Proceedings of the 2001 Winter Simulation Conference
B. A. Peters, J. S. Smith, D. J. Medeiros, and M. W. Rohrer, eds.

OPEN-SOURCE SML AND SILK FOR JAVA-BASED, OBJECT-ORIENTED SIMULATION

Richard A. Kilgore

SML Consortium and ThreadTec, Inc.
P. O. Box 7
Chesterfield, MO 63006, U.S.A.

ABSTRACT

Silk® and SML are software libraries of Java, C++, C# and VB.Net classes that support object-oriented, discrete-event simulation. SML™ is a new open-source or "free" software library of simulation classes that enable multi-language development of complex, yet manageable simulations through the construction of usable and reusable simulation objects. These objects are usable because they express the behavior of individual entity-threads from the system object perspective using familiar process-oriented modeling within an object-oriented design supported by a general purpose programming language. These objects are reusable because they can be easily archived, edited and assembled using professional development environments that support multi-language, cross-platform execution and a common component architecture. This introduction supports the tutorial session that describes the fundamentals of designing and creating an SML or Silk model.

1 INTRODUCTION

Silk and SML were designed to be different things to different people (SML 2001; Healy and Kilgore 1997; Kilgore, et al. 1998, 2000, 2001). To some users, the languages are a set of basic simulation class libraries that can be creatively assembled into a variety of new modeling constructs. To others, the languages are a process-oriented modeling language that offers the power and flexibility of a standard programming language. To others, they are a a visual modeling environment where modeling components can be graphically assembled to quickly create simulation applications. SML and Silk can also be a very practical tool for building object-oriented simulation components and domain-specific simulators.

These languages are not so much a simulation language in itself as they are simulation extension of the base languages. The power, flexibility and extensibility of the languages derive directly from their base language and the clean way in which the simulation-related extensions have been integrated into the language. Achieving such a high level of integration would not be possible were it not for a unique combination of features in the base languages. One is a simple yet powerful framework that greatly facilitates the implementation of object-oriented design methodology and its capabilities for creating flexible, modular, and reusable programs. Another is the built-in support for multi-threaded execution, which is essential to representing in a natural way the flow of entities in process simulation models. By incorporating the modeling environment into Silk and SML, users also have direct access to native support for browser-based execution, standard Internet communication protocols, database connectivity and graphical user interface development. Java and .NET's platform neutral design also means that models can be developed and executed on practically any combination of computer hardware and software platforms.

This paper is intended to serve primarily as an introduction to the language level features of SML and Silk which serve as the foundation for developing reusable simulation components and higher level domain-specific simulators. Other articles dealing with other important aspects of SML and the Silk language are listed in the References section. Section 2 contains an overview of creating an object-oriented model with SML and Silk. Section 3 describes the development of SML and Silk in Java Integrated Development Environments. Section 4 is an overview of JavaBean components for visual modeling and animation. Section 5 contains concluding remarks.

2 OBJECT-ORIENTED DESIGN

Object-oriented simulation is the most powerful when the user follows a consistent design pattern for object-oriented modeling in which each "intelligent" component is modeled as an independent entity class.

To illustrate this concept, consider the typical single-server queuing system of a customer served by a bank teller. There are two intelligent components in the system capable of independent thought and action, the Customer and the Teller. This system could be modeled as a single class from either a pure Customer-push or Teller-pull

perspective. But there are substantial design benefits to adherence to the proper object-oriented simulation representation of the system in which there is one Silk simulation class for each intelligent system component. The representation of the Customer and Teller classes are described in Figures 1 and 2. The numbers in brackets in the text will refer to the line numbers in these figures and color is used in electronic versions of this paper to distinguish Java keywords (blue), keywords (red), comments (green) and user-defined identifiers (black).

2.1 The Customer Class

A simulation class is like any typical Java programming class. The package of classes referenced are identified in an import statements [1] and the user-defined class name Customer is declared as an extension of the Entity class [2]. The class structure consists of the data declarations [3-11] which will define the characteristics of the simulation entities created from this class and the default Silk *process* method [12-26] that will change those entity characteristics as the state of the system changes. The essence of object-oriented simulation is the use of these Entity methods, Java statements and other objects within this process method to represent exactly what behavior that the real systems entity experiences.

