Object-Oriented Simulation Languages and Environments:

A Four-Level Architecture

John A. Miller∗†

Walter D. Potter∗†

Krys J. Kochut∗†

Deepa Ramesh∗

∗Department of Computer Science and

†Artificial Intelligence Center

415 Graduate Studies Research Center

University of Georgia

Athens, Georgia 30602-7404

USA

Phone: (706) 542-3440

Email: jam@cs.uga.edu

Keywords: Object-Oriented Simulation, Simulation Modeling Environments, Object-Oriented Databases, Integrated Information Systems, Repositories.

1. Introduction

Simulation modeling and analysis is a very complex undertaking. Its primary purpose is to provide useful and accurate information to designers and decision makers. As such it can be viewed as a special type of information system. Since the centerpiece of modern information systems are advanced Database Management Systems (DBMS's) such as Extended Relational DBMS's or Object-Oriented DBMS's, they should also play this role in simulation environments. A variety of other tools and systems are also vitally important in today's information systems (those for development as well as those for end-use). To provide an Integrated Information System (IIS), these tools and systems need to be kept track of, managed and glued together in a loosely-coupled fashion. As an emerging technology, meta-databases or repositories are beginning to be used for this purpose. In this chapter, we present a four-level architecture for loosely-coupled, database-centered simulation environments.

Recently, object-oriented programming techniques have been increasingly applied to simulation, using languages such as C++ [Stro91] and ModSim II [CACI90]. The benefits of utilizing this programming paradigm are being realized in the simulation area as they have been in the software industry in general. It is, however, somewhat misleading to imply that simulation is following this general trend in software development. In the mid 1960's a pioneering simulation language called Simula-67 [Birt73, Birt87] was developed. Motivated by the need to more faithfully model entities or objects in the real-world, the designers of Simula-67 introduced the core concepts that now form the object-oriented paradigm. Simula-67 also strongly influenced the designs of Smalltalk and C++, the two languages which are primarily responsible for the enormous popularity of object-oriented programming today. Therefore, it is only natural for object-oriented programming to be important in simulation, as that is where it originated.

In this chapter, we will present a Four-Level Architecture suitable for object-oriented simulation languages/environments. The goal of this architecture is to provide a framework in which a Query Driven Simulation (QDS) system can be built. Users see a QDS system as an integrated information system that is able to present the results of simulations in an understandable manner, and if adequate results are not available, it can execute simulation models to produce an adequate set of results [Mill89, Mill90, Pott90, Mill91a, Koch91, Mill91b, Mill91c, Pott92, Mill92, Pott93b]. Much of our previous work has focused on developing a tightly-coupled system based on using an object-oriented database system (Active KDL [Mill91a]) which includes a database programming language that is sufficiently powerful to express simulation models. (Note, Active KDL is an extension to KDL which was designed by Potter and Kerschberg [Pott86, Pott88, Pott89, Pott93a]).

We now feel that although the tightly-coupled approach is viable, it may be unwieldy. The trend today is to provide interoperable components that fit within a superstructure. Components should be able to work independently as well as in an integrated fashion. It should also be possible to replace components with superior ones (i.e., unplug an old component and plug in a new replacement). Consequently, the focus of this chapter will be on providing a framework suitable for a loosely-coupled QDS system. Object-oriented approaches provide the necessary foundation for this degree of modularity and reusability. Note, this is important in both developing a QDS system, as well as developing simulation models using a QDS system.

Foundational capabilities will be provided via C++, the most widely used object-oriented programming language. The loosely-coupled system that we are developing extends the base capabilities of the language by including some widely available class libraries (a general-purpose one, the GNU C++ Library [Lea92], and two simulation oriented ones, SimPack [Fish92] and AWESIME [Grun91, Grun92]). An overview of the simulation libraries is provided in the Appendix.

In this chapter, we use a few simple mathematical tools to build an architecture for a loosely-coupled simulation support environment based on QDS. The architecture is built up by using these tools to construct a *Four-Level Framework*. This framework is based on emerging international standards which are intended to be used to build standardized environments for both Software Engineering and Integrated Information Systems. In both cases, *Repositories* are used to maintain the information in the upper levels of the framework. A repository can be thought of as a meta-database. Just as a data-dictionary keeps track of the types of information available in a database system, a repository provides this capability on a much larger scale, for an entire information system. In addition to keeping track of the types of available information, a repository must also keep track of the tools, systems and applications that access and manipulate this information, as well as keep track of design information and activities (see [Hobb93, Mill94a, Mill94b, Tann94]).

2. Background

2.1. Integration of Data, Knowledge, and Model Management

A logically centralized repository can serve as the foundation for comprehensive information management and tool integration. Recently, several commercial repository products [Mill94b] have been developed to facilitate integration for enterprise-wide information systems. Because of the necessity for breadth in such systems, it is difficult to explore certain deeper issues. Therefore, our approach would be to limit the breadth, while increasing the depth. The area of simulation is ideal for this, since there are many stereotypical kinds of objects and interactions involved in simulation modeling. A QDS system requires sufficient meta-data to facilitate on-the-fly selection of simulation models as well as composition of simulation models from model components stored in the repository. It should also provide a simulation analysis interface that is as easy to use as querying an ordinary database.

Our loosely-coupled approach provides the flexibility to purchase and include a full-blown repository manager. (Because of their currently high cost we are not doing this in our prototype.) Ideally, like the state-of-the-art repository vendors, one should provide repository capabilities using an Object-Oriented Database Management System (OODBMS), (e.g., ObjectStore, ONTOS, UniSQL or ActiveKDL). Our initial prototype will, however, use a slightly extended Relational Database Management System (RDBMS) (Oracle, Version 7). (Oracle, Version 7 includes some capabilities found in OODBMS's.) The DBMS will be used to represent information in a variety of different forms, including Stored Data, Rules, Constraints, and Models (e.g., simulations). These forms allow information to be retrieved, derived, checked, and generated.

2.2. Query Driven Simulation

Query Driven Simulation (QDS) is based on the following simple notion: Simulation analysis involves the retrieval and manipulation of simulation data. Because of the complexity of simulation programs, this data should not be thrown away, but should be stored in a database. This is being done to a limited extent today using Relational DBMS's. It can be done more fully using OODBMS's. This is the first step of QDS; all simulation data no matter how complex their structure should be stored in a database. Therefore, more simulation data would become available. If the necessary information is not available, a QDS system will analyze the user's query to try to find a model to instantiate, and execute it in order to fill in the missing data. This is step two, which requires models, as well as formal descriptions of models, to be stored in a highly organized fashion in the repository. Finally, step three of QDS facilitates the rapid development of simulation models/programs via model composition and even model synthesis.

Layered on top of the DBMS and the repository manager will be a collection of tools and interfaces that support the notion of Query Driven Simulation (QDS). Simulation data will be stored in the DBMS, while information about this data (i.e., meta-data) will be stored in the repository. In addition, information about development tools, simulation analysis tools, and simulation models (abstraction of a simulation program), as well as the definitional aspects of the simulation programs themselves, will be stored in the repository. The repository will be the vehicle for the integration of fundamentally different types of data/knowledge. Users' queries may be answered by simple data retrieval, complex query processing, querying requiring heuristic knowledge and inference, or even model instantiation and execution. This will provide a testbed for experimentation with accuracy-time tradeoffs involved with retrieval versus information derivation/generation.

The technological foundation for Query Driven Simulation (QDS) consists of the following technologies:

1. Object-Oriented Programming Languages and Translators,
2. Object-Oriented Database Management Systems, and
3. Object-Oriented Repository Managers.

In particular, the DBMS and the Repository Manager support QDS by providing access to integrated knowledge, data, and model bases.

Using these technological foundations, a QDS system can be built to provide a variety of different types of users (e.g., simulation model builders, simulation analysts, and even business end-users) with a powerful, yet easy-to-use environment for simulation modeling and analysis. In a QDS system, users will predominantly interact with the system by formulating queries to retrieve stored data, data inferred from knowledge, or data generated from models. The system allows access to not only data, but also to knowledge and models. Information of vastly different character can be integrated using both algorithmic and AI-based techniques. Queries formulated using the query language (or GUI-based query tool) will allow users to retrieve information about the behavior of systems under study. Data from simulation runs will be stored by the system. If the information requested by a query is already stored in the database, more or less conventional query processing techniques will be applied. If rules are available to derive the requested information, then they will be fired and the results will be presented. However, if the amount of information retrievable (either stored or derived) is below a user-settable threshold, model instantiation and execution may be automatically carried out to generate the desired information.

