Proceedings of the 2010 Winter Simulation Conference
B. Johansson, S. Jain, J. Montoya-Torres, J. Hugan, and E. Yücesan, eds.

USING DOMAIN SPECIFIC LANGUAGE FOR MODELING AND SIMULATION: SCALATION AS A CASE STUDY

John A. Miller
Jun Han
Maria Hybinette

Department of Computer Science
University of Georgia
Athens, GA, 30602, USA

ABSTRACT

Progress in programming paradigms and languages has over time influenced the way that simulation programs are written. Modern object-oriented, functional programming languages are expressive enough to define embedded Domain Specific Languages (DSLs). The Scala programming language is used to implement ScalaTion that supports several popular simulation modeling paradigms. As a case study, ScalaTion is used to consider how language features of object-oriented, functional programming languages and Scala in particular can be used to write simulation programs that are clear, concise and intuitive to simulation modelers. The dichotomy between "model specification" and "simulation program" is also considered both historically and in light of the potential narrowing of the gap afforded by embedded DSLs.

1 INTRODUCTION

As one learns simulation the importance of the distinction between "model specification" and "simulation program" is made clear. In the initial period (Nance 1996), the distinction was indeed important as models were expressed in a combination of natural language (e.g., English) and mathematics, while the simulation programs implementing the models were written in Fortran. The gap was huge. Over time, the gap has narrowed through the use of more modern general-purpose programming languages (GPLs) with improved readability and conciseness.

Besides advances in general-purpose languages, the developers of Simulation Programming Languages (SPLs) have made major contributions. Some of the better known SPLs include GPSS, SLAM, Siman, Simscript and Simula. These SPLs allow models to be implemented concisely, clearly and in a fashion in which the implementation's connection to the model is not obscured by low-level, machine-oriented details that are largely irrelevant from the point of view of modeling. Closing the gap allows simulation programs to be developed more quickly and reliably. SPLs do, however, have the following weaknesses:

- SPLs usually execute more slowly than GPLs.
- Simulation programs often require some conventional coding for which SPLs were not designed, making such coding more difficult. Note, Simscript and Simula provide many of the features of GPLs.
- SPLs do not have the comprehensive and convenient libraries that modern object-oriented GPLs, such as Java, have.

Clearly, there are significant advantages as well as disadvantages to writing simulation programs in either SPLs or GPLs. The purpose of this paper is not to rehash this debate, but to examine a third possibility: use a Domain Specific Language (DSL). Specifically, an embedded or internally defined DSL, which is a language customized to a domain via extensibility features provided by some modern GPLs.

The notion of DSLs (van Deursen, Klint, and Visser 2000) began long ago, and under a broad definition could include special-purpose programming languages of which SPLs would be categorized. In this paper, we are focusing on embedded or internally-defined DSLs. These embedded DSLs are not simply a subroutine library or class library, but rather make full use of syntactic features to manifest a sublanguage for a particular domain that provides greater expressive power and readability for that domain. One category of languages is particularly adept at allowing a DSL to be defined, namely object-oriented, functional programming languages. They tend to be more concise and expressive than other languages and are rapidly increasing in popularity. Using a fairly broad view of functional languages, the following seven languages may be considered to be object-oriented, functional programming languages. In some cases, it might be more appropriate to refer them as object-oriented languages with functional features.

- OCaml (Smith 2006) developed by Didier Remy
- F# (Syme, Granicz, and Cisternino 2007) developed Don Syme
- Scala (Odersky, Spoon, and Venners 2008) developed by Martin Odersky
- Python (Watters, Ahlstrom, and Van Rossum 1996) developed by Guido van Rossum
- Ruby (Thomas and Hunt 2000) developed by Yukihiro Matsumoto
- Groovy (Koenig et al. 2007) developed by Guillaume Laforge
- C# (Hejlsberg, Wiltamuth, and Golde 2003) developed by Anders Hejlsberg

A reference to a first popular book on the language as well as the name of the main developer are also given. Note, Java (Arnold and Gosling 1996) and C++ (Stroustrup 1986) have proposals to add the closure feature which is a first step toward providing some functional programming capabilities. Also, it should be mentioned that functional programming languages, which began before object-oriented, functional programming languages, were introduced with Lisp (McCarthy 1960). Today there are a large number of functional programming languages including two of major importance: Standard ML (Milner 1984) and Haskell (Hudak et al. 1992). These languages are also very concise, but possibly due to their unfamiliar syntax and the fact that are not ideal for programming tasks that are best solved by modifying existing data, they have yet to develop a large user base. It is difficult to establish a bright line to determine when a language should be considered functional. For example, Scala supports most of the features of functional languages, including functions as first class values, closures, list comprehensions, curried functions, lazy evaluation and pattern matching (Odersky 2010), although in some cases less elegantly than languages such as Standard ML, Haskell or OCaml. Most would agree that the first two languages in the list are functional, and many of those would agree that Scala is functional, while below Scala some would consider it a stretch.