Each instance of this Customer class is assigned two unique, user-defined attribute identifiers, *attArrivalTime*, *attServiceDelay* [4]. Since a simulation is a Java program, these attributes can be any Java or data type. In this case, a Java double precision variable is needed, but the service times might be an entire array of process objects that define tasks, resources required and service times (please see <http://www.threadtec.com/models> for more industrial-strength examples).

While each Customer instance will have these unique attribute identifiers, all instances of the Customer class will share common *static* class variables representing Java or simulation objects [6-11]. Only objects for random variable generation and statistics are shown in this example, but again remember that these models are Java programs so the entire collection of Java data types and objects available. For example, a more complex model

```
1.  import threadtec.silk.*;           // Silk general purpose classes

2.  public class Customer extends Entity {

3.    // Attributes (instance variables) unique to each customer
4.    double attArrivalTime, attServiceDelay;

5.    // Silk objects (class variables) common to all customers
6.    static Exponential   expInterArrivalTime = new Exponential( 10.0 ),
7.                         expServiceDelay     = new Exponential( 8.0 );
8.    static Observational obsTimeInSystem     = new Observational( "Time in System" ),
9.                         obsTimeInQueue      = new Observational( "Time in Queue" );
10.   static Queue         queCustomer         = new Queue( "Customer Queue" );
11.   static TimeDependent timQueue            = new TimeDependent( queCustomer.length,"In Queue");

12. public void process( ){

13.   // create next customer arrival and record arrival time
14.   create( expInterArrivalTime.sample( ) );
15.   attArrivalTime = time;

16.   // assign service time for this customer and wait for service
17.   attServiceDelay = expServiceDelay.sample( );
18.   queue( queCustomer );

19.   // queue delay controlled by teller
20.   halt( );                    // suspend process until teller activates
21.   obsTimeInQueue.record( time - attArrivalTime ); // record queue time

22.   // service delay controlled by teller
23.   halt( );                    // suspend process until teller activates
24.   obsTimeInSystem.record( time - attArrivalTime ); // record system time

25.   dispose( );

26.   }// end of process method
27. }// end of Customer class
```

Figure 1: Customer Class Definition

might contain an array of all of the required processing delay distributions that this entity might require.

A significant advantage of SML and Silk over previous object-oriented languages is the use of process-oriented methods familiar to users of other simulation language. Every class must contain a *process* method containing these statements (or references to other classes that contain these statements) and it is here that the power of object-oriented modeling becomes evident. The *process* method [12-26] describes line for line the sequence of actions and information processing that defines the intelligent behavior of this system component. When the component is waiting for a decision or action of another intelligent component, the entity will halt its process until activated.

In this example, the Customer creates [14] the arrival of the next Customer using a sample from a Silk Exponential random variable object created in the data declaration. The *attArrivalTime* variable is then set to the current value of simulation *time* [15]. The "att" prefix is not required and has no special significance other than to remind the modeler that this is an instance variable unique to this object. Next, the *attServiceDelay* variable is then assigned a sample value from the appropriate service time distribution [17]. More complex models would likely have different distributions for different Customer classes and the use of an attribute for service delay will allow the Teller object access to the required processing time for each Customer instance and type.

This assignment of the service time to an attribute of the Customer object is an important object-oriented design choice. Is the time required for service an attribute of the Customer or should it be defined as a characteristic of the Teller? If different Tellers have different performance characteristics in performing the required service, those factors properly belong in the Teller class definition. But the basis for the service requirement is a characteristic of the customer and new customer types (which might inherit from this Customer class) should have the ability to modify the default customer service requirement without modification in Teller classes. Small design choices such as this are crucial to the adherence of a consistent design that will make models easier to reuse.

The Entity queue method then places this Customer instance in a *queue* [18] object which is simply an ordered list of Customer entities. Note that this queue is not linked with any particular *Resource* object so an Entity can be simultaneously listed in any of a number of *Queues.* This is extremely useful for modeling complex server behavior and facilitates proper statistics collection.

