Using Domain Specific Languages for Modeling and Simulation: ScalaTion as a Case Study

John A. Miller
Jun Han
Maria Hybinette

Department of Computer Science
The University of Georgia
Conceptual Model vs. Simulation Program

What is the difference?

**Conceptual Model**, e.g., SysML

(http://www.omgsysml.org/)

**Simulation Program**, e.g., Fortran

(http://www-d0.fnal.gov/~hirosky/trigger/l2prod/l2_comp_leta.for)

---

1. **Structure**

   ![Structure Diagram](image)

   - **Vehicle System Specification**
   - **Braking Subsystem Specification**

   **requirements**

   - *Precondition*: The vehicle shall stop from 60 mph within 135 ft on a clean dry surface.

2. **Behavior**

   ![Behavior Diagram](image)

   - **Vehicle Library Interface**
   - **Braking Library Interface: Eeb or Hydraulic Valve**

3. **Requirements**

   ![Requirements Diagram](image)

   - **Vehicle System Specification**
   - **Braking Subsystem Specification**

4. **Parametrics**

   ![Parametrics Diagram](image)

   - **Vehicle Library Interface**
   - **Braking Library Interface: Eeb or Hydraulic Valve**

---

OKOK = .FALSE.
NRUN = IQ(LHEAD+6)
NEV = IQ(LHEAD+9)

C

IF (IB.NE.2) THEN
    CALL ERRMSG('L2_COMPARE','L2_COMP_LETA','L2_COMP_LETA called for IB.NE.2 ! Not allowed!!!','F')
    RETURN
ENDIF

C

C make sure there is LETA - no mistakes
C

IF (LLETA(1).LE.0) THEN
    WRITE(LUN,'('' L2_COMP_LETA: Run/Event '',2I7,'' has NO LETA bank for SIMULATION'')') NRUN,NEV
    GOTO 999
ENDIF

IF (LLETA(2).LE.0) THEN
    WRITE(LUN,'('' L2_COMP_LETA: Run/Event '',2I7,'' has NO LETA bank for DATA'')') NRUN,NEV
    GOTO 999
ENDIF

ENDIF
"To those who do not know mathematics it is difficult to get across a real feeling as to the beauty, the deepest beauty, of nature"

\[
S = \int \partial^\mu \phi \partial_\mu \phi + \frac{\lambda}{4!} \phi^4.
\]
Progress from the Right:
Evolution of General-purpose Programming Languages (GPLs)

• First Programming Language
  – **UNIVAC SHORT CODE, 1949**

• First Machine Independent Programming Language
  – **FORTRAN, 1954**

• First Structured Programming Language
  – **ALGOL, 1958**

• First Functional Programming Language
  – **LISP, 1958**

• First Object-oriented Programming Language
  – **SIMULA, 1967**

• First Functional Object-oriented Programming Language
  – **Common Lisp Object System (CLOS), 1988**
## Object-oriented Languages with Functional Features

<table>
<thead>
<tr>
<th>Language</th>
<th>Developer</th>
<th>Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCaml</td>
<td>Remy</td>
<td>Smith 2006</td>
</tr>
<tr>
<td>F#</td>
<td>Syme</td>
<td>Syme et al. 2007</td>
</tr>
<tr>
<td>Scala</td>
<td>Odersky</td>
<td>Odersky et al. 2008</td>
</tr>
<tr>
<td>Python</td>
<td>van Rossum</td>
<td>Watters et al. 1996</td>
</tr>
<tr>
<td>Ruby</td>
<td>Matsumoto</td>
<td>Thomas and Hunt 2000</td>
</tr>
<tr>
<td>Groovy</td>
<td>Laforge</td>
<td>Koenig et al. 2007</td>
</tr>
<tr>
<td>C#</td>
<td>Hejlsberg</td>
<td>Hejlsberg et al. 2003</td>
</tr>
</tbody>
</table>
Progress from the Left:
Enrichment of Conceptual Modeling

• Conceptual Modeling Artifacts
  – Requirements Document
  – Goals and Objectives
  – Terminology/Ontology
  – Model Design Specification

• Not meant to be executable, but should facilitate the consistency checking
Progress from the Left (cont.)