Model instantiation occurs when the DBMS does not have sufficient data or knowledge to provide a satisfactory answer. In such a case, the QDS system will automatically create model instances that are executed to generate enough data to give a satisfactory answer to the user. The process centers around the creation of sets of input parameter values which are obtained by schema and query analysis. Model instantiation has the potential to require an enormous amount of computation in response to a query. Therefore, user settable thresholds as well as more sophisticated information generation rules incorporating user preferences are provided to control the amount of computation. If the threshold is at 100%, then all parameter sets implied by the query whose results are not already stored in the database, are used to instantiate models. At the opposite extreme if the threshold is at 0%, then only data generated from previous simulation runs will be retrieved.

2.3. Comparison with Other Ongoing Research

Some of the most closely related work to our QDS approach was presented in two sessions on Database Centered Simulation Environments held at the 1993 Winter Simulation Conference. (organized and chaired by Professor Miller).

1. *The Object Flow Model for Data-Based Simulation*, by Lois M.L. Delcambre (Oregon Graduate Institute) and Lissa F. Pollacia (Northwestern State). "The Object Flow Model uses an object-oriented database to describe entities and methods for manipulating entities and provides a visual formalism called the Object Flow Diagram (based on network-based process-oriented discrete event simulation languages), to describe the dynamic processing of an application" [Delc93].

2. *Databases: Designing and Developing Integrated Simulation Modeling Environments*, by Martha A. Centeno and Charles R. Standridge (Florida State). "Database management systems (DBMS) provide robust information storage, retrieval, and indexing functions needed by a simulation modeling environment (SME)" [Cent93]. Their work has been to develop ever-increasingly more sophisticated prototypes of SME's. Notably, Dr. Standridge while working for Pritsker and Associates (SLAM) developed the first commercial SME (called TESS [Stan87]) back in the mid 1980's. TESS used a custom-built DBMS. Dr. Centeno (1990) was one of the first to build a prototype SME based on a commercial relational DBMS, namely Oracle. (Another major effort along these lines was by Balci and Nance [Balc87]; they used INGRES).

3. *Applying Active Database Models for Simulation*, by Aloysius Cornelio (Bellcore) and Shamkant B. Navathe (Georgia Tech). "In this paper we propose a data model for simulation which builds on object oriented and active database principles to represent physical systems in terms of its structure and function" [Corn93]. Dr. Navathe is one of the leading researchers on database systems and is the only one we know of applying his expertise to simulation.

4. *Employing Databases for Large Scale Reuse of Simulation Models*, by Martin Hitz, Hannes Werthner and Tuncer Oren (University of Vienna). "To enhance reusability in the field of simulation, model bases must be equipped with powerful tools for retrieval, modification, and aggregation of simulation models. In this paper, the role of database management systems supporting a modeling environment is discussed and reuse oriented query interfaces are presented" [Hitz93]. Their future work includes replacing their Relational DBMS with an Object-Oriented DBMS.

5. *The Concept of Views in Simulation*, by Margarita Rovira, David Spooner and Jorge Haddock (Rensselaer). "The concept of views is widely used in the database community as a tool to reorganize and extend a database. It allows different users to look at the same data in different ways without changing the original data or violating its integrity. The same concept can be applied to a simulation model to find new high level interaction patterns among the components of a model without having to define new models and in a way that maintains consistency among the results" [Rovi93].

6. *A Prototype Implementation of a Model Management System for Discrete-Event Models*, by Melanie Lenard (Boston University). "The conceptual foundation for this MMS is Structured Modeling (SM) and Extended Structured Modeling (ESM). Using the SM and ESM

frameworks makes it possible to take an integrated approach to representing and managing both models and data" [Lena93].

Most of the important related work is reported in earlier papers by these authors. We also mentioned above the work of Balci and Nance. More distantly related to our QDS research is the work done on knowledge-based simulation. This work is highlighted by the following important projects: KBS [Redd86, Fox89], ROSS [McAr85], and SES/MBase [Zeig87, Zhan89, Zeig90]. Finally, a lifecycle perspective is given in a special issue on Object-Oriented Simulation and Simulation Support [Mill95].

3. The Object-Oriented Paradigm and the Core Model

In the last several years the Object-Oriented Paradigm has emerged as one of the most successful approaches to software development. The collection of technologies and methodologies that embody it, have matured sufficiently to make the object-oriented approach appropriate for numerous situations: systems programming/development, applications programming/development, and of course, simulation. Reports from industry have indicated that the benefits provided by the object-oriented approach (better modularity and code reusability) have yielded substantial productivity gains [Wilk93].

Both Object-Oriented Programming Languages (OOPL's) and Object-Oriented Database Systems (OODBMS's) share a common conceptual framework. This framework is often referred to as the core model, and it consists of the following conceptual elements.

1 *Objects*. The principal construct for the creation of programs, applications or systems is that of an object. An object has a state and a set of well-defined operations (or methods). The state is made up of attribute or data values. The collective action of the methods over time defines the behavior of the object.

2. *Object Identity*. The object itself is uniquely identifiable, irrespective of the values (or state) it contains. (This is in sharp contrast to the notion of keys which are used for these purposes in relational databases.)

3. *Classes*. Objects are classified or grouped into classes. The particular class that they are an $instance-of$ constrains their state and defines (or at least governs) their behavior.

4. *Encapsulation*. Objects can only function correctly if their state is not corrupted. An object is said to be fully encapsulated if its state is only accessible via the object's methods. An object is partially encapsulated if exceptions to this are allowed (e.g., public attributes or read-only attributes).

5. *Information Hiding*. This takes encapsulation one step further. Not only can the state of an object never be accessed directly, but its form is to be known only by the class implementor. This has very practical benefits for large systems; if a change to the form or structure of the state is made, the impact is minimal (e.g., primarily relinking instead of recompiling all modules).

6. *Message Passing*. A single uniform syntax is used for invoking all methods, whether the actual implementation corresponds to a regular function call (early binding), an indirect function call (late binding), or a remote procedure call (rpc) involving the transport of a packet over a Local-Area Network. The syntax should also be suggestive of the fact that one object is requesting another object to do something. The syntax is generally of the form `object method parameters` with suitable intervening punctuation.

7. *Inheritance*. Since the form and function of objects are specified by the classes, much of system development centers on specifying (e.g., in `.h` files) and implementing (e.g., in `.cc` files) classes. Inheritance is a mechanism by which new classes can be readily built

(or derived) from existing classes. In particular, if a derived class `inherits-from` a base (or parent) class, it inherits all the attributes and methods of the base class. The derived class has the option of redefining (or overriding) any of these as well as adding new ones.

8. *Polymorphism*. C++ allows base class pointers to point at an object of this or any derived class. Pure polymorphism comes into play when a method is to be invoked. If the statement `ptr->Display();` (where `ptr` is of type `Person*`) is to be executed, then which `Display()` method should be invoked, the one defined for the `Person` class, or one defined for a class derived from `Person`? Since `ptr` can point at objects of type `Person` or any type derived from it, typically, it is most useful for the particular method invocation to be determined on the basis of the type of object pointed to, rather than the type of the pointer. This will be the case in C++, if `Display()` is defined to be a virtual member function. This requires a method to be invoked by an indirect function call (late binding), since the specific type of an object must be determined at run time.

4. Multi-Level Framework

Tools and facilities in a loosely-coupled QDS system need to be carefully organized into a multi-level framework. The bottom levels of the framework will contain facilities such as programming language compilers and database management systems. In both cases, from an object-oriented point of view, we are dealing with objects and their types (or classes). Higher levels in the framework will contain facilities such as Data Dictionaries (DD’s), Repository Managers (RM’s), Model Management Systems (MMS’s), and Computer-Aided Software Engineering (CASE) tools.