Until this point, the Customer entity is an intelligent component that has "pushed" through process methods to join the Teller queue. In the actual system, control of the choice of which Customer is served next is now passed to the Teller object. Consequently, the Customer object is halted by a *halt* method [20]. This distinction may seem cumbersome at first and the traditional entity-push

```
1.  import threadtec.silk.*;            // Silk general purpose classes

2.  public class Teller extends Entity {

3.    static Resource resTeller = new Resource ("Teller");
4.    static TimeDependent timTeller = new TimeDependent( resTeller.numBusy, "Utilization");

5.    public void process ( ) {

6.      while ( true ) {  // Teller not scheduled, continuously seeks new Customers

7.      // wait while condition is true (no customers in queue
8.      while( condition (  Customer.queCustomer.getLength( ) == 0 ) );

9.      // obtain reference to first customer in queue and remove it
10.     Customer entCustomer = (Customer)Customer.queCustomer.remove(1);

11.     // process customer and release teller
12.     seize ( resTeller );
13.     entCustomer.activate( );        // end halt for customer in queue
14.     delay ( entCustomer.attServiceDelay );
15.     entCustomer.activate( );        // end halt for customer in system
16.     release ( resTeller );

17.     }// end of while block for Teller processing

18.   }// end of process method

19. }// end of Teller class
```

Figure 2: Teller Class Definition

approach could be used throughout the process definition. But the object-oriented design requires that data characteristics and behavior of each object to be encapsulated within that object. The significance of this approach will become clearer as the behavior of the Teller object is described below.

The Customer is "pulled" from the queue and activated by the Teller object [shown in Figure 2, line 13]. The Customer object then continues the process method by recording the time spent in the halted state in a Silk Observation statistic object [21]. The TimeDependent object for Customer queue length [11] is automatically updated each time that the queue characteristic *length* is changed. Similarly, the end of service is also under the control of the Teller object so the Customer is again halted [23] until service is completed and the Customer is activated by the Teller object [Figure 2, 15]. Statistics for system time are then recorded for system time [24], and this instance of the Customer class is then disposed [25]. The dispose method actually places the entity object in a pool of Customer objects to be reincarnated as representations of future customers.

2.2 The Teller Class

The description of the Teller class is found in Figure 2. It defines the simulation system data and behavior from the perspective of the Teller. Since the Teller class is also a system component with independent intelligence, it is a modeled as an Entity [2]. A *Resource* object created to represent the Teller state [3]. The responsibility for when and how to change this state from busy to available is left to the *process* method for the Teller [5-18]. The Java *while* block [6] is used to continuously loop the single instance of the Teller throughout the simulation. By default, an entity executes the *process* method only once so this Java construct is necessary to allow the instance of Teller entity to continuously repeat the process method for subsequent Customers.

Interaction between Silk objects is reserved for the Silk/Java construct known as the *while(condition( ))* . The *while(condition( ))* combines the Java *while* statement and the Silk *condition* method [8]. Similar to the *halt* method, this statement temporarily stops the process of a Silk entity until activated by another process. In this case, the entity proceeds only when the expression defined within the condition method evaluates to false. The user is responsible for stating the conditions for the wait based on the state of Queues, Resources and other Silk or user-defined *state variables*. The corresponding construct in SML uses a similar but distinct structure that halts the entity until the combination of conditions is true.

At first look, this structure may appear cumbersome for simple systems. But more experienced modelers will appreciate the ability to create compound conditions for modeling resource behavior based on a variety of factors. Performance is less affected by this complexity as Boolean conditions are re-evaluated only when those objects which appear in the methods change value. Note that while many entities may be waiting for the same condition, only one is activated at a time to allow the activated entity an opportunity to change the condition (by seizing a resource or joining a queue).

```
1.  import threadtec.silk.*;        // references Silk methods found in this package

2.  public class Simulation extends Silk {

3.    public void init ( ) {

4.      // instantiate Silk Entity objects prior to the beginning of run
5.      Customer entCustomer = (Customer)newEntity( Customer.class ); // create first Customer
6.      entCustomer.start( 0.0 );

7.      Teller entTeller = (Teller)newEntity( Teller.class );         // create first Teller
8.      entTeller.start( 0.0);

9.    } // end init method

10.   public void run ( ) {

11.     // initialize Silk settings and flags prior to beginning of run
12.     setReplications( 1 );                    // End simulation at the end of 1 replication.
13.     setRunLength( 10000. );                  // Execute the simulation for 10000 time units
14.     setControlConsole (true);                // Use Control Console for interactive control

15.   } // end run method

16. } // end Simulation class
```

Figure 3: Simulation Class Definition

The net result in the case of the Teller is that the arrival or existence of an entity in the Customer queue results in the continuation of the Teller process. The Teller calls the remove method of the Queue object to obtain a Customer reference and remove the Customer entity from the queue [10]. This statement shows the use of a declaration of an object type within an expression (Customer entCustomer) and also the casting of the object type returned by the remove method to a Customer object type. Users commonly "wrap" complex methods like these within other simpler user-defined methods of their own creation. But the power of open-source SML is the ability of the development community to create and extend the language without sacrificing the underlying power and flexibility of the basic Entity methods.