Although non-scientific it is interesting to look into the issues of conciseness, since many studies indicate that the number of lines of code that a typical programmer can write is essentially language invariant, greater conciseness leads to greater productivity. One place to looks is (Rosetta-Code 2010) where code to compute a powerset (set of all subsets) is given. The functional (including object-oriented, functional) languages typically require very few lines of code (e.g., 1 for Scala, Standard ML, OCaml and Haskell vs. many lines of code for C, C++ and Java).

One issue among the current crop of object-oriented, functional programming languages that is relevant to modeling and simulation is the issues of typing: OCaml, F#, Scala and C# are statically typed, while Python, Ruby and Groovy are dynamically typed. Dynamic typing has the following advantages: more flexible coding, reduction of type annotation (and hence more conciseness), while static typing has its own advantages: catches typing errors earlier (during compilation), runs faster (up to an order of magnitude) and with type inference can still provide conciseness. The speed of execution is particularly salient for the domain of modeling and simulation.

Recently, there have been several projects to develop DSLs using Python, Ruby and Scala. As a case study to explore the usefulness of creating DSLs for Modeling and Simulation (M&S), we have chosen Scala as the implementation language. As indicated in the last paragraph, it is well suited for developing DSLs and because it is statically typed, its execution performance is good. Scala generates Java byte code. It is marginally slower than Java, because in some cases it utilizes Java reflection capabilities. (Fulgham 2010) has a rough comparison of running times.

The rest of this paper is organized as follows: In Section 2, we discuss various types of DSLs and then cover the suitability of Scala for developing a DSL for modeling and simulation. Section 3 presents ScalaTion as a case study for developing and using a DSL for M&S. Finally, conclusions and future work are given in Section 4.

2 DOMAIN SPECIFIC LANGUAGES

A Domain Specific Language (DSL) may be defined as follows: "A domain-specific language (DSL) 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" (van Deursen, Klint, and Visser 2000). The key advantages may be summarized as follows: "DSLs trade 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" (Mernik, Heering, and Sloane 2005).

The creation and use of Domain Specific Languages (DSLs) has recently gained popularity, since there are now several relatively new programming languages, including Python, Ruby and Scala, which make it easier to develop DSLs (Wampler and Payne 2009). Use of a DSL allows one to hide the implementation details and improve the readability of the code for those knowledgeable of the application domain. In some cases, the code can look almost like mathematics, while in other cases almost like English. Such code is less imperative and more declarative as well as being more concise.

2.1 Externally vs. Internally Defined DSLs

In the past, a DSL was often created using language processing tools such as parsers and pre-processors to translate code to a host language. One could also write a full interpreter or compiler for an external DSL. This approach has the advantage of putting fewer constraints on the language designer. Unfortunately, more work is required to make such a DSL and too much design freedom may lead to a longer learning curve for new users of the language, since it may not follow familiar patterns.

The new trend is to use features of the language itself to define the DSL. This is referred to as an internally defined or embedded DSL (van Deursen, Klint, and Visser 2000; Zdun and Strembeck 2009). Several modern general-purpose programming languages have powerful and flexible syntax that allows customization to a particular domain, so that solutions can be expressed concisely and naturally to practitioners in that domain.

2.2 DSLs in Scala

One of the design goals of Scala was to support the development of Domain Specific Languages. There is a growing list of projects doing just that. A couple of related examples are ScalaQL (Spiewak and Zhao 2010) and ScalaDBC (Dubochet 2006), both of which use Scala language constructs to provide SQL-like languages, much like C#'s LINQ. The Kiama project (Sloane 2008) created language processing tools by developing DSLs embedded in Scala. Parser combinators were used to create a DSL with an EBNF-like notation for specifying grammars (Moors, Piessens, and Odersky 2008). Finally, DSLs are integral enough to Scala to merit their own chapter in an on-line Scala textbook (Wampler and Payne 2009).

The following language features of Scala enhance one's ability to create effective DSLs.

- Operator Overloading and Infix Notation. Several modern languages support operator overloading. It was left out of Java because of some complexities associated with it in C++, e.g., overloading of (). In Scala, operator overloading is available as a consequence of two other language features: (i) infix notation and (ii) flexible method naming including operator symbols (Odersky, Spoon, and Venners 2008). For mathematical expressions, operator overloading can greatly improve both conciseness and readability (e.g., given three matrices `a`, `b` and `c`, `a * b + c` is more readable than `a.multiply(b).add(c)`). Infix notation is also very useful in a similar way for methods whose names are not operator symbols. For example, since `*` is used for vector multiply (the Hadamard product), `dot` is used for the inner product and `outer` is used for the outer product (e.g., `u dot v` is easier to read than `u.dot(v)`).
- Type Inference. Strongly and statically typed languages like Java help programmers discover mistakes earlier, so that they are much easier to find and fix. Dynamically typed languages like Python and Ruby avoid repetitious type annotations at the cost doing type checking at run-time, which can slow down execution as well as delay the discovery of mistakes. Type inference as provided by many functional as well as some object-oriented, functional programming languages (e.g., Scala) are to a degree, the best of

both worlds. They sacrifice a little flexibility (so long as the type system is powerful enough) for the gain of static type checking.

