we can not simply say that $m(a □ c) = m(a) + m(c)$. However, the interaction is completely determined by the amounts of the attributes in $a$ and $c$ which is why we have property which says

$$m(a □ c) = m(b □ d).$$

From this property, we see that the interactions are equal when the amounts of the attribute in the combined entities are equal. However, the interacting described by Weyuker is not determined by the amounts of the attributes in $P$, $Q$, and $R$. That is why we can have $m(P; R) \neq m(Q; R)$. We can get different interactions from programs with the equal complexities.

4 Non-linear orderings for entity sets

There is an additional phenomenon that occurs in software engineering and that seems to make software measurement different from measurement in the classical sense. Let's consider directed flowgraphs comparable to what are used in defining the McCabe metric. (We are purposefully keeping our flowgraph definition simple.)

Our directed graphs will have four types of nodes. They are a begin node, denoted by $b$; an end node, denoted by $e$; test nodes, denoted by $t$; and statement nodes, denoted by $s$. The begin node has in-degree 0 and out-degree 1. The end node has out-degree 0. Each test node has out-degree 2, and each statement node has out-degree 1.

We begin with four simple graphs. Our first basic graph represents the program with one statement. This graph is denoted by $G-s$, and $G-s = (\{b, s, e\}, \{(b, s), (s, e)\})$. The second basic graph, denoted by $G-ss$, consists of two sequential statements. $G-ss = (\{b, s-1, s-2, e\}, \{(b, s-1), (s-1, s-2), (s-2, e)\})$. The third basic graph is the if-then-else graph and is denoted by $G-if$. $G-if = (\{b, t, s-1, s-2, e\}, \{(b, t), (t, s-1), (t, s-2), (s-1, e), (s-2, e)\})$. The fourth and final basic
graph is the do-while graph and is denoted by G-do. G-do =

\{(b, t, s, e), \{(b, t), (t, s), (s, t), (t, e)\}\}.

We recursively define our set of flowgraphs by beginning
with the four basic graphs as flowgraphs and by saying in any
flowgraph a statement node may be replaced by a flowgraph
with the beginning and end nodes removed. For example, if
we replace the statement node in G-do with the if-then-else
flowgraph, we can get a new flowgraph; let’s call it G’. G’ =

\{(b, t, t-1, s-1, s-2, e), \{(b, t), (t, t-1), (t-1, s-1), (t-1, s-2),
(s-1, t) (s-2, t), (t, e)\}\}. Further, the only graphs which we will
consider to be flowgraphs are those which we can recursively
constructed as described here.

We define a natural order ≤ on the set of flowgraphs as
follows. For each flowgraph G, G-s ≤ G. Also, if G-1 and
G-2 are flowgraphs, then G-1 ≤ G-2 if and only if beginning
with G-1, we can construct G-2 in the manner in which G’ was
constructed from G-do, and hence, G-do ≤ G’. This is a
natural order on flowgraphs, and it is one which could be of
use in studying control flow and complexity issues. Further, it
is a partial order which is not a linear order. The graphs G-do
and G-if are not comparable.2

5 Conclusions

Some of the measurement activities in software
measurement are not measurement in the classical sense, and
we need to develop a measurement theory which can be used
as the basis for software measurement.

6 References

application of measurement theory in software engineering”,
Research Network, technical report #ISERN-95-04.

of Measurement and Calculation, Longmans, Green and Co.

software measurement”, Software Measurement (A. Melton,
pp. 27-38

[4] Norman E. Fenton and Shari Lawrence Pfleeger, Software
Metrics: A Rigorous and Practical Approach, PWS Publishing
Printing.

theory principles in software measurement”, Proceedings
of the 16th International Conference on Software Engineering
and Data Mining (Las Vegas, Nevada), 2007, pp. 344-348.

[6] Linda M. Ott, “The early days of software metrics:
Looking back after 20 years”, Software Measurement (A.
Melton, ed.), International Thomson Computer Press, London,
1996, pp. 7-25.

of physical measurement to the measurement of psychological
magnitudes, with three experimental examples”, Psychological
Monographs, The American Psychological Association,
Evanston, IL, Whole No. 251, 55, 1943, no. 3, pp. ii-89, The
American Psychological Association, Evanston, IL.

measures”, IEEE Transactions on Software Engineering, 14,
1988, no. 9, pp. 1357-1365.

---

2 The idea that software measurement might have naturally
occurring non-linear orders is not new. From the middle of
the 1980s through the middle of the 1990s, there was a metrics
research group called the Grubstake Group. The group
consisted of Albert Baker, James Bieman, Norman Fenton,
David Gustafson, Austin Melton, and Robin Whitty. During
the last couple years of the group’s active existence, Linda Ott
replaced Baker. On more than one occasion this group
discussed the possibility of non-linear orders, and in
particular, we talked about comparing G-do and G-if.
However, we never quite put all the pieces together to
establish in a natural way that G-do and G-if are not
comparable.