At this point we go into a more detailed analysis of what constitutes a repository. The organization of a repository is best understood as a multilevel framework. We motivate this by considering the general concept first. A database is a large integrated collection of information which is organized according to type. Consequently, it is a two level concept, with instances (actual data) forming the lower level, and type information (schema specification) forming the upper level. Modern databases view this type information as important enough to store data about these types in a separate small database, known as a data dictionary. Since a data dictionary is itself a database, it is again a two level concept. Therefore, modern databases actually consist of two databases: a database storing the application data, and a database storing data about this application data (in other words meta-data). These notions can be clarified with a diagram consisting of three levels (see Table 1).

Dictionary Schema	**meta-model**	
Dictionary Data	**meta-data**	**Application Schema**
	data	**Application Data**

Table 1: Three-Level Framework.

The top level, the meta-model level, simply defines a type structure, but contains no data. Therefore, the meta-model specifies the type structure for the meta-data. Relevant types include `Table` (or `Class`), `Column`, and `Constraint`. (`Class` in an OODBMS is analogous to `Table` in an RDBMS). The instances that fill up these types would be stored at the level below. Examples of instances of `Class` type that one might find in say a QDS system consisting of transportation system simulators are the following:

`Person, Employee, Department, Car, Engine, Body`

Again, these instances at the meta-level represent types for the level below. Instances for the `Employee` type might include `employee A`, `employee B`, and `employee C`.

Multi-level frameworks, such as the one above, exhibit an interesting regularity. Each of the middle levels viewed from below represents type information, while viewed from above represents instances. The exceptions are the top-most and bottom-most levels. At the data level, there is no view from below, so that the type viewpoint does not exist. Similarly, at the meta-model level, there is no view from above, so no instance (or data) viewpoint exists.

The above discussion may lead one to ask how the meta-model is itself structured. The answer is that it is fixed and hard-coded. The vendor decides what kind of meta-data is needed for its product and supplies this. However, to meet the special demands of some of its customers, some vendors allow their data dictionary to be extended. A natural way to do this is to let the structuring information for the meta-data (i.e., the meta-model) be defined in yet another database, which is conceptually one level higher. Consequently, one ends up with a four-level framework, as shown in Table 2.

	meta-meta-model	**Definition Schema**
Dictionary Schema	**meta-model**	**Definition Data**
Dictionary Data	**meta-data**	**Application Schema**
	data	**Application Data**

Table 2: Four-Level Framework.

This new fourth level provides for extensibility and interoperability. Since this upper level database is relatively small, its structure can be fully specified in standards, and even if there are multiple standards, translations between these standards will be facilitated. An example of data at this fourth level might be the data that fills-in a type called `Meta-Class` which could consist of instances such as `Table`, `Column`, and `Constraint`. If an enterprise wishes to include information on application programs, they can extend the dictionary by adding an instance called `Application` to the `Meta-Class`.

In summary, we now have three databases, an Application Database, a Dictionary Database, and a Dictionary Definition Database, with each database consisting of a lower level (instances) and an upper level (types). We can think of these two levels as forming level-pairs, called, Definition, Dictionary, and Application. Hence, we have four levels and three level-pairs. This highly useful framework is the basis for several International Standards [Thom92]:

1. **IRDS** (Information Resource Dictionary System). This standardization effort was sponsored by ANSI (American National Standards Institute). A standard was completed in 1988 [Law88], but there are several proposals under review which would extend the standard. A draft of ANSI IRDS V2 [Hodg92, Gold93] which includes the ATIS (A Tool Integration Standard) specifications developed by DEC and Atherton Technologies has recently been produced [Hobb93]. Requirement #4 (of 259) for IDRS2 states that "the IRDS shall be object-oriented, with multiple inheritance and complex objects" [Gold93].

3. **PCTE** (Portable Common Tool Environment). This standardization effort was sponsored by ECMA (European Computer Manufacturer's Association). Standards documents [Simo93] were produced in 1986 (PCTE V1.4), 1988 (PCTE V1.5 and PCTE+), and 1991 (a joint report by both ECMA and NIST (National Institute of Standards and Technology)). Extensions are ongoing today, for example, a language-specific binding to C++ is currently under development. Recently, IBM has announced support for PCTE within their proposed

AIX repository [Thom93].

3. **CDIF** (Case Data Interchange Format). This standardization effort was initiated by a group of CASE vendors and was sponsored by EIA (Electronics Industry Association). The standard specifies a common format for import/export files.

Within these standards, the names for the levels vary, but the concepts remain the same. The various names are shown in Table 3.

general	IRDS	PCTE	CDIF	QDS
meta-meta-model	Fundamental	Meta Schema	Meta Meta	Meta-Meta-Class
meta-model	IRD Definition	SDS	Meta	Meta-Class
meta-data	IRD	Object	Model	Class
data	Application	(undefined)	Application	Object

Table 3: Standard Terminology for Levels.

Note, IRD stands for Information Resource Dictionary, while SDS stands for Schema Definition Set. In the section below we develop our QDS framework, level-by-level, starting with the object or bottom level.

5. Levels in a QDS System

As shown in Table 3, the four QDS levels are from bottom to top: the object-level, the class-level, the meta-class-level (or just meta-level), and finally at the top, the meta-meta-class-level (or simply just meta-meta-level). At each level, we are interested in three particular notions: the notion of an instance, the notion of a class extent, and the notion of a full extent. At the object-level, an instance simply corresponds to an object. All of the objects in the system constitute the full extent (i.e., the set of all objects). At the class-level, an instance corresponds to a class, while the full extent is the set of all classes. The same idea applies at the next two levels. Table 4 summarizes the situation.

Level	Instance	Class Extent	Full Extent
Meta-Meta-Class	$\mu^2{}_i$	-	-
Meta-Class	μ_i	M_j	**M**
Class	κ_i	K_j	**K**
Object	θ_i	Θ_j	Θ

Table 4: Level-by-Level Notation.

As yet, a connection between the levels has not been formally established. The mechanisms of instantiation and classification provide one clean way to establish the connection. These ideas are easiest to understand when dealing with the class--object level-pair.

For the class--object level-pair case, given a class κ_j, an application of the *instantiation* mechanism will produce some unique object θ_i. Minimally, this will require that space (volatile and/or persistent) be allocated for the object and its object identifier maintained. Typically, much of its state will be initialized at this time. Abstractly, this process adds another ordered pair to the $instance\!-\!of$ function.

$$instance\!-\!of\ :\Theta \rightarrow \mathrm{K}$$

The $instance\!-\!of$ function maps every $\theta_i \in \Theta$ to a unique $\kappa_j \in \mathrm{K}$. This instantiation process then establishes the link between the object-level and the class-level. The information stored about a class is used to give form and function to an instantiated object. It may also put constraints on the values that make up the state of the object.

Since the $instance\!-\!of$ function is not injective, the inverse of this function produces what we call a *classification* relation. Corresponding to each class κ_j will be a set of objects (i.e., all instances of κ_j). We may therefore define the *class extent* for say κ_j to be the following set:

$$\Theta_j = \{ \theta_i \mid instance\!-\!of\ (\theta_i) = \kappa_j \}$$

Clearly, the union of all the class extents Θ_j is equal to the full extent Θ.

The instantiation mechanism can be applied for higher level-pairs as well. For the meta-class--class level-pair, a meta-class μ_j can be used to instantiate an ordinary class κ_i. Various techniques for producing new classes are encoded as methods of the meta-class called `Class` (see Section 5.3).

Finally, at any level, there are a variety of ways to establish collections of instances. For example, at the object-level, a collection of objects can be established by drawing objects from some Θ_j. Typically, collections are either sets, bags or lists. Bags and lists allow an object to be drawn more than once, while a set does not. In addition, the objects in a list are totally ordered.

5.1. The Object-Level and Its Relationships

Beyond the core model, there are conceptual elements that elaborate and differentiate the way which objects can be related to each other. Although more relevant to database systems, these concepts are becoming more common in programming languages and in particular in Object-Oriented Design (OOD). For example, the Object Modeling Technique [Rumb91] is a design methodology that incorporates many of these elements and is used to design object-oriented implementations in languages such as C++. A great deal of work on semantic data models has contributed to the understanding of how objects relate to each other [Hull87]. (Note that while objects are named using noun phrases, relationships are named using verb phrases.)

Several types of relationships are generally recognized as being useful. Directed graphs will be used to formally model these relationships. A *Directed Graph* (or Digraph) $G(N, E)$ consists of a set of nodes N, and a set of edges (ordered pairs of nodes) E. ($G.N$ and $G.E$

denote the graph's node and edge set, respectively.)