• Higher-level of Discourse than Simulation Programs
  – Design Diagrams
    • Process flow diagrams
    • Activity cycle diagrams
    • Petri nets
    • Event graphs
    • UML (Unified Modeling Language)
  – Component Descriptions
  – Mathematical Models for Elements or Verification
  – Alignment of a Domain Ontology with a Modeling Ontology, e.g., DeMOforge
## Progress in the Middle:
Simulation Programming Languages (SPLs)

<table>
<thead>
<tr>
<th>Language</th>
<th>Developer</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>GASP</td>
<td>Kiviat</td>
<td>1961</td>
</tr>
<tr>
<td>GPSS</td>
<td>Gordon</td>
<td>1961</td>
</tr>
<tr>
<td>SIMSCRIPT</td>
<td>Markowitz</td>
<td>1963</td>
</tr>
<tr>
<td>SIMULA 67</td>
<td>Nuggard and Dahl</td>
<td>1967</td>
</tr>
<tr>
<td>SLAM</td>
<td>Pritsker</td>
<td>1979</td>
</tr>
<tr>
<td>SIMAN</td>
<td>Pegden</td>
<td>1985</td>
</tr>
</tbody>
</table>

Later advances: simulation environments, animation and graphical model construction
Creating DSLs for M&S

• Is there a faster way to bridge the gulf?
• **GPLs** and **SPLs** each have their own pros and cons

Why not try using a Domain Specific Language (**DSL**)?
A Domain Specific Language (DSL)

• Definition:
  – “is a **programming language or executable specification language** that offers, through appropriate notations and abstractions, **expressive power** focused on, and usually restricted to, a **particular problem domain**”

• Key Advantage:
  – “trades **generality for expressiveness** in a limited domain. By providing notations and constructs tailored toward a particular application domain, they offer **substantial gains in expressiveness and ease of use** compared with GPLs for the domain in question, with corresponding **gains in productivity and reduced maintenance costs**”
Domain Specific Language (DSL)

• Types
  – Externally Defined DSLs
    • Requires pre-processors, parsers and code generators
    • This category includes some SPLs
  – Internally Defined or Embedded DSLs
    • Definable using the advanced features of the parent language
    • Easy to develop such DSLs
    • Easy to learn for those familiar with the parent language
## Languages Facilitating DSLs

<table>
<thead>
<tr>
<th>Languages</th>
<th>Object oriented</th>
<th>Functional</th>
<th>Type checking</th>
<th>Conciseness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>Impure</td>
<td>Very little</td>
<td>Static</td>
<td>Low</td>
</tr>
<tr>
<td>Python</td>
<td>Lack of encapsulation</td>
<td>Many Features</td>
<td>Dynamic</td>
<td>High</td>
</tr>
<tr>
<td>Ruby</td>
<td>Pure</td>
<td>Many Features</td>
<td>Dynamic</td>
<td>High</td>
</tr>
<tr>
<td>Scala</td>
<td>Pure</td>
<td>Almost All</td>
<td>Static</td>
<td>High</td>
</tr>
</tbody>
</table>
## Static Typing vs. Dynamic Typing?