- Type Alias. Because Scala has a rich type system, some type expressions may become long. A type alias can be used to provide a convenient shorthand (e.g., `type MatrixD = MatrixN [Double]`).
- First-Class Functions and Closures. In functional programming languages, functions are the central feature of the language. In object-oriented languages like Java, methods are used to specify functions, but there are no functions defined outside of classes, no function types, no function variables and no parameters of functional type. Scala supports first-class functions, meaning that functions can be viewed as objects that can be assigned to function variables or passed as functional arguments. Function types (e.g., `type AggregateFunction = (Set [Double]) => Double`) can also be defined. A closure may be thought of as a function, that when defined captures its local environment so that values of variables in its local scope can be accessed, when the function is invoked. This is very useful in modeling and simulation, since it makes it easy to pass functions as arguments, e.g., to a Runge-Kutta differential equation numerical integrator. It also makes it easy to define a vector of derivative functions for solving systems of differential equations. Closures are a very powerful feature that can also be used to create custom control structures and generally allow for the creation of very concise programming solutions.
- Functional Programming. As a functional programming language, closures are just the starting point for Scala. Beyond closures, Scala supports immutable variables, iterator methods, higher order functions, currying and partial function applications. In ScalaTion, variables are immutable (val) unless this results in awkward or inefficient code, in which case mutable (var) variables are used. Wide use of immutable variables typically makes debugging easier. The `foreach` iterator method, allows one to easily traverse through data structures, e.g., the `foreach` method in `MatrixN` allows one to iterate over the matrix, row vector by row vector, for retrieval or other function applications. Higher order functions, currying and partial function applications allows functions to be partially evaluated, i.e., some of its parameters are passed in, but not all, so a function with fewer parameters is defined that later can be called passing it the remaining parameters. This can make code more concise, since new functions can be built from existing ones.
- Default Arguments. Scala provides named and default arguments. Use of default values for arguments can simplify the use an Application Programming Interface (API) as well as allow methods to be unified leading to additional simplification. Default arguments are widely used in ScalaTion.
- Parser Combinator Library. Some functional programming languages provide an easier why to develop parsers, particularly for small, experimental languages. The idea is to create a parser to recognize or consume a portion of the language and then combine parsers to process the entire language. For example, `def term: Parser [Any] = factor rep ("*" factor | "/" factor)` can be used to consume terms in arithmetic expressions. Adding similar definitions for `expr` and `factor` provides a complete parser for an arithmetic expression language using very few lines of code (Odersky, Spoon, and Venners 2008).

3 ScalaTion: A DSL FOR M&S

An early version of ScalaTion is available on the Web (Miller 2009). It contains over 12 thousand lines of Scala code. ScalaTion is the fourth simulation package that we have developed and builds on the other three.

- SIMODULA (Miller and Weyrich 1989), coded in Modula-2
- JSIM (Nair, Miller, and Zhang 1996; Miller et al. 1997), coded in Java
- SASSY (Hybinette et al. 2006), coded in Java

Some of the 50 thousand lines in the codebase of JSIM served as a basis for parts of ScalaTion.

The package structure of ScalaTion reflects the organization of the Discrete-event Modeling Ontology (DeMO) (Miller et al. 2004). This ontology is based on the three traditional simulation world-views: event-scheduling, process-iteration and activity-scanning. In addition, it has a fourth major category for state oriented models. ScalaTion currently consists of nine packages, starting with a small package of utilities (`scalation.util`) that provides some generally useful functionality. In the subsections below, each of the major eight packages will be discussed,

from low to high level reflecting the package dependencies. The last four packages correspond to the subclasses of DeMO's `DeModel` class.

3.1 `scalation.scala2d` Package

The `scalation.scala2d` package provides access to Java 2D including much of the functionality of `java.awt` and `java.awt.geom` so that packages using it can avoid directly importing Java. It also provides several additional shapes including triangles, quads, squares, pentagons, hexagons, octagons and polygons. Further, it provides `Arrows` (`Line2D` with an arrowhead), `QCurves` (`QuadCurve2D` supplemented with parametric traversal useful for animation) and `QArcs` (`QCurve` with an arrowhead).

3.2 `scalation.mathstat` Package

The `scalation.mathstat` package provides mathematical and statistical traits, objects and functions that are useful for modeling and simulation. These include generic implementations for vectors and matrices. The ability to use operator overloading and infix notation allows for expressions to be written almost as one would see in a textbook. For example, the code shown in Listing. 1 performs matrix multiplication using the vector `dot` product.

Listing 1: Matrix Multiplication.

```
def * (b: MatrixN [T]) (implicit nu: Numeric [T]): MatrixN [T] = {
    val c = MatrixN [T] (dim1, b.dim2)
    for (i <- c.range1; j <- c.range2) c(i, j) = row(i) dot b.col(j)
    c
} // *
```

Note, `[T]` is a type parameter. The `MatrixN` class works on any `Numeric` data type. As a longer example, Listing 2 shows the primary method inside the `Regression` class. The `fit` method uses the least squares method to fit the parameter vector (b) in the multiple regression equation $y = Xb + \varepsilon$. Under least squares, $b = (X^t X)^{-1} X^t y$. Scala allows the code to look very similar to the mathematical expression. Note, `MatrixD` is simply a shorthand type alias for `MatrixN [Double]`.