5.1.1. Aggregation

One of the strengths of the object-oriented approach is that it allows complex objects to be built up from other objects. In the real-world, objects contain sub-objects which themselves are made up of sub-objects. This process can continue indefinitely or until one reaches the level of fundamental particles (i.e., the limit of our knowledge). In practice, the process stops when one reaches the point where the next level of sub-objects are not important for the purpose at hand. The objects that are not further decomposed are referred to as atomic objects. Those that are composed of sub-objects are referred to as composite or complex objects.

Given an object θ_i, its aggregation sub-graph is a directed graph consisting of it and all of its sub-objects. These form the nodes of the graph. The edges in the graph represent `has_a` relationships, that is, the edge (θ_i, θ_j) means that object θ_i contains sub-object θ_j, which is read "θ_i `has_a` θ_j". Objects for which there are no incoming aggregation (or `has_a`) edges are said to be independent objects. Otherwise, objects are dependent on the objects that contain them. If all of the objects on which θ_j is dependent are deleted, then θ_j should be deleted. (Note, the commonly used `is_part_of` relationships are simply the inverse of `has_a` relationships.)

A complete *aggregation graph* is simply the union of the sub-graphs for each independent object in the system. The directed graph is *Acyclic* since it makes no sense for an object to be a sub-object of itself. Thus, in general, aggregation graphs are Directed Acyclic Graphs (DAG's). Let us consider some other potential properties for the graph:

1. Connected. Although there are likely to be connected components, in a large system it is

unlikely for the entire graph to be connected.

2. Planar. In a theoretical sense this is irrelevant, but in a practical sense it is desirable as it enables CASE tools to display the graph without crossing lines.

3. Hierarchical. If the structure must be hierarchical, then an object may not be a part of more than one object. That is, there would be no *sharing* of sub-objects. This implies that the in-degree of each node is at most one. Combined with the fact that the graph is acyclic, this means that the graph is a forest (or collection of trees). Some systems enforce this restriction, but more commonly it is a constraint that can be added (or left out) at the discretion of the designer.

Let us consider a simple example of a C++ class that could be used in a simulation of transportation systems (see Figure 1). One type of transportation vehicle to be used is a `Car`. From this class specification actual cars or car objects may be created. The attributes are shown as protected members, while the methods are shown as public members. Public members are generally accessible, while protected members are only available to objects of type `Car` (or types derived form `Car`).

```
class Car {
protected:
    String        model;         // Attributes
    String        year;
    String        VIN;             // Vehicle ID No.
    Engine        car_Engine;    // Aggregation
    Body          car_Body;
    Manufacturer* make;          // Association
    Person*       owner;
    Person*       driver;
    Person*       passenger [3];
public:
    Car ();                      // Methods
    ~Car ();
    Boolean Enter (Person* p, Boolean is_Driver = FALSE);
    Person* Exit ();
    Drive (double delay);
}; // Car

class Engine {
protected:
    int horsepower;
    ...
public:
    ...
}; // Engine

class Body {
protected:
    Color paint;
    ...
public:
    ...
}; // Body

class Person {
protected:
    int ssn;
    ...
public:
    ...
}; // Person

class Manufacturer {
protected:
    String name;
    ...
public:
    ...
}; // Manufacturer
```

Figure 1: Example Classes.

The attributes `car_Engine` and `car_Body` are examples of sub-objects. Therefore aggregation relationships will exist between car objects and engine objects (and between car objects and body objects). Consequently, if the destructor (`~Car()`) for a car object is executed, `car_Engine` and `car_Body` will also be deleted.

5.1.2. Association

Sometimes the notion of association is not differentiated from the notion of aggregation. In complex systems, it may be important to do so. The idea is that an object can participate in associations when deemed appropriate, that is, they can be formed and disbanded at any time without impacting the existence of any of the participating members. The most common forms of associations can also be modeled as digraphs.

An *association graph* consists of all the objects in the system forming the nodes, and all of the associations forming the edges. In particular, a directed multigraph is needed, since a given pair of objects (θ_i, θ_j) can participate in multiple associations.

In the association graph, it is also very useful to include identifying edge-labels. An edge-label indicates which type of association an edge represents. For example, `department` objects may be involved in a couple of kinds of associations with `employee` objects (e.g., `employs` associations and `is_managed_by` associations). Although specifying the end-points (θ_i, θ_j) will not select a unique edge, adding the edge-label $(\theta_i, \theta_j; k)$ will.

In addition, the graph should be directed since the objects play different roles in the association. The designer should choose the object playing the more significant role to be the source, and the other object to be the destination. For example, since money flows from a `department` θ_i to an `employee` θ_j, a reasonable choice for a designer would be to specify associations as follows: $(\theta_i, \theta_j; \texttt{employs})$ which reads "`department` θ_i `employs employee` θ_j". (Similarly, $(\theta_i, \theta_j; \texttt{is\_managed-by})$ reads "`department` θ_i `is_managed-by employee` θ_j".) Note, unlike edges in aggregation graphs, the direction of edges in association graphs may at times be somewhat arbitrary. Because of this arbitrariness, it is possible for an association graph to be cyclic. (Some systems may wish to eliminate this possibility, since it complicates the management of objects. This control can be exercised during schema design.)

The example in Figure 1 also has attributes that represent association relationships. These are the `driver` and `passenger` attributes. Clearly, the existence of a passenger should not depend on whether s/he has entered or exited a car.

5.2. The Class-Level and Its Relationships

Class-level graphs can be derived from corresponding object-level graphs in relatively straightforward ways. Their utility is twofold: They provide a broader picture of the system, and more importantly, they can be used to constrain and/or give form to the lower level. The form of each association is defined in a class-level schema specification.

5.2.1. Association

Class-level associations provide the principal tool for designing database systems. Indeed, the Entity-Relationship Model [Chen76] is built primarily on this concept. A class-level graph may be constructed as follows: If a labeled edge in the object-level graph touches any object in some class κ_i, then there will exist a corresponding edge in the class-level graph.

The form of the association determines what classes the participating objects must come from. For example, an association (with label k) defined on classes κ_i and κ_j is a subset of

$$\{ (\theta_l, \theta_m; k) \mid \theta_l \in \Theta_i, \theta_m \in \Theta_j \}$$

where Θ_i is the class extent of κ_i and Θ_j is the class extent of κ_j. Note, it is not necessary for κ_i and κ_j to be different.

In addition to edges having labels, edges in the class-level graph also have an important property called cardinality. Given an edge in the class-level graph, (κ_i, κ_j), there are four possible values this property (the cardinality property) can take on.

1. One-to-One. For all edges $(\theta_i, \theta_j; k)$ with the same label k there is at most one θ_j for a given θ_i, and vice versa.

2. One-to-Many. For all edges $(\theta_i, \theta_j; k)$ with the same label k there may be many θ_j's for a given θ_i, but only one θ_i for a given θ_j.

3. Many-to-One. This is just the flip side of the previous case.

4. Many-to-Many. For all edges $(\theta_i, \theta_j; k)$ with the same label k there may be many θ_j's for a given θ_i, and vice versa.

Since associations among objects within the same class are not uncommon, such graphs may be cyclic. Also, since we are restricting ourselves to graphs, we are not able to model higher arity relationships such as ternary relationships found in Entity-Relationship (ER) Models. It is straightforward to extend our approach by using hypergraphs to represent higher arity relationship. We chose not to do so, feeling that the marginal value of introducing higher arity relationships is outweighed by their negative impact on complexity.

5.2.2. Aggregation

The class-level aggregation graph is constructed in much the same manner. If there is a `has_a` edge (θ_i, θ_j) in the object-level graph, there must be an edge (κ_i, κ_j) in the class-level aggregation graph. Although it makes no sense for an object to be part of itself, having cycles at the class-level is not unusual. For example, if one was modeling Pascal procedures, one component of a procedure can be another procedure since Pascal supports nested procedures.

5.2.3. Inheritance

An *inheritance graph* is a Directed Acyclic Graph (DAG) where each node corresponds to a class and each edge (κ_i, κ_j) indicates that class κ_j directly `inherits-from` κ_i. If a system does not support multiple inheritance (i.e., it supports only single inheritance), then the additional constraint that each node has an in-degree ≤ 1, makes the DAG into a forest (collection of trees). Unlike association or aggregation graphs, inheritance graphs can not be derived from the object-level. (Note, in the case above, it is commonly said that κ_j is a specialization of (or `is-a`) κ_i, or conversely, that κ_i is a generalization of κ_j.)