<table>
<thead>
<tr>
<th>language</th>
<th>runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>C GNU gcc</td>
<td>1.08</td>
</tr>
<tr>
<td>C++ GNU g++</td>
<td>1.09</td>
</tr>
<tr>
<td>Ada 2005 GNAT</td>
<td>1.34</td>
</tr>
<tr>
<td>Java 6 -server</td>
<td>1.59</td>
</tr>
<tr>
<td>Scala</td>
<td>2.06</td>
</tr>
<tr>
<td>Fortran Intel</td>
<td>2.19</td>
</tr>
<tr>
<td>Pascal Free Pascal</td>
<td>2.35</td>
</tr>
<tr>
<td>Haskell GHC</td>
<td>2.48</td>
</tr>
<tr>
<td>C# Mono</td>
<td>2.5</td>
</tr>
<tr>
<td>OCaml</td>
<td>3.03</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>language</th>
<th>runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>F# Mono</td>
<td>3.22</td>
</tr>
<tr>
<td>Lisp SBCL</td>
<td>3.87</td>
</tr>
<tr>
<td>Go 6g 8g</td>
<td>4.53</td>
</tr>
<tr>
<td>Clojure</td>
<td>10.81</td>
</tr>
<tr>
<td>Erlang HiPE</td>
<td>12.86</td>
</tr>
<tr>
<td>Ruby JRuby</td>
<td>45.71</td>
</tr>
<tr>
<td>Python CPython</td>
<td>46.5</td>
</tr>
<tr>
<td>Python 3</td>
<td>49.58</td>
</tr>
<tr>
<td>Ruby 1.9</td>
<td>63.78</td>
</tr>
<tr>
<td>Perl</td>
<td>64.81</td>
</tr>
</tbody>
</table>

[http://shootout.alioth.debian.org/](http://shootout.alioth.debian.org/)
Language Features for Building DSLs

• Operator Overloading and Infix Notation
• Type Inference
• Type Alias
• First-Class Functions and Closures
• Functional Programming
  – Immutable variables, iterator methods, higher order functions, currying and partial function applications
• Default Arguments
• Parser Combinator Library
ScalaTion

• Simulation system coded in Scala
  – Since a design goal of Scala is to facilitate the construction of DSLs
• Utilizes or recodes some modules of JSIM
  – Portions of the 50 Kloc JSIM code-base were recoded with approx. 80% reduction in loc
• Supports the modeling paradigms of the Discrete-event Modeling Ontology (DeMO)
  – Event, process, activity and state
case class Arrival (customer: Entity)
extends Event (protoArr, customer, 
alinks, this, Array (150., 200., 50., 50.)) {
  override def occur () {
    super.occur () // handle casual links
    nArr += 1 // update the current state
    nIn += 1
  } // occur
} // Arrival class

case class Departure (customer: Entity)
extends Event (protoDep, customer, dLinks, 
this, Array (450., 200., 50., 50.)) {
  override def occur () {
    super.occur () // handle casual links
    nIn -= 1 // update the current state
    nOut += 1
  } // occur
} // Departure class
case class Customer () extends SimActor ("c", this) {
  def act () {
    entry2tellerQ.move ()
    if (teller.busy) tellerQ.waitIn ()
    teller.utilize ()
    teller.release ()
    teller2door.move ()
    door.leave ()
  } // act
} // Customer
Conclusions

• Narrowing the gap between model and program
• Using an embedded Domain Specific Language (DSL) rather than a General Purpose Language (GPL) or Simulation Programming Language (SPL)
• ScalaTion prototype looks promising – needs further development and testing
• Other new statically-typed functional object-oriented languages may be suitable as well (e.g., F#)
Future Work

• **scalation.dynamics**: add an integrator more suitable for stiff systems and extend our LinearDiffEq class to handle complex eigenvalues.

• **scalation.optimization**: add simplex method, quadratic programming, steepest descent, conjugate gradient and quasi-newton.

• **scalation.scala3d**: add 3D animation package that interacts with Java OpenGL (JOGL)
Future Work (cont.)

• Adding Unicode Support
  – Use of Greek Symbols
    ```scala
    case class Normal (µ: Double, σ: Double, ψ: Int)
    ```
  – Use of Math Symbols
    ```scala
    def ≤ (y: T): Boolean = x <= y
    def ∙ (x: T, y: T): T = x * y
    def Σ (v: VectorN [T]): T = v.foldLeft (0) (_ + _)  
    ```