Listing 2: Multiple Regression.

```
def fit (x: MatrixD, y: VectorD): Tuple2 [VectorD, Double] = {
    val b    = (x.t * x).inverse * x.t * y // parameter vector
    val e    = y - x * b                   // error vector
    val sse  = e.norm2                     // sum of squared errors
    val ssto = y.norm2 - (pow (y.sum, 2)) / y.dim
    val r2   = (ssto - sse) / ssto         // coefficient of determination
    Tuple2 (b, r2)
} // fit
```

On the mathematical side, this package also provides a trait for combinatorial functions, a class for complex numbers and classes for computing eigenvalues/eigenvectors, On the statistical side, this package also provides a random number generator, 26 random variate generators (e.g., `Exponential` and `Normal`), statistical quantiles, sample and time-persistent statistics as well basic plotting capabilities.

3.3 `scalation.animation` Package

The `scalation.animation` package is predicated on the assumption that most simulation models can be represented as or transformed to a graph. For example, event graphs are naturally graphs where the nodes represent types of events and the edges represent causal links between the events. Since ScalaTion supports many different types of simulation models, the plan was to build one general purpose animation engine that works for all the models. The scalation 2D animation engine provides an animation command interface for all of the simulation engines to use. Being graphically oriented, it includes commands such as CreateNode, MoveNode, DestroyNode. There are also similar commands for edges and tokens. Tokens may move in the graph along nodes and edges as

well as freely move about the drawing canvas. The animation engine works by processing animation commands in time order from a queue shared with the simulation engine. It will sleep according to the time difference between timestamped animation commands, wake up, change the animation state according to the command and then call `paintComponent`.

3.4 `scalation.dynamics` Package

The `scalation.dynamics` package is still a work in progress. It currently supports models represented as a system of Ordinary Differential Equations (ODEs). It provides a solver for first-order, constant coefficient, linear ODEs and a numerical integrator for more general ODEs. The solver is dependent on the `scalation.mathstat` package for finding eigenvalues and eigenvectors. A framework for including multiple integrators is set up, which currently contains the following numerical integrators: a 4th-order Runge-Kutta and a (4,5)th-order Dormand-Prince. As this package is further developed, we plan to create a general-purpose simulation engine for systems dynamics/continuous simulation. Now it provides basic capabilities and functionality for hybrid discrete-continuous models, such as Hybrid Functional Petri Nets.

As stated earlier, the four remaining packages to be discussed correspond the major subclasses of DeMO's Discrete-event Model (DeModel) class: ActivityModel, EventModel, ProcessModel and StateModel. Simple example models are given for each of the four.

3.5 `scalation.activity` Package

In DeMO, activity oriented models are defined as models that are organized around the notion of activity. An activity is something that an entity in a simulation does that has a beginning (start event) and an end (end event). One could visualize such models as having entities/tokens that flow through a graph. Activities or transitions are presented by certain nodes in the graph. For example, a Petri Net can be represented by a bipartite graph consisting of two types of nodes: transitions and places. Tokens accumulate at place nodes and when enough to fire a transition have accumulated at each incoming place, the transition fires, moving the tokens to output places. The `scalation.activity` simulation engine provides very general types of Petri Nets including Hybrid Functional Petri Nets.

A simple example with just one transition that models a biochemical reaction is shown in Listing 3. It represents one step in a biochemical pathway simulation in which enzymes control a pathway that synthesizes glycans (a form of carbohydrate). Fluids (the continuous analog of tokens) of certain colors reside in places, while reactions via a Petri Net transition move the fluids from input to output places. The Petri Net is created by defining the colors, places and transitions and then connecting the places and transitions via arcs. The amount of fluid flow is determined by differential equations; an array of derivative functions is passed in for this calculation.

Listing 3: Petri Net Model of Reaction.

```
object Reaction extends Application {
    // green for glycans, blue for enzymes
    val hues  = Array [Color] (green, blue)
    val place = Array [PlaceD] ( new PlaceD (100, 250, new VectorD (20., 0.)),
            new PlaceD (200, 350, new VectorD ( 0., 10.)),
            new PlaceD (500, 250, new VectorD ( 0.,  0.)))
    val tran = Array [Transition] (new Transition (300, 240,
                                       Deterministic (4), hues))
    val pnet = new PetriNet (hues, place, tran)

    def derv1 (t: Double, y: Double) = .1 * y //derivatives governing inflow
    def derv2 (t: Double, y: Double) = 1.

    tran(0).connect (pnet,
        Array [ArcD] ( new ArcD (place(0), tran(0), true,
          new VectorD (0., 0.), null, Array [Derivative] (derv1, derv2)),
          new ArcD (place(1), tran(0), true, new VectorD (0., 10.))),
```

```
        Array [ArcD] ( new ArcD (place(1), tran(0), false,
          new VectorD (0., 10.)),
          new ArcD (place(2), tran(0), false, new VectorD (10., 0.))))

    println (pnet)
    pnet.simulate (2, 20)
} // Reaction object
```

A fully commented version of the code shown in Listing 3 is `examples/activity/Reaction.scala` of (Miller 2009). A screenshot of an animation of a Hybrid Functional Petri Net modeling a portion of a biochemical pathway is shown in (Silver et al. 2009).