Since inheritance is a transitive property, the transitive closure of the inheritance graph will show the totality of inheritance. Given a class-level inheritance graph, G_{CI}, the transitive closure of this graph is denoted by $G_{CI}{}^*$.

If κ_j `inherits-from` κ_i, we often refer to κ_i as the super-class and κ_j as the sub-class. Since the sub-class can add features (attributes, methods, and constraints), the sub-class will typically have more features than its super-class. If we move down a level, though, and look at the corresponding class extents, Θ_i and Θ_j, the more specific class extent Θ_j is a subset of the more general class extent Θ_i $(\Theta_j \subseteq \Theta_i)$

If κ_i has another sub-class, say κ_k, then both Θ_j and Θ_k are subsets of Θ_i. It is often useful to allow the designer to selectively apply additional constraints on this sub-classing, such as requiring the sub-classes to be disjoint. As an example, let us consider two sub-classes of `Person`, namely, `Employee` and `Customer` as shown in Figure 2.

```
class Employee : public virtual Person {
protected:
    int         salary;
    Department* home_Department;
    ...
public:
    ...
}; // Employee

class Customer : public virtual Person {
protected:
    string credit_Rating;
    ...
public:
    ...
}; // Customer
```

Figure 2: Examples of Single Inheritance.

Because of the typing rules used by C++, it is not possible to create an object that is common to both `Employee` and `Customer`. However, if the designer specifies a third class called *Emp_Cust* that is derived (using multiple inheritance) from both `Employee` and `Customer`, then it is possible to have objects common to both.

```
class Emp_Cust : public Employee, public Customer {
protected:
    int discount;
    ...
public:
    ...
}; // Emp_Cust
```

Figure 3: Example of Multiple Inheritance.

Some additional terminology is in order here. If object θ_i is an $instance\!-\!of$ `Emp_Cust` then it is also a member of all the super-classes, `Person, Employee,` and `Customer`. It is an $instance\!-\!of$ the unique class from which it was constructed. More formally, we may define the closure of a class extent Θ_j as follows:

$$\Theta_j{}^* = \cup \{ \Theta_k \mid k = j \ or \ (\kappa_j, \kappa_k) \in G_{CI}{}^*.E \}$$

5.2.4. Communication

An additional type of relationship between classes is depicted by the *communication graph*. The nodes in the graph represent classes. The existence of a directed edge between κ_i and κ_j, (κ_i, κ_j), means that there exists a flow of control (thread path) that will result in an object θ_i, an $instance\!-\!of\ \kappa_i$, sending a message to θ_j, an $instance\!-\!of\ \kappa_j$. This graph can be produced, for example, by performing a static source code analysis of C++ implementation files. Unfortunately, finding a minimally sufficient graph is an undecidable problem. However, a close-enough approximation (which is still sufficient) can be built by assuming (i) that every condition in the program can be evaluated to both TRUE and FALSE values, and (ii) that where polymorphism is possible, all relevant virtual functions can be invoked. Note, as with the inheritance graph, the communication graph has no object-level equivalent. (Some systems may indeed have an object-level equivalent if authorizations are necessary for accessing objects.)

5.2.5. Binding Energy

Each type of relationship has a different degree of binding energy or cohesive force. From strongest to weakest we have inheritance, aggregation, association, and then communication.

5.3. The Meta-Level and Its Relationships

The meta-level is conceptually produced via the classification/instantiation mechanism which groups abstractly related classes together. As shown in Table 4, for every class-level class extent K_i there is a corresponding meta-class μ_i. An $instance\!-\!of$ a meta-class is an ordinary class, and the fact that it is, implies that it must conform to the abstract specifications given by the meta-class. In other words, meta-classes can impose constraints on the form and function of ordinary classes. This is analogous to the influence that classes have over objects, just at a higher level of abstraction.

From an object-oriented point of view, programs written in languages like C++ primarily consist of classes as well as functions. Clearly, we would then need meta-classes for these. Each of these meta-classes would contain descriptive information about actual classes. In particular, for example, the meta-class called `Class` would contain attributes and methods. An instance of this meta-class would be a *fully elaborated* class specification. From this, a standard class specification (e.g., in the form of a `.h` file) can be generated. In addition to standard information found in `.h` files, the fully elaborated specification could include the following: documentation on the purpose and use of a class, diagrammatic information suitable for CASE tools, and information about relationships with other classes.

Because of the complex nature of ordinary class specifications, the `Class` meta-class will have many attributes and methods. These will involve other meta-classes such as `Attribute`, `Method`, `Type`, and `Function`. Furthermore, the `Class` meta-class will have several sub-meta-classes derived from it. This allows additional information and control to be maintained over a set of classes which share a common abstract behavior. This more detailed information is also the basis for computer-aided simulation model development. Useful sub-meta-classes include `Event_Class`, `Thread_Class`, `Entity_Class`, `Monitored_Class`, `Persistent_Class`, `Class_Template` and `Abstract_Class`.

Let us examine the details of what would appear in some of the meta-level classes. Probably the most important meta-level class is the one called `Class`. A specification of this meta-class would look something like the following. Note that for the sake of brevity and simplicity the methods for these meta-classes are left out.

```
class Class : public Type {
protected:
    Set<Attribute> new_Attribute;   // New attributes
    Set<Attribute> red_Attribute;   // Redefined attributes
    Set<Method>    new_Method;      // New methods
    Set<Method>    red_Method;      // Redefined methods
    Set<Assertion> new_Invariant;   // New invariants
    Set<Assertion> red_Invariant;   // Redefined invariants
    ...
public:
    ...
}; // Class

Set<Class> Class_Collection;   // Full Extent for Class-Level
```

Figure 4: The Principal Meta-Class.

Some of the important methods for meta-class `Class` are methods to retrieve information from the graphs (inheritance, aggregation, association and communication, see Figure 6 below). There would be a method to return a string that contains a definition of the class suitable for a compiler to process. In addition, methods to support schema evolution are very important. Schema evolution embodies technologies and methodologies that extend, adapt and change schema. There need to be methods to do the following:

1. Create a class.

2. Remove a class.

3. Derive a sub-class using the inheritance mechanism.

4. Generate a class by instantiating a class template.

5. Add a feature (`Attribute, Method,` or, `Assertion`) to a class.

6. Remove a feature from a class.

7. Turn assertion checking on or off.

Of these, deriving (sub-) classes from super-classes, and generating (template) classes from class templates provide the greatest opportunity of code reuse. Class derivation using the inheritance mechanism has been explained before. However, a short explanation of templates is in order. In C++, templates can be used to generate functions and classes. An example of a class template is shown in Figure 5. The unique aspect of the template is that the Set data structure is parameterized by type. Actual classes can be generated from this template as follows:

```
template<class T> class Set { ... };

Set<int>    Integer_Set;
Set<Person> Person_Set;
```

Figure 5: Example of a Template.

The class-level graphs (inheritance graph, aggregation graph, association graph and communication graph) are maintained by the following meta-level classes.

```
class Digraph {
protected:
    Set<Adjacency_List> out_Node_List;
public:
    ...
}; // Association

class DAG : public Digraph {
public:
    ...     // Methods will be redefined to prevent cycles
}; // DAG

class Multigraph {
protected:
    Set<Labeled_Adj_List> out_Node_List;
public:
    ...
}; // Multigraph

class Adjacency_List {
protected:
    Class      source;
    Set<Class> dest;
public:
    ...
}; // Adjacency_List

class Labeled_Adj_List {
protected:
    Class            source;
    Set<Label_Class> dest;
public:
    ...
}; // Labeled_Adj_List

class Label_Class {
protected:
    String edge_Label;
    Class  node;
public:
    ...
}; // Label_Class

DAG        Inheritance_Graph;    // Instances
Digraph    Aggregation_Graph;
Multigraph Association_Graph;
Digraph    Communication_Graph;
```

Figure 6: Meta-Classes for Graphs.