The Teller object uses the reference to the Customer entity *entCustomer,* to access the service delay attributes of the Customer [14] and to invoke the activate method to resume the process method for the halted Customer entity as described earlier [13,15]. The seize and release methods in [12,16] modify the busy state of the Teller Resource object to allow the TimeDependent object to automatically track Teller utilization [4].

Finally, a brief note is necessary to explain the entity-thread concept enabled by Java. Entities are in SML and Silk Java threads. Java's support for multi-threaded execution enables the various types of entity behaviors (halt, delay and while(condition) described above. It is also an essential aspect to the implementation of a natural process-oriented modeling capability in Java. An executive thread running in the background coordinates the management of simulated time and the resumption of suspended threads.

2.3 The SIMULATION Class

All Silk models require a Simulation class, as shown in Figure 3, primarily for the purpose of creating the first instance of each class in the *init* method [3]. The *newEntity* method [5] is responsible for the creation and use of the Silk entity object pool of the indicated class and returns a reference to a new or existing member of that pool. The *start* method [6] then begins the execution of the Entity *process* method after a delay of the appropriate time units. In addition, other global parameters may be declared in the Simulation *init* method since all Entities extend Simulation and thus have access to all public variables and methods defined in the Simulation class. Finally, the *run* method of the Simulation class is automatically called by Silk to start the execution of the desired number of runs and run length [12,13]. Execution will end with the creation of a Summary Report window or the user can ask that a *Control Console* be used for interactive execution, tracing and animation control [14]. The Simulation class also has a *finish* method that is called at the end of each replication of the simulation to allow programmed execution of complex experimental designs.

2.4 Object-Oriented Design Choices

As seen in this example Java-based simulation provides great flexibility regarding the choice of object-oriented design patterns. Consider the decision to declare the *Queue* object to be a characteristic of the Customer class [Fig. 1, 10]. Even in this simple example, a user has at least four choices as to the proper assignment of this Queue object. One option if for the Queue object to be declared public and instantiated in the Simulation class which makes the queue reference available in all entity processes. But object-oriented design principles encourage the encapsulation of data and methods in their respective classes so that only those classes which need access to these objects can access these objects. The choice is then between the Teller class, the Customer class or a third class which might contain the physical description of the facility in which the Teller is located.

This decision is very important for complex model design and simulation object reusability. Modelers are encouraged to create process methods that reflect the actual characteristics and behavior of the corresponding intelligent system component. In this system, the Customer is in control of the behavior regarding which queue to join (and in more complex models, how long to wait in the queue chosen or whether to switch lines, etc.). For that reason, the queue definitions are made in the Customer class so that other versions of the model can change Customer queueing behavior without modifying the Teller class.

3 DEVELOPMENT ENVIRONMENTS

The Silk and SML simulation extensions to the Java language are themselves implemented entirely in Java. The only requirements for building and executing simulation models are a Java language compiler and run-time Virtual Machine that are compatible with Sun's JDK 1.3 specification of the language. Most commercial simulation software vendors constrain users to a single proprietary and often cumbersome development environment. Users can choose from a variety of professional, third-party Java Integrated Development Environments (IDE's). Each of these IDE's provides a sophisticated graphical interface and a rich collection of tools for project management, source code creation and modification, compilation, debugging, and deployment as standalone applications, browser-based applets, or server-based servlets. Figure 4 contains a screen snapshot of the example problem from the previous section within the Visual Café development environment.

4 JAVABEANS COMPONENT MODELING

Simulation without programming is unrealistic in most industrial strength models and no simulation vendor has the omniscience necessary to create components that do not require modification to represent the subtle, but important differences between system alternatives. But starting a model with graphical Silk components to create a simulation might be preferable to developing applications from scratch. Component-based simulation applications bring economies of speed in development and testing by capitalizing on previous successes (Pidd, 1000). Java-based simulation libraries like SML and Silk allow users to leverage the use of JavaBeans as a set of classes and programming conventions that constitute a component development model for the Java language.

JavaBeans are designed to be manipulated graph-ically within visual development environments like Visual Café. The bias is these early years of Java tool design has been toward use of JavaBeans for GUI development where property changes are easier than code changes. But the emergence of Enterprise JavaBeans for cross-platform, cross-developer application development is driving changes in these tools beneficial for simulation-component development. Visual programming allows for the concentration and separation of skills among developers. Skilled programmers build and make available beans for other developers with more domain-specific knowledge (and typically less technical programming expertise) to assemble visually into custom applications. This model works as well for simulation development applications as it does for complex programming applications.

JavaBeans can be applied to any aspect of an simulation application. It is a relatively simple matter to write self-contained, simulation modeling components based on SML and Silk that automatically make known their functionality and interoperability when incorporated into a JavaBeans visual development environment. Within this environment, they can be added to user-defined component toolboxes or palettes. Users can then assemble components visually into a model by placing them in a workspace and editing their properties to create a desired behavior. None of these manipulations require code to be written by the application developer.

While JavaBeans provides a means for packaging functionality into reusable units; beans by themselves do

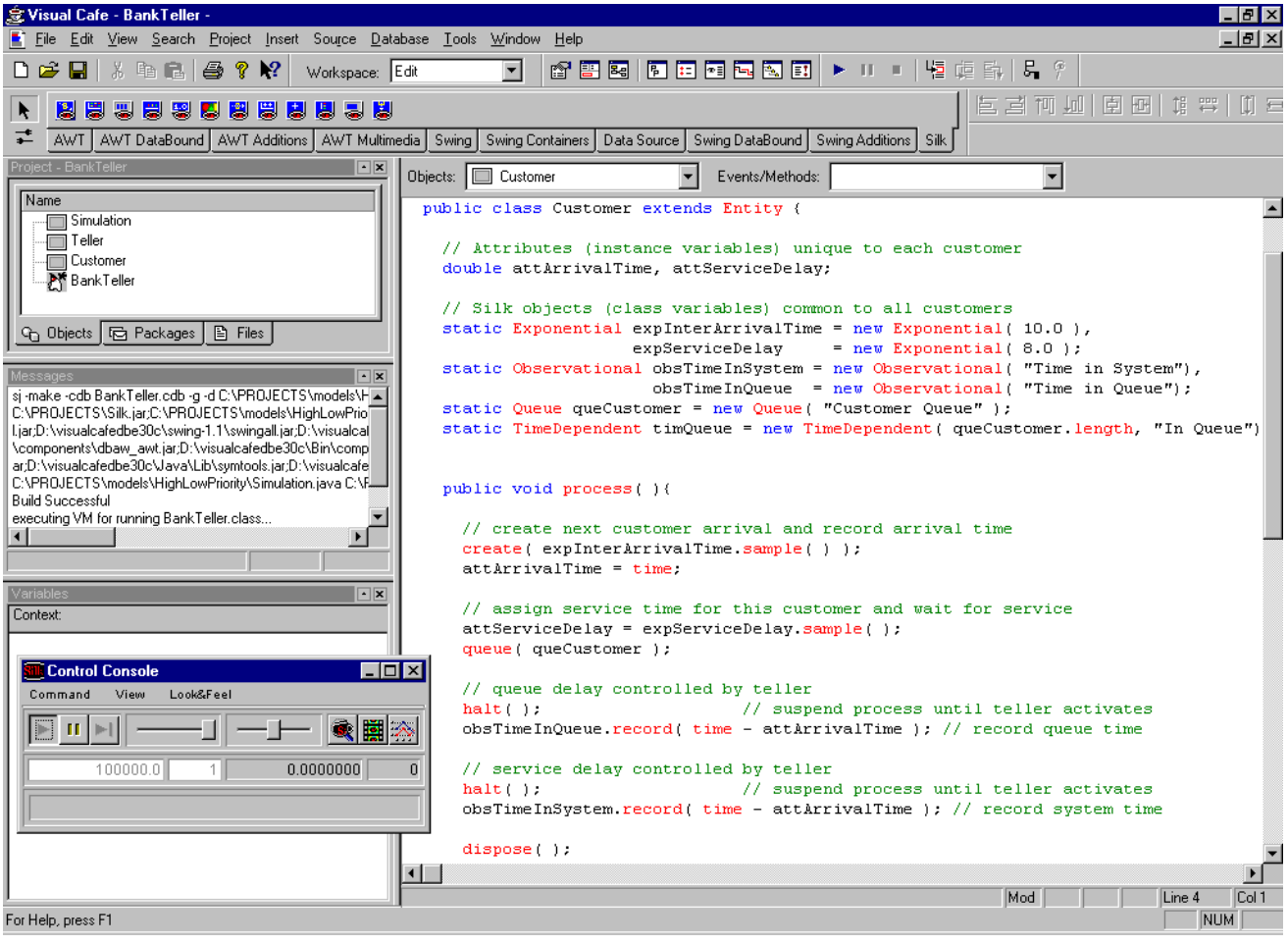

Figure 4: Modeling using the Visual Café Integrated Java Development Environment

not ensure reusability. To exploit the potential that simulation components have to offer, policies that define the functionality and modes of interoperability that allow components to be reused must be developed and adhered to. For example, there exist a set of policies and supporting classes that define the ways in which components must interact to produce animated displays of system state changes. These conventions were used in the implementation of a core capability in Silk that provides for animating entity movements, entity queueing, entity delays, and numeric and analog displays of state variable values among others. If the prescribed conventions are followed, it is a simple matter for users to modify these existing components or define new ones that will interoperate with any simulation model.

Developing guidelines for enterprise modeling components will be more challenging. Consideration will need to be given to the application domain as well as the range of model granularity the components are required to accommodate. Silk, SML and JavaBeans, however, significantly facilitate the manner in which these issues can be approached - both from a design and imple-mentation standpoint. In combination, they have the potential to raise component model development, interoperability, and reusability, to a new level.

5 SUMMARY

The Java language extensions that constitute SML and Silk were designed to encourage better discrete-event simulation through better programming by better programmers. Since the modeling language is integrated into the Java programming language, the full power and flexibility of the Java programming language is available. Unlike proprietary modeling environments, Silk users also benefit from the growing number of commercially available professional Java development tools. And unlike proprietary software, SML users can benefit from the large community of simulation researchers and practitioners who can guide and participate in SML development. The open-source licensing of SML will encourage developers to share language-level and component-level advances via the Internet and will also foster increased activity in the development of high-level, domain-specific simulation tools that end-users favor.

REFERENCES

Healy, K. and R. Kilgore. 1997. Silk™: A Java-based process simulation language. *Proceedings of the 1997 Winter Simulation Conference*, ed. S. Andradóttir, K. Healy, D. Withers, and B.L. Nelson, 475-482. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Kilgore, R. A., Healy, K. J. and Kleindorfer, G. B. 1998. The future of Java-based simulation. *Proceedings of the 1998 Winter Simulation Conference Proceedings*, ed. D. J. Medeiros, E. F. Watson, J. S. Carson, M. S. Manivannan, 1707-1712. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Kilgore, R. and K. Healy. 1998. Java, enterprise simulation and the Silk™ simulation language. *Proceedings of the 1998 International Conference on Web-Based Modeling & Simulation*, ed. P. Fishwick, D. Hill, and R. Smith, 442-449. SCS, San Diego CA..

Kilgore, R., K. Healy, and G. Kleindorfer . 1998. Silk™: usable and reusable Java-based object-oriented simulation. *Proceedings of the 12th European Simulation Multiconference*. SCS International, Ghent, Belgium.

Kilgore, R. 2001. Open-Source Simulation Modeling Language (SML). In *Proceedings of the 2001 Winter Simulation Conference,* ed., B. Peters, J. Smith. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Pidd, Michael , N. Oses and R. J. Brooks. 1999. Component-based simulation on the web? In *Proceedings of the 1999 Winter Simulation Conference,* ed., P. A. Farrington, H. B. Nembhard, D. T. Sturrock, and G. W. Evans, 1438-1444. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

SML, Simulation Modeling Language. Available online via <http://www.threadtec.com/sml> [accessed July 1, 2001].

AUTHOR BIOGRAPHY

RICHARD A. KILGORE is a co-author of the Silk language and a consultant in the development of industrial simulation and scheduling solutions. Dr. Kilgore is a founding member of the open-source Simulation Modeling Language (SML) Consortium. He has over 20 years of experience as a modeling consultant to Fortune 500 firms in a variety of industries with a variety of simulation and scheduling tools. He received his B.B.A. and M.B.A degrees from Ohio University and Ph.D. in Management Science from the Pennsylvania State University. Formerly, he was a capacity-planning analyst with Ford Motor Co. and Vice-President of Products for Systems Modeling Corp. His e-mail address is <kilgore@threadtec.com>.