Proceedings of the 1997 Winter Simulation Conference
ed. S. Andradóttir, K. J. Healy, D. H. Withers, and B. L. Nelson

Silk^{TM} : A JAVA-BASED PROCESS SIMULATION LANGUAGE

Kevin J. Healy
Richard A. Kilgore

Thread Technologies
P. O. Box 7
Chesterfield, MO 63017, U.S.A.

ABSTRACT

We discuss various aspects of the design, implementation, and use of *Silk*, a general purpose simulation language based on the Java programming language. *Silk* merges familiar process-oriented modeling structures with powerful object-oriented language features in an intelligent design that will encourage model simplicity and reusability. An important aspect of this design is the use of Java's built-in support for multithreaded execution which is employed as a means to coordinate the concurrent entity flows in process-oriented simulations. We also demonstrate how the use of JavaBeans supports graphical assembly of *Silk* modeling components in software environments such as Symantec's Visual Café, IBM's VisualAge, and Microsoft's J++. It is argued that a language such as *Silk* is essential to keeping simulation modeling on track with other revolutionary changes taking place in Internet-based computing.

1 INTRODUCTION

Simulation requires programming. If you disagree, you probably don't understand what your simulation software is doing. Behind that table or dialog box or icon is a section of programming code which is being modified and assembled into a new computer program. The business of simulation software has profited from promotion of the promise that the programming "burden" can be eliminated. It is a myth that has led to wasted client investment in models that poorly mirror system behavior and have no potential for distribution and reuse within the enterprise.

While software tools for discrete-event simulation have evolved rapidly over the last fifteen years, anyone who employs these tools in the solution of significant problems encounters a great deal of dissatisfaction. Much of the dissatisfaction stems from flawed commercial attempts to make simulation modeling methodology more *accessible* to *non-programmers*.

Increased accessibility to non-programmers has been accomplished through increased *separation* of the user from the base languages used to construct the simulation. Process simulation languages such as SIMAN (Pegden, Shannon and Sadowski 1995) separated the user from the libraries of FORTRAN subroutines which actually performed the simulation. Simulators such as PROMODEL (Benson 1996) encouraged greater separation by supplying users with an underlying model whose general structure is unchangeable.

Separation has not eliminated the need for programming in simulation model building. In fact, successful industrial modelers are those who overcome separation by "programming" around the limitations caused by separation. Separation is also an obstacle to the long-term model development and maintenance because this "programming" skill is outside of the mainstream of information systems training in academia and within the enterprise. The solution to separation involves a multi-layered approach suggested in the design of SLX (Henriksen 1996) but implemented in an integrated general purpose programming and discrete-event simulation language.

The current generation of discrete-event simulation languages are limited in ways that make their use in implementing sophisticated end-user models a difficult exercise. In particular, many either lack completely or implement only crudely many modern general purpose language design features. Missing are features such as true object-orientation, structured data representation, input/output mechanisms, graphical windowing interfaces, and support for database and spreadsheet interfaces necessary to support the development of advanced modeling tools.

A significant opportunity now exists to address these limitations through the use of Internet-related software technologies. Just as the personal computer revolution fostered prior advancements in manufacturing simulation (e.g. animation), the Internet revolution will foster new methods of employing simulation across the enterprise. One of the most promising of these enabling technologies is the Java programming language.

The advent of the Java programming language and other Internet-related technologies facilitate a fundamentally different approach to the design and implementation of simulation software that specifically addresses the shortcomings of existing tools. In particular, the combination of process-oriented simulation, Java, and the JavaBeans component architecture will allow models to be packaged in a way that increases accessibility to the user without the compromises required by increased separation of the user from the underlying modeling language.

The work described here represents an effort to change the current course of simulation modeling. It is not just another simulation language and it is not just another modeling environment. It is an opportunity to establish simulation programming in the mainstream of the computing world. It is an invitation for the next generation of programmer/modelers to make simulation a tangible asset that can be shared throughout the enterprise beyond the tenure of the model creator. It is a methodology that promotes convenience but not through the limitation of creativity or skill.

In sections 2 and 3 of this paper we review some fundamentals of process-oriented models, object-oriented programs and the Java programming language. Examples of the integration of process-orientation and object-orientation in the Java-based *Silk* simulation language are presented in Section 4. Section 5 demonstrates the use of JavaBeans component architecture for developing self-contained, reusable modeling components.

2 PROCESS-ORIENTED SIMULATION AND OBJECT-ORIENTED METHODS

Process-oriented simulation is a phrase commonly used to denote a particular world-view employed in modeling the dynamics of a discrete-event system. The prevailing frame of reference is through the transactions of transient system entities. A process-oriented model is a description of the sequence of processing steps these entities experience as they flow through the system. The approach has significant intuitive appeal and is the predominant modeling world-view supported by commercial simulation software tools.