A complete specification of the meta-level would be quite involved. We will thus limit ourselves to some meta-classes related to the meta-classes given above.

```
class Class_Template : public Class {
protected:
    Set<Parameter> argument;
public:
    ...
}; // Class_Template

class Type {
protected:
    String name;
    String description;
public:
    ...
}; // Type

enum Protection_Level { public, protected, private };

class Attribute {
protected:
    String           name;
    String           description;
    Type             domain;
    Protection_Level visibility;
public:
    ...
}; // Attribute

class Method {
protected:
    String           name;
    String           description;
    Set<Assertion>   pre_Conditions;
    Set<Assertion>   post_Conditions;
    Set<Parameter>   parameter;
    Type             return_Type;
    Protection_Level visibility;
public:
    ...
}; // Method
```

Figure 7: Other Related Meta-Classes.

Recall that definitions such as the ones above really relate to a level-pair, in this case, the meta-class--class level-pair. The type specifications are properly at the meta-level, while the instances (or sets of instances) are the level below, the class-level. For example, the instances of the `Class` meta-class (call it μ_1) constitute the set called `Class_Collection` $= K_1 = K$. Sub-classes of `Class`, for example, μ_i, will produce subsets (K_i) of `Class_Collection`. The situation is the same for the class-level graphs which record the relationships between ordinary classes. Their form is specified at the meta-level, while the actual data is stored as class-level information.

Please note that what is important here are the concepts and not the particular syntax. Since we have a loosely-coupled system, we could have just as well defined `Class` as an Oracle, Version 7 table, or as a class using the syntax of ONTOS or ObjectStore. In the latter two cases, no major changes to the code given above would be needed, since their schema definition languages are based on C++.

As you might expect, a rich set of relationships is possible at this level, the meta-level. For one, since we are using C++ syntax at this level to define meta-classes, it should be apparent that graphs analogous to those provided at the class-level (namely, inheritance, aggregation, association and communication graphs) are useful at this level also. These higher level graphs keep track of relationships between meta-level classes. A couple of examples of such relationships are the following:

1. An edge in the meta-level inheritance graph is (`Type, Class`). This edge indicates that the meta-class `Class` is derived from the meta-class `Type`.

2. An edge in the meta-level aggregation graph is (`Class, Method`). This edge indicates that the meta-class `Method` is a part of the meta-class `Class`. Furthermore, a property

of this aggregation is that methods can not be shared.

It is also possible to conceive of meta-level graphs whose chief purpose is not to structure the meta-level, but rather to provide abstract constraints on the graphs at the level below. For example, there might be a type of aggregation graph at the meta-level, that indicates what family of ordinary classes can come together to produce composite classes. These families of classes could be specified by choosing appropriate sub-meta-classes of `Class`. That is, for some μ_j the class extent K_j would define a family of classes.

In summary, all four types of meta-level graphs (inheritance, aggregation, association and communication) are useful at this level. The data for these graphs is stored as meta-level information, while their form (or type) is specified at the meta-meta-level.

5.4. The Meta-Meta-Level

The principal reasons for having this even more abstract level are interoperability and extensibility. Conceptually, it is much like the meta-level. The goals at these two levels are, however, different. The meta-level should be comprehensive in that one should be able to store all varieties of meta-data, and should be able to tailor and extend the capabilities to match current demands. In contrast, the meta-meta-level should be minimalistic and standardized.

The meta-meta-level is conceptually produced via the classification/instantiation mechanism that groups abstractly related meta-classes together. As shown in Table 4, for every meta-class-level class extent M_i there is a corresponding meta-class $\mu^2{}_i$. An $instance\!-\!of$ a meta-meta-class is a meta-class, and the fact that it is, implies that it must conform to the abstract specifications given by the meta-meta-class.

Essentially, the main purpose of the meta-meta-level is to provide a fixed set of modeling facilities to allow meta-level modeling facilities to be defined, extended and processed. The fact that two systems share the same top-level modeling facilities and have the ability to process (e.g., translate) instances from the level below, makes it easier for them to interoperate. In particular, the fact that the level below (i.e., the meta-level) is fully defined in terms of this standardized top-most level, means that general-purpose translators can be constructed. Consequently, new special-purpose translators do not need to be developed every time there is a new (or upgraded) tool or package to integrate into a system, as is the current state of affairs.

Previously, standards organizations have based the modeling facilities for the meta-meta-level on Extended Entity-Relationship (EER) Models. However, their new draft proposals are based on Object-Oriented (OO) Models. The most important meta-meta-class is the class called `Meta_Class`. This records information about the actual meta-classes available (e.g., `Class, Digraph, Attribute, Method`). The meta-level can be extended simply by adding a new instance to `Meta-Class` (e.g., `Function`).

In any event, the fixed set of modeling primitives provided by the meta-meta-level should be powerful enough to allow the meta-level to be created using the four fundamental types of relationships (inheritance, aggregation, association and communications).

6. Summary and Conclusions

Before we become *level-lost*, let us review the situation. The easiest way to see the connections is to consider the three level-pairs and the instantiations that provide the interlocking.

1. Meta-Meta-Class--Meta-Class Level-Pair. A meta-class $\mu_j \in M$ is an $instance-of$ some meta-meta-class $\mu^2{}_i$.

2. Meta-Class--Class Level-Pair. An (ordinary) class $\kappa_k \in K$ is an $instance-of$ some meta-

class μ_j.

3. Class--Object Level-Pair. An object $\theta_l \in \Theta$ is an $instance-of$ some class κ_k.

Perhaps the easiest way to understand all of the graphs is also in terms of level-pairs. Recall that each of the levels except the top-most (the meta-meta-level) had a variety of types of graphs. For each of these levels the graphs record and depict the relationships between instances (be they objects, classes or meta-classes). It is also important to realize that the form (or type) of the graph is determined by the level above (e.g., for a class-level graph, the instances are classes, while the form of such a graph is specified as (or at least deducible from) meta-level information). We summarize this situation in Table 5, which for each level-pair, shows the types of graphs to be found.

Level-Pair	Association	Aggregation	Inheritance	Communication
M-M-Class--Meta-Class	**D M**	**D G**	**D A G**	**D G**
Meta-Class--Class	**D M**	**D G**	**D A G**	**D G**
Class--Object	**D M**	**D A G**	**-**	**-**

Table 5: Graphs for each Level-Pair.
Key: D - Directed, A - Acyclic, G - Graph, M - Multigraph

In this chapter, we have outlined a four-level architecture for simulation environments. Some of the advantages of this approach are the following:

1. Object-Oriented. The object-oriented approach has been shown to be successful in software development (analysis (OOA), design (OOD) and programming (OOP)), in database management (OODBMS), and in simulation (OOS). It thus, should be the basis for most integrated information systems.

2. Database-Centered. Databases and meta-databases should be critical elements (if not the centerpieces) of any complex integrated information system, such as a QDS system.

3. Loosely-Coupled. The wide variety of useful tools, systems and applications available today argue for a loosely-couple approach to integration, where components can be mixed and matched.

4. Relationship Rich. The considerable body of research on semantic data models indicate important benefits accrue by providing a full palette of relationships (inheritance, aggregation and association). Because OODBMS have methods it is also beneficial to keep track of their pattern of message passing via the communication relationship.

5. Flexible and Comprehensive Meta-Level. As is the case in emerging repository technology, we have split the meta information into two parts: a small standardized top level called the meta-meta-level, and an extensible and comprehensive lower level called the meta-level.

7. References

[Balc87] Osman Balci and Richard E. Nance, "Simulation Model Development Environments: A Research Prototype," *Journal of the Operational Research Society*, Vol 38, No. 8 (1987) pp. 753-763. (1987).

[Birt73] G. Birtwistle, O. Dahl, B. Myhaug, and K. Nygaard, *Simula Begin*, Studentlitertur and Auerbach Publishers (1973).

[Birt87] G. Birtwistle, *DEMOS: A System for Discrete Event Modelling on SIMULA*, Springer-Verlag, NY (1987).

[CACI90] CACI, *MODSIM II: The Language for Object-Oriented Simulation*, Los Angeles, CA (January 1990).

[Cent93] Martha A. Centeno and Charles R. Standridge, "Databases: Designing and Developing Integrated Simulation Modeling Environments," Proceedings of the 1993 Winter Simulation Conference, Los Angles, CA (1993) pp. 526-534.

[Chen76] Peter Chen, "The Entity-Relationship Model - Toward a Unified View of Data," *ACM Transactions on Database Systems*, Vol. 1, No. 1 (March 1976).

[Corn93] Aloysius Cornelio and Shamkant B. Navathe, "Applying Active Database Models for Simulation," Proceedings of the 1993 Winter Simulation Conference, Los Angles, CA (1993) pp. 535-543.

[Delc93] Lois M.L. Delcambre and Lissa F. Pollacia, "The Object Flow Model for Data-Based Simulation," Proceedings of the 1993 Winter Simulation Conference, Los Angles, CA (1993) pp. 519-525.

[Fish92] Paul A. Fishwick, "SimPack: Getting Started with Simulation Programming in C and C++," *Proceedings of the 1992 Winter Simulation Conference*, Arlington, VA (December

1992), pp. 154-162.

[Fox89] M.S. Fox, N. Husain, M. McRoberts, and Y.V. Reddy, "Knowledge-Based Simulation: An Artificial Intelligence Approach to System Modeling and Automating the Simulation Life Cycle," in *Artificial Intelligence, Simulation and Modeling*, L.E. Widman, K.A. Loparo, and N.R. Nielsen (Eds.), Wiley Interscience, NY (1989).

[Gold93] Alan Goldfine, *Information Resource Dictionary System Requirements Specification Standing Document*, NIST,(February 1993).

[Grun91] Dirk C. Grunwald, *A Users Guide to AWESIME: An Object Oriented Parallel Programming and Simulation System*, Technical Report, University of Colorado, Boulder, CO (1991).

[Grun92] Dirk C. Grunwald, *User's Guide to AWESIME-II: A Widely Extensible Simulation Environment*, Technical Report, University of Colorado, Boulder, CO (1992).

[Hitz93] Martin Hitz, Hannes Werthner and Tuncer Oren, "Employing Databases for Large Scale Reuse of Simulation Models," Proceedings of the 1993 Winter Simulation Conference, Los Angles, CA (1993) pp. 544-551.

[Hobb93] Lilian Hobbs and Ken England, *Digital's CDD/Repository*, Digital Press (1993).

[Hodg92] Robert E. Hodges, *Repository Context <Information Resource Dictionary System (IRDS)> Reference Model Technical Report*, NIST, (September, 1992).

[Hull87] R. Hull and R. King, "Semantic Database Modeling: Survey, Applications, and Research Issues," *ACM Computing Surveys*, Vol. 19, No. 3, (September, 1987).

[Koch91] Krys J. Kochut, John A. Miller and Walter D. Potter, "Design of a CLOS Version of Active KDL: A Knowledge/Data Base System Capable of Query Driven Simulation," *Proceedings of the 1991 AI and Simulation Conference*, New Orleans, LA (April 1991) pp.

139-145.

[Law88] Margaret Henderson Law, *Guide to Information Resource Dictionary System Applications: General Concepts and Strategic Systems Planning*, NBS Special Publication 500-152 (1988). (NBS is now NIST)

[Lea92] Doug Lea, *User's Guide to the GNU C++ Library*, Free Software Foundation, Inc., Boston, MA (1992).

[Lena93] Melanie Lenard, "A Prototype Implementation of a Model Management System for Discrete-Event Models," Proceedings of the 1993 Winter Simulation Conference, Los Angles, CA (1993) pp. 560-568.

[McAr85] D. McArthur, P. Klahr and S. Narain, *The ROSS Language Manual*, The RAND Corporation, N-1854-1-AF (September 1985).

[Mill89] John A. Miller and Orville R. Weyrich, Jr., "Query Driven Simulation Using SIMODULA," *Proceedings of the 22nd Annual Simulation Symposium*, Tampa, FL (March 1989) pp. 167-181.

[Mill90] John A. Miller, Walter D. Potter, Krys J. Kochut and Orville R. Weyrich, Jr., "Model Instantiation for Query Driven Simulation in Active KDL," *Proceedings of the 23rd Annual Simulation Symposium*, Nashville, TN (April 1990) pp. 15-32.

[Mill91a] John A. Miller and Nancy D. Griffeth, "Performance Modeling of Database and Simulation Protocols: Design Choices for Query Driven Simulation," *Proceedings of the 24th Annual Simulation Symposium*, New Orleans, LA (April 1991) pp. 205-216.

[Mill91b] John A. Miller, Krys J. Kochut, Walter D. Potter, Ender Ucar and Ali A. Keskin, "Query-Driven Simulation Using Active KDL: A Functional Object-Oriented Database System," *International Journal in Computer Simulation*, Vol. 1, No. 1 (1991) pp. 1-30.

[Mill91c] John A. Miller, Walter D. Potter, Krys J. Kochut, Ali A. Keskin and Ender Ucar, "The Active KDL Object-Oriented Database System and Its Application to Simulation Support," *Journal of Object-Oriented Programming*, Special Issue on Databases, Vol. 4, No. 4 (July-August 1991) pp. 30-45.

[Mill92] John A. Miller, Walter D. Potter and Krys J. Kochut, "Knowledge, Data, and Models: Taking an Objective Orientation on Integrating these Three," *IEEE Potentials*, Vol. 11, No. 4 (December 1992) pp. 13-17.

[Mill94a] John A. Miller, Walter D. Potter, Krys J. Kochut, Sunderraman Krishnan, Bonnie Edwards, Wensheng Zhang and Jayesh Sahasi, *Design of a WSRC Repository with an End-User Emphasis*, Westinghouse Savannah River Company (WSRC) Technical Report, Aiken, SC (January 1994).

[Mill94b] John A. Miller, Walter D. Potter and Krys J. Kochut, "Repository Design with an End-User Emphasis," *Proceedings of the 5th International Users Group Conference: Repository/Architecture/Development*, Chicago, Illinois (November 1994) 23 pages.

[Mill95] John A. Miller, Guest Editor, *International Journal in Computer Simulation*, Special Issue on Object-Oriented Simulation and Simulation Support, Vol. 5, No. 4 (1995).

[Pott86] Walter D. Potter and Larry Kerschberg, "A Unified Approach to Modeling Knowledge and Data," *Proceedings of the IFIP TC2 Conference on Knowledge and Data (DS-2)*, Algarve, Portugal (November 1986). (Published by North-Holland as *Data and Knowledge DS-2* (1988).)

[Pott88] Walter D. Potter and Robert P. Trueblood, "Traditional, Semantic and Hyper-Semantic Approaches to Data Modeling," *IEEE Computer*, Vol. 21, No. 6 (June 1988) pp. 53-63.

[Pott89] Walter D. Potter, Robert P. Trueblood and Carolyn M. Eastman, "Hyper-Semantic Data Modeling," *Data & Knowledge Engineering*, Vol. 4 (1989) pp. 69-90.

[Pott90] Walter D. Potter, John A. Miller, Krys J. Kochut and Sanford W. Wood, "Supporting an Intelligent Simulation/Modeling Environment Using the Active KDL Object-Oriented Database Programming Language," *Proceedings of the Twenty-First Annual Pittsburgh Conference on Modeling and Simulation*, Vol. 21, Part 4, Pittsburgh, PA (May 1990) pp. 1895-1900.

[Pott92] Walter D. Potter, Terry Anthony Byrd, John A. Miller and Krys J. Kochut, "Extending Decision Support Systems: The Integration of Data, Knowledge, and Model Management," *Annals of Operations Research*, Special Issue on Model Management, Vol. 38 (1992) pp. 501-527.

[Pott93a] Walter D. Potter, Krys J. Kochut, John A. Miller, Veena P. Gandham and Ramakrishna V. Polamraju, "The Evolution of the Knowledge/Data Model", *International Journal of Expert Systems*, Special Issue on Artificial Intelligence and Databases, Vol. 6, No. 1 (1993) pp. 39-81.

[Pott93b] Walter D. Potter, John A. Miller and Krys J. Kochut, "A Hyper-Semantic Approach to Intelligent Information Systems," *Integrated Computer-Aided Engineering*, Special Issue on Intelligent Information Systems (1993).

[Redd86] Y.V. Reddy, M.S. Fox, N. Husain, and M. McRoberts, "The Knowledge-Based Simulation System," *IEEE Software* (March 1986).

[Rovi93] Margarita Rovira, David Spooner and Jorge Haddock, "The Concept of Views in Simulation," Proceedings of the 1993 Winter Simulation Conference, Los Angles, CA (1993) pp. 552-559.

[Rumb91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy and William Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, Inc. Englewood Cliffs, NJ (1991).

[Simo93] Alan R. Simon, *The Integrated CASE Tools Handbook*, Van Nostrand Reinhold, NY (1993).

[Stan87] Charles Standridge and Alan Pritsker, *TESS: The Extended Simulation Support System*, Halstead Press, NY (1987).

[Stro91] Bjarne Stroustrup, *The C++ Programming Language*, Second Edition, Addison-Wesley, Reading, MA (1991).

[Tann94] Adrienne Tannenbaum, *Implementing a Corporate Repository: The Models Meet Reality*, John Wiley & Sons, Inc., NY (1994).

[Thom92] A. Kenneth Thompson, "CASE Data Integration: The Emerging International Standards," *ICL Technical Journal* (May 1992) pp. 54-66.

[Thom93] Graham Thompson, "The Need for CASE Integration," R&O, Inc. Technical Report, (April 1993).

[Wilk93] George Wilkie, *Object-Oriented Software Engineering: The Professional Developer's Guide*, Addison-Wesley, Reading, MA (1993).

[Zhan89] G. Zhang and Bernard P. Zeigler, "The System Entity Structure: Knowledge Representation for Simulation Modeling and Design," in *Artificial Intelligence, Simulation and Modeling*, L.E. Widman, K.A. Loparo, and N.R. Nielsen (Eds.), Wiley Interscience, NY (1989).

[Zeig87] Bernard P. Zeigler, "Hierarchical, Modular Discrete Event Modelling in an Object Oriented Environment," *Simulation Journal* 49, 5 (November 1987).

[Zeig90] Bernard P. Zeigler, *Object-Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems*, Academic Press, Inc., Boston, MA (1990).

8. Appendix: Example C++ Packages

A couple of widely available (and free) packages suitable for providing the foundation of the simulation language component of the QDS environment are SimPack [Fish92] and AWESIME [Grun91, Grun92]. In this appendix, we provide an overview of these two simulation libraries.

8.1. SimPack

SimPack is a simulation package or toolkit made up of a collection of C and C++ libraries. It supports several different simulation approaches including discrete event simulation, continuous simulation and combined or multimodel simulation. The toolkit aims at providing the user with a basic set of utilities to produce a working simulation from a model description. The toolkit includes examples to show how several special purpose simulation programming languages can be constructed using language translation software within the SimPack utilities.

SimPack was created with certain issues in mind. The important issues are outlined below.

1. Variety of Model Types. Since different models are suitable for different problems, the creators of SimPack felt that a simulation package must provide a set of tools that allow translation from unique graphing approaches (e.g., Petri nets for resource graphs, and queueing graphs for queueing problems) into callable routines. SimPack arose out of these ideas and hence provides many varieties of model types when compared to other packages.

2. Creation of Template Algorithms. The creators of SimPack found that in most cases users find it difficult to know exactly where to begin when creating a simulation for a particular application. To make it easier for users, SimPack provides a set of template algorithms so that the users can see how to solve a simple problem of a specific type and develop their

simulations based on these templates.

3. No Special Language Syntax To Be Learned. As in the case of CSIM and SMPL, the aim of SimPack was to make sure that any C or C++ programmer would be able to write simulations, without having to learn a new language.

4. Event and Process Oriented Discrete Event Simulation. SimPack was also designed to show the relationship between Event and Process oriented Discrete Event Simulation. SimPack views process oriented simulation as one level higher than that of event orientation. Users interested in programming at a process level can use a SimPack tool such as miniGPSS which compiles process code into event code. Hence, event orientation provides a base level in SimPack and process orientation is achieved by building software that hides this level.

SimPack is available free of charge to anyone who wants to build any of a wide range of simulation models. SimPack Version 2.22 is available for anonymous ftp from ftp.cis.ufl.edu (cd to pub/simdigest/tools, specify ’binary’ and get simpack-2.22.tar.Z and simpack-2.1++.tar.Z).

Some of the model types supported by SimPack are as follows:

1. Declarative Models. These emphasize explicit state to state changes as found in finite state automata and Markov models.

2. Functional Models. These deal with functions or procedures as in queuing networks, block models, pulse processes and stochastic Petri nets.

3. Constraint Models. These focus on differential and difference equations.

4. Multimodels. These deal with conglomerates of more than one model connected in a graph or network to solve combined simulation problems at multiple abstraction levels.

Some of the components of SimPack of particular interest to us are the following:

1. SimPack Queuing Library. The queuing library includes a set of functionally complete routines, similar to the SMPL routine library by M.H. Mac Dougall. The SMPL library was used as a base and SimPack added several routines and extended options. The event list can be made of any one of the following types: HEAP, CALENDAR or LINKED. SimPack also includes the code necessary to implement nine data structures for use in the SimPack Queuing Library. To use them a call to initialize is needed with any one of the following nine data structures: HEAP, LINKED, CALENDAR, HENRIK, BINOMIAL, LEFIST, PAGODA, PAIR, SKEWDN, SKEWUP, SPLAY or TWOLIST.

2. MiniGPSS Compiler. A subset of the GPSS Simulation Programming Language, MiniGPSS has been built using YACC and LEX. This makes it very easy to construct a simulation language for any other purpose using this compiler. Given a GPSS model description, the MiniGPSS compiler compiles it into C code. The generated C code also allows the user to see the manner in which the simulation is implemented.

8.2. AWESIME

AWESIME is short for A Widely Extensible SImulation Environment. It is an object-oriented library for parallel programming and for process-oriented simulation on computers with a shared address space. AWESIME is written in C++. Its features go beyond that of other similar packages, in that, AWESIME has a group of C++ classes that can manipulate or specify different aspects of a parallel programming environment. Classes can simply be added to these classes to build more complex process oriented simulations. In this added functionality, AWESIME takes after the CSIM simulation package.

At present, AWESIME has been ported and configured on the following architectures: Sun SPARC, Motorola 680x0, Intel i386, the Motorola 88K family, the National Semiconductor NS32K family and the MIPS R2000/R3000 machines. Just like SimPack, AWESIME is free and is available for anonymous ftp from ftp.cs.colorado.edu (cd to /pub/distribs/Awesime and get Awesime-12-21-93.tar.gz and users-guide.ps.Z).

AWESIME consists of a library of C++ classes, built into a Class Hierarchy. The classes provided include container classes (e.g., for queues and priority queues), classes for random variates, classes for statistics collection and analysis (including a sub-class for histograms), and a variety of classes to support process oriented simulation. The latter are of particular interest to us, so we will discuss them further.

8.2.1. Threads

Threads represent the lowest level of concurrency in AWESIME. User defined threads (threads in any application) must be declared as a sub-class of this class and threads must only be created using the function new. Note that defining threads statically at the main level is not allowed and though possible within routines it is not recommended. The class Threads includes a few class instance variables that are used to record details of the calling environment. The thread constructor is used to provide this information to the class. The destructor is just to clean up the class variable. When a thread is executed for the first time, it must be scheduled and a series of other actions result. One of these is a call to routine main(). An absence of main() is reported as an error.

8.2.2. Thread Container Class

A thread container is something that a thread is placed in. The threadContainer class by itself is an abstract one and only its sub-classes are of interest to us. A few of the methods in this class are: (1) add a thread, (2) remove a thread, (3) return the size of the container (# of threads in it), and (4) test for emptiness of the thread container.

Some important sub-classes of this class are the following:

1. CPU. The abstract processor which executes the AWESIME program. The CPU class includes several sub-classes that implement CPU multiplexing (CPU switching between threads when a synchronization constraint delays a specific thread).
2. FifoScheduler. As the name suggests, this provides first-in-first out thread schedulers.
3. HeapScheduler. This provides priority thread schedulers.

8.2.3. Synchronization Classes

Synchronization is done by two levels of locking. One is at the CPU level and the other is at the Thread level. CPU Level Locking is done using the SpinLock class. The reserve and release members perform the functions to reserve and release the entire CPU and not just the thread currently using the CPU. SpinLocks are efficient and use inline functions for reserve and release to reduce function call overhead. Thread Level Locking can be accomplished by a variety of specialized classes: Monitor, Barrier Synchronization, EventSequence. Condition, Join, Semaphore, Facility, and Owned Facility.