3.6 `scalation.event` Package

The simpliest way to develop a simulation engine to support discrete-event simulation is to implement event-scheduling. Such an engine can be organized along the lines of Event Graphs (Schruben 1983). This involves creating the following four classes: `Event`, `Entity`, `CausalLink` and `Model`. An `Event`, defined as an instantaneous occurrence that can trigger other events and/or change the state of the simulation, is represented as a node in an event graph. An `Entity`, such as a customer in a bank, flows through the simulation. A `CausalLink` emanating from an event/node is represented as an outgoing edge in the event graph. The `Model` serves as a container/controller for the whole simulation.

For example, to create a simple bank simulation, one could use the four classes defined in the event-scheduling engine to create subclasses of `Event`, called `ArrivalEvent` and `DepartureEvent`, and one subclass of `Model`, called `BankModel`. In more complex situations, one would typically define a subclass or instances of `Entity` to represent the customers in the bank. The Scala code shown in Listing 4 was made more declarative than typical code for event-scheduling to better mirror event graph specifications, where the causal links specify the conditions and time delays. For instance, `() => nArr < nArrival` is a closure returning `Boolean` that will be executed when arrival events are handled. In this case, it represents a stopping rule; when the number of arrivals exceeds a threshold, the arrival event will no longer schedule the next arrival. The `serviceRV` is a random variate to be used for computing service times. The main thing to write within the subclasses of `Event` is the `occur` method. If an event can trigger other events, it must first call `super.occur ()` to schedule these future events. It then can make changes to the state variables, in this case `nArr`, `nIn` and `nOut`. For animation of the event graph, a prototype for each type of event is created and displayed as a node. The edges connecting these prototypes represent the casual links.

Listing 4: Event Graph Model of a Bank.

```
class BankModel (name: String, nArrival: Int, arrivalRV: Variate,
    serviceRV: Variate) extends Model (name) {
    var nArr = 0.  // no. of customers that have arrived
    var nIn  = 0.  // no. of customers in the bank
    var nOut = 0.  // no. of customers that have finished and left the bank

    val protoArr = Arrival (null)    // prototype for all Arrival events
    val protoDep = Departure (null)  // prototype for all Departure events

    val aLinks = Array (CausalLink ("link2A", this,
        () => nArr < nArrival, protoArr, () => Arrival (null), arrivalRV),
                        CausalLink ("link2D", this,
        () => nIn == 0, protoDep, () => Departure (null), serviceRV))
    val dLinks = Array (CausalLink ("link2D", this,
        () => nIn > 1, protoDep, () => Departure (null), serviceRV))
    protoArr.displayLinks (aLinks)
    protoDep.displayLinks (dLinks)

    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

    simulate (0., Arrival (null))
} // BankModel class
```

A fully commented version of the code shown in Listing 4 is `examples/event/Bank.scala` of (Miller 2009). A screenshot of animation of the event graph for this model is shown in Figure. 1. Each node in the graph will have a token (black dot) if the corresponding event is occurring or scheduled.

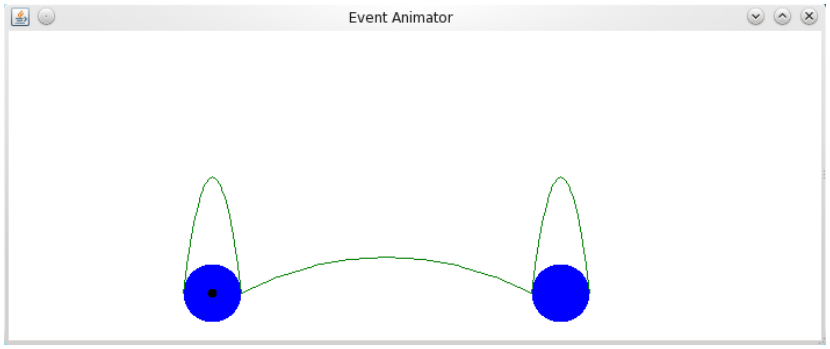

Figure 1: Event Graph Animation of a Bank.

3.7 `scalation.process` Package

Many discrete-event simulation models are written using the process-interaction world view, because the code tends to be concise and intuitively easy to understand. Take for example the process-interaction model of a bank shown in Listing 5. Following this world view, one simply constructs the simulation components and then provides a script for entities (`SimActors`) to follow while in the system. In this case the `act` method for the customer class provides the script (what entities should do), i.e., enter the bank, if the tellers are busy wait in the queue, then receive service and finally leave the bank.

Listing 5: Process-Interaction Model of a Bank.

```
class BankModel (name: String, nArrival: Int, iArrivalRV: Variate,
  nUnits: Int, serviceRV: Variate, moveRV: Variate)
  extends Model (name) {
```

```
  val entry         = new Source ("entry", this, Customer, nArrival,
                                  iArrivalRV, Array (100, 100, 30, 30))
  val tellerQ       = new WaitQueue ("tellerQ", Array (210, 100, 80, 30))
  val teller        = new Resource ("teller", tellerQ, nUnits, serviceRV,
                                    Array (290, 100, 30, 30))
  val door          = new Sink ("door", Array (400, 100, 30, 30))
  val entry2tellerQ = new Transport ("entry2tellerQ",
                                    moveRV, entry, tellerQ)
  val teller2door   = new Transport ("teller2door", moveRV, teller, door)

  addComponents (
        List (entry, tellerQ, teller, door, entry2tellerQ, teller2door))

  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

  startSim ()
} // BankModel class
```

Currently, ScalaTion includes several types of model components: `Junction`, `Resource`, `Sink`, `Source`, `Transport` and `WaitQueue`. Sources produce entities, while sinks consume them. Resources provide services to entities (typically resulting in some delay). Often, a resource is fronted by a waiting queue that serves as a holding area for entities. Transports are used to move entities from one component to the next, while junctions are used to connect transports. A screenshot of the animation of the bank model is shown in Figure. 2.

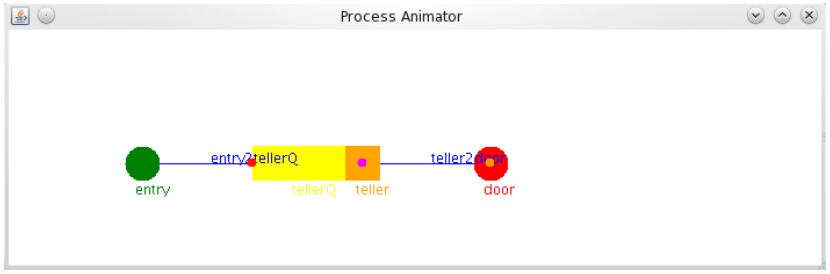

Figure 2: Process-Interaction Animation of a Bank.

Scala provides very little overhead in expressing the model. It is necessary to indicate the components and how they are connected, as well as, the script for entities to follow. Even if a Simulation Programming Language (SPL) is used, it would be difficult to provide a significantly more concise and intuitive specification. JSIM provides a similar form, but it is less concise and less elegant. The two codes can be compared between `examples/bank2/Bank.java` of (Miller et al. 1997) and `examples/process/Bank.scala` of (Miller 2009). The Scala language also made it easier to write the process-interaction simulation engine, since it provides the Actor model for concurrent programming; whereas, in JSIM lower-level threads were utilized. Note, SIMODULA used Modula-2's coroutines, which are ideally suited for implementing efficient process-interaction simulation engines.

3.8 `scalation.state` Package

Although not based on one of the traditional three simulation world views, state oriented models can be useful for modeling and simulation. The emphasis of such models are states and state transitions. Classical examples are Discrete-time and Continuous-time Markov Chains. The `scalation.state` package provides for the simulation and animation of sample paths as well as steady-state and transient-state solutions (i.e., probability vectors indicating the probability of being in a particular state). The vector and matrix classes in `scalation.mathstat` make it straightforward to find such solutions. For example, the following method shown in Listing 6 provides a steady-state solution for a Continuous-time Markov Chain (CTMC).

Listing 6: Limiting Probabilities for CTMC.

```
def limit: VectorD = {
    tr.t.slice (0, tr.dim1 - 1).nullspace.normalize
} // limit
```

The object `tr` is the transition matrix, `t` transposes the matrix, `slice` eliminates the last row (it is redundant), `nullspace` computes a vector in the nullspace of the matrix, and finally `normalize` normalizes the vector so that its elements add to one.

4 CONCLUSIONS AND FUTURE WORK

This paper considers two issues in the development of simulations: (i) narrowing the gap between model and program and (ii) using an embedded Domain Specific Language (DSL) rather than a General Purpose Language (GPL) or Simulation Programming Language (SPL).

On the first issue, we believe that narrowing the gap is beneficial, because it makes the code more readable, makes the mapping between the model and the code more obvious and reduces the development time. Less code is required and it is written in a form that is familiar to those working in the domain. However, completely eliminating the gap could be counter-productive in that it could reduce creativity, slow down innovation and make model specification more cumbersome. An appropriately balanced view might be that as modeling techniques advance, the DSL used for implementing the models should follow, but never catch.

On the second issue, it will be difficult for a GPL to keep up with advancing modeling methodologies. For example, many simulations are written in C, a language that has changed little in almost forty years. DSLs can be updated and extended more readily than whole programming languages. SPLs share many of the advantages of DSLs, particularly in terms of being high-level and oriented toward a domain (e.g., manufacturing simulations). However, as mentioned earlier, there are certain limitations to SPLs, such as limitations in language constructs and the requirement to learn a new language. For an embedded DSL, these problems are reduced, since one can always rely on the parent language (e.g., Scala for ScalaTion).

For further information about ScalaTion, including download instructions is hosted in the Scalation Web page (Miller 2009).

The following items are on our agenda for future work:

- We are in the process of improving our `scalation.dynamics` package by adding an integrator more suitable for stiff systems, as well as extending our `LinearDiffEq` class to handle eigenvalues that are complex numbers.
- We have started to investigate Scala's existing support of Unicode for adding Greek letters (e.g., `Normal(`μ`,` σ`)`) as well as operators that look more like their mathematical counterparts (e.g., `def` $\leq$ `(y: T): Boolean = x <= y`), see (Scala-Forum 2009). Note, an interesting language that makes full use of Unicode is Fortress (Allen et al. 2007).
- We hope to add a package for 3D animation that interfaces with Java OpenGL (JOGL).

REFERENCES

Allen, E., D. Chase, C. Flood, V. Luchangco, J. Maessen, S. Ryu, and G. L. Steele, Jr. 2007. Project Fortress: A Multicore Language for Multicore Processors. *Linux Magazine* 38 (43).

Arnold, K., and J. Gosling. 1996. *The Java Programming Language*. Reading, Massachusetts, USA: Addison-Wesley.

Dubochet, G. 2006. On Embedding Domain-specific Languages with User-friendly Syntax. In *Proceedings of the 1st Workshop on Domain-Specific Program Development*, 19–22. Nantes, France.

Fulgham, B. 2010. The Computer Language Benchmarks Game. Available via: <shootout.alioth.debian.org>.

Hejlsberg, A., S. Wiltamuth, and P. Golde. 2003. *The C# Programming Language.* Reading, Massachusetts, USA: Addison-Wesley.

Hudak, P., S. P. Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. 1992. Report on the Programming Language Haskell: A Non-strict, Purely Functional Language version 1.2. *ACM SIGPLAN Notice* 27 (5): 1–164.

Hybinette, M., E. Kraemer, Y. Xiong, G. Matthews, and J. Ahmed. 2006. SASSY: A Design for a Scalable Agent-based Simulation System using a Distributed Discrete Event Infrastructure. In *WSC '06: Proceedings of the 38th Conference on Winter Simulation*, ed. L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, 926–933. Monterey, California, USA: IEEE, Inc.

Koenig, D., A. Glover, P. King, G. Laforge, and J. Skeet. 2007. *Groovy in Action.* Greenwich, Connecticut, USA: Manning Publications Co.

McCarthy, J. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine. *Communications of the ACM* 3 (4): 184–195.

Mernik, M., J. Heering, and A. M. Sloane. 2005. When and How to Develop Domain-specific Languages. *ACM Computing Surveys (CSUR)* 37 (4): 316–344.

Miller, J. A. 2009. Scalation Project. Available via: <www.cs.uga.edu/~jam/scalation>.

Miller, J. A., G. T. Baramidze, A. P. Sheth, and P. A. Fishwick. 2004. Investigating Ontologies for Simulation Modeling. In *ANSS '04: Proceedings of the 37th Annual Symposium on Simulation*, 55–71. Washington, DC, USA: IEEE, Inc. Available via: <www.cs.uga.edu/~jam/DeMO>.

Miller, J. A., R. S. Nair, Z. Zhang, and H. Zhao. 1997. JSIM: A JAVA-Based Simulation and Animation Environment. In *ANSS '97: Proceedings of the 30th Annual Simulation Symposium*, 31 – 42. Washington, DC, USA: IEEE, Inc. Available via: <www.cs.uga.edu/~jam/jsim/>.

Miller, J. A., and O. R. Weyrich, Jr. 1989. Query Driven Simulation using SIMODULA. In *ANSS '89: Proceedings of the 22nd Annual Symposium on Simulation*, 167–182. Los Alamitos, California, USA: IEEE, Inc.

Milner, R. 1984. A Proposal for Standard ML. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, 184–197. New York, NY, USA: ACM.

Moors, A., F. Piessens, and M. Odersky. 2008. Parser Combinators in Scala. Technical Report CW491, Department of Computer Science, Katholieke Universiteit Leuven.

Nair, R. S., J. A. Miller, and Z. Zhang. 1996. Java-based Query Driven Simulation Environment. In *WSC '96: Proceedings of the 28th Conference on Winter Simulation*, ed. J. M. Charnes, D. J. Morrice, D. T. Brunner, and J. J. Swain, 786–793. Piscataway, New Jersey: IEEE, Inc.

Nance, R. E. 1996. A History of Discrete Event Simulation Programming Languages. In *History of Programming Languages—II*, 369–427. New York, NY, USA: ACM.

Odersky, M. 2010. A Postfunctional Language. Available via: <www.scala-lang.org/node/4960>.

Odersky, M., L. Spoon, and B. Venners. 2008. *Programming in Scala: A Comprehensive Step-by-step Guide*. 1st ed. Mountain View, California, USA: Artima Press.

Rosetta-Code 2010. Power Set. Available via: <rosettacode.org/wiki/Power_set>.

Scala-Forum 2009. Aliasing <= Method to ≤ (\u2264) for All Ordered[T]. Available via: <www.scala-lang.org/node/4691>.

Schruben, L. 1983. Simulation Modeling with Event Graphs. *Communications of the ACM* 26 (11): 957–963.

Silver, G. A., K. R. Bellipady, J. A. Miller, W. S. York, and K. J. Kochut. 2009. Supporting Interoperability Using the Discrete-event Modeling Ontology (DeMO). In *WSC '09: Proceedings of the 41th Conference on Winter Simulation*, ed. M. D. Rossetti, R. R. Hill, B. Johansson, A. Dunkin, and R. G. Ingalls, 1399 – 1410. Austin, Texas, USA: IEEE, Inc.

Sloane, T. 2008. Experiences with Domain-specific Language Embedding in Scala. In *Proceedings of the 2nd International Workshop on Domain-Specific Program Development (DSDP)*, ed. J. Lawall and L. Réveillère, 7. Nashville, Tennessee, USA.

Smith, J. B. 2006. *Practical OCaml.* Berkeley, California, USA: Apress.

Spiewak, D., and T. Zhao. 2010. ScalaQL: Language-Integrated Database Queries for Scala. In *Software Language Engineering*, ed. M. van den Brand, D. Gaevic, and J. Gray, Volume 5969 of *Lecture Notes in Computer Science*, 154–163. Springer-Verlag Berlin.

Stroustrup, B. 1986. *The C++ Programming Language.* Reading, Massachusetts, USA: Addison-Wesley.

Syme, D., A. Granicz, and A. Cisternino. 2007. *Expert F# (Expert's Voice in .Net).* Berkeley, California, USA: Apress.

Thomas, D., and A. Hunt. 2000. *Programming Ruby: The Pragmatic Programmer's Guide.* Boston, Massachusetts, USA: Addison-Wesley.

van Deursen, A., P. Klint, and J. Visser. 2000. Domain-specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices* 35 (6): 26–36.

Wampler, D., and A. Payne. 2009. *Programming Scala: Scalability = Functional Programming + Objects*. Sebastopol, California, USA: O'Reilly Media, Inc. Available via: <programming-scala.labs.oreilly.com/ch11.html>.

Watters, A., J. C. Ahlstrom, and G. Van Rossum. 1996. *Internet Programming with Python.* New York, NY, USA: Henry Holt and Co., Inc.

Zdun, U., and M. Strembeck. 2009. Reusable Architectural Decisions for DSL Design: Foundational Decisions in DSL Development. In *Proceedings of the 14th European Conference on Pattern Languages of Programs (EuroPLoP)*, 1–37. Irsee Monastery, Germany.

AUTHOR BIOGRAPHIES

JOHN A. MILLER is a Professor of Computer Science at the University of Georgia and has also been the Graduate Coordinator for the department for 9 years. His research interests include database systems, simulation, Web services and bioinformatics. Dr. Miller received the B.S. degree in Applied Mathematics from Northwestern University in 1980 and the M.S. and Ph.D. in Information and Computer Science from the Georgia Institute of Technology in 1982 and 1986, respectively. During his undergraduate education, he worked as a programmer at the Princeton Plasma Physics Laboratory. He is the author of over 150 publications in the areas of database systems (transactions, object-oriented databases and XML databases), modeling & simulation (modeling methodology, simulation environments, Web-based simulation and ontology driven simulation), workflow and Web services (discovery, composition, semantics and quality of service) and bioinformatics (workflows, databases, ontologies and simulation). Dr. Miller has been active in the organizational structures of research conferences in all these areas. He has served in positions from Track Coordinator to Publications Chair to General Chair of the following conferences: Annual Simulation Symposium (ANSS), Winter Simulation Conference (WSC), Workshop on Research Issues in Data Engineering (RIDE), NSF Workshop on Workflow and Process Automation in Information Systems, and Conference on Industrial & Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE). He is an Associate Editor for the ACM Transactions on Modeling and Computer Simulation, IEEE Transactions on Systems, Man and Cybernetics and SIMULATION: Transactions of the Society for Modeling and Simulation International as well as an Editorial Board Member for the Journal of Simulation and International Journal of Simulation and Process Modelling. In addition, he has been a Guest Editor for the International Journal in Computer Simulation and IEEE Potentials. His email address is <jam@cs.uga.edu>.

JUN HAN is a Ph.D. student in Computer Science at the University of Georgia. His research interests include Simulation, Web services and bioinformatics. Jun Han received a B.S. in Computer Science from BeiHang University and an M.S. in Computer Science from Institute of Software, Chinese Academy of Sciences. His email address is <jun@cs.uga.edu>.

MARIA HYBINETTE is a researcher in high performance and effective simulation systems. In earlier work she developed a number of methods for boosting the performance of discrete event simulations on parallel multiprocessors. Her current focus is on hybrid (both micro & macro) simulation of multi-agent behavior, e.g., in the domains of Financial Markets and Social Animal Research, and mechanisms for making them more usable. She is an associate professor in the Computer Science Department at the University of Georgia (UGA). She completed her Ph.D. at Georgia Tech and then was a research scientist also at Georgia Tech. Before joining UGA she was employed as a staff simulation & modeling engineer at the MITRE Corporation. She now directs the Distributed Simulation Laboratory (DSL) at UGA. Her email address is <maria@cs.uga.edu>.