The origins of this approach can be traced to the creators of SIMULA (Kirkerud 1989), an ALGOL-based programming language with process simulation extensions developed in the 1960's. It is also significant that the process-oriented features in SIMULA embodied in their design and implementation the essence of what are now commonly called object-oriented methods. Only after a lengthy period of relative anonymity were these concepts rediscovered and more generally formalized.

Despite common origins and the ideal suitability of object-oriented methods to the task of structuring process models, the simulation user community has been slow to adopt object-oriented methods. A contributing factor has been a lack of commercial simulation software tools with coherent and accessible support for object-oriented process modeling. Much of what is available tends to couch simple concepts in arcane terminology and implementations with unnecessarily cumbersome syntax and semantics. In fact, an appreciation for object-oriented methods and their attendant benefits requires only the understanding of a few simple concepts; namely, *encapsulation*, *classes*, *messages*, and *inheritance*.

Objects and their software implementation are patterned after real-world objects. They have data (attributes, characteristics, properties, etc.) that represent the *state* of the object and a set of *behaviors* that describe the ways in which they can be operated on. In an object-oriented approach, the association between the state of an object and it's set of behaviors is made explicit via *encapsulation* whereby both are defined in an integral, self-contained unit called a *class*. This collective definition serves as a template or blueprint for creating particular *instances* of the corresponding *class*. Each *instance* (of which there may be many) possesses its own unique copy of the state-related data defined by the class but shares the behaviors. Communication between objects is confined to a formalized system of *messages*. It is convenient to think of message generation and processing as just additional types of behaviors defined on the sending and receiving classes. Finally, *inheritance* is a mechanism by which new classes can be defined as extensions of existing ones. The derived class has all the characteristics and behaviors of the parent class (which may itself be derived from others) plus some added functionality in the form of new characteristics and/or behaviors.

It is easy to see in these concepts two particularly natural applications to process-oriented simulation modeling. One is the use of encapsulation to make explicit the association between the representation of an entity and its sequence of processing steps. Another is the encapsulation of extended sequences of low-level processing steps into submodels whose behavior can be invoked as a single high-level processing step by instances of a desired entity class. These two notions are central to the design of *Silk* which implements an object-oriented, process modeling capability within the framework of the Java programming language.

3 WHY JAVA?

The idea of adding process-oriented simulation capabilities to a general purpose object-oriented programming language is not new. CSIM (Schwetman 1995) and YANSL (Jones and Roberts 1996), both of which are based on C++, take this approach. There are, however, unique aspects to the Java language that lead to fundamental distinctions between our approach and others.

The Java language has several features that are ideally suited to the implementation of advanced discrete-event simulation architectures and reusable simulation software components. 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. Also, where other languages rely on a host of third-party supported libraries, Java includes native support for networking and common Internet protocols, database connection (via Java Database Connectivity), multithreading, distributed objects (via Remote Method Invocation), and graphical user interfaces (via the Abstract Windowing Toolkit). Another well recognized strength and appeal of Java is the "Write Once, Run Anywhere" platform independence it provides developers. Perhaps the most compelling argument for a Java-based approach is to leverage the rapidly growing number of resources being dedicated to the support of Java as a programming standard.

Java was released by Sun Microsystems in May of 1995. The language is actually the by-product of an initiative by a small group within the company attempting to develop a compact, portable programming language for consumer electronics (O'Connel 1995). Despite the unprecedented level of attention devoted to Java during the relatively short time since its introduction, there are some widely held misconceptions regarding the language. This is the result of an interesting combination of phenomena. Foremost among these is the fact that few people have firsthand experience with Java. Instead, most people's impressions are likely to have come from superficial accounts that appear in the popular press which tend to focus on the potential business implications of Java's widespread acceptance on Microsoft Corp.'s dominance of the market for desktop computing software (Schlender 1996). Accounts that address its use as a programming tool linked to Internet Web browsers tend to emphasize applications involving simple types of dynamic behaviors for Web pages or so-called "thin" client programs whose purpose is merely to provide users a remote interface to server-based applications. In fact, Java is full-fledged general purpose language for creating safe, portable, robust, object-oriented, multithreaded, interactive programs for virtually any area of application. (Naughton 1996). And while the Internet aspects of the language are a key feature, it is also the case that non-network-based Java applications can be easily configured to run in a stand-alone environment that does not require Internet access.

Another widely held misconception involves the interpretive nature of the Java runtime virtual machine and its effect on execution speed. In fact, with so-called just-in-time compilation techniques the translation to machine level instructions from byte code that occurs when a Java method is first invoked is cached so that subsequent calls execute as if pre-compiled. This results in execution speeds that are comparable to traditional compiled languages.

4 *Silk* : A JAVA SIMULATION LANGUAGE

The design of *Silk* was motivated by the desire for a small but powerful general purpose process–oriented modeling capability that could be extended by users in a straightforward and unrestricted manner. The idea was to start with a superior general purpose programming language and add to it process-oriented simulation modeling capabilities.

In the *Silk* simulation language, models are developed directly in the Java programming language using a package of classes consisting of a relatively few but powerful process-oriented modeling features. Both the design and implementation of the modeling constructs exploit the inherent object-oriented nature of the Java language to produce a harmonious blend of process-oriented semantics and syntax. The powerful and well-designed general purpose object-oriented framework of the Java language also provides modelers with a structured mechanism to easily extend the core modeling components without limitation.

Many of the features of the *Silk* modeling formalism can be illustrated by way of a few simple examples. Although helpful, detailed knowledge of the Java programming language is not required. Figure 1 contains a Java class named `Customer` that models the representation and behavior of entities processed by a single-server queue.

The `Customer` class definition begins with declarations of the data representing the state of these entity types. Each instance of the `Customer` class is assigned an attribute that stores its time of arrival to the system. The interarrival and processing time

```
import silk.*;

public class Customer extends Entity {

   double time_of_arrival;

   static Uniform inter_arv_time     = new Uniform( 2, 5 );
   static Uniform process_time       = new Uniform( 1, 4 );
   static Queue in_q                 = new Queue(“Input Queue”);
   static Resource server            = new Resource(“Server”);

   static Time_Weighted              = new Time_Weighted( in_q.length, “Queue Size”);
   static Observational system_time  = new Observational(“Time in system”);

   public void run ( ) {
      create  ( inter_arv_time.sample( ) );
      time_of_arrival = time;
      queue   ( in_q );
      while   ( condition( server.getAvailability( ) == 0 );
      dequeue ( in_q );
      seize   ( server );
      delay   ( process_time.sample( ) );
      release ( server );
      system_time.record( time – time_of_arrival );
      dispose ( );
   }
}
```

Figure 1: Example of *Silk* Process-Oriented Syntax (Single-Server Model).

generators, the server, and its associated queue also constitute part of the state of the `Customer` class. The `static` qualifier that precedes each of these declarations means that these data are shared by all instances of the `Customer` class rather than being unique to each entity that is created. It is significant that to note that each of these process simulation specific data types (`Uniform`, `Queue`, and `Resource`) are themselves implemented as classes in the *Silk* package. Access to them from the `Customer` class is enabled by the inclusion of the `import` statement. The remaining components of the state consists of `static` instances of other *Silk* classes used to record time-weighted statistics on the queue length and observational statistics on the time customers spend in the system.

In Java, class behaviors are implemented as *methods* which are similar to executable functions in traditional procedural programming languages. The *Silk* formalism requires that each entity class definition contain, at the least, the distinguished method named `run` which serves as the starting point of execution for each instance that is created. In this case, the `run` method consists of a sequence of other method invocations that model the familiar “inter-create, wait-for-server, seize-server, delay, release-server” logic that characterizes the activity in a single server queue. The corresponding methods that implement these individual process modeling behaviors are all inherited from the predefined *Silk* class named `Entity` by virtue of the `extends` qualifier included in the `Customer` class definition.

It is also interesting to note the explicit uncoupling of the queueing, status delay, and seize processes that is employed in the design of *Silk*. Delays based on the state of the system are modeled more generally by the powerful and flexible `while(condition( ))` construct which delays an entity until the prescribed condition is satisfied. In fact, the `queue` and `dequeue` processes are in this case employed only as a means to collect statistics on the number of entities awaiting service.

In addition to the entity classes like `Customer`, two other classes are needed to produce a working simulation. The first is a class named `Simulation` that implements the behavior associated with the initialization of a model. At a minimum, this includes setting the simulation run length and creating the first instance of an entity class. The `Simulation` class also serves to define any state-related data that is global to the simulation model. The other required class is part of every Java program and provides the point-of-entry at which the program begins execution. By convention, this is the `init` method for programs implemented as browser-based applets or the `main` method for those implemented as stand-alone applications. *Silk* simulations can be implemented as either. The only requirement is that this distinguished class create an instance of the predefined class named `Silk` and invoke its `start` method. The two class definitions appearing in Figure 2 along with the `Customer` class defined in Figure 1 constitute a working simulation model of the single server queue.

```
import silk.*;

public Class Example {

  public static void main(String args[]){
    Silk my_silk = new Silk( );
    my_silk.start();
  }
}

public Class Simulation extends Silk {

  public void run ( ) {
    end_time = 100.;
    Customer first_one = new Customer( );
    first_one.start( );
  }
}
```

Figure 2: Required *Silk* Simulation Classes

Native support for multithreaded execution is a crucial aspect to the implementation of a natural process-oriented modeling capability in Java. Instances of classes that extend `Entity` run as separate threads of execution that are alternately suspended and resumed to coordinate the time-ordered sequencing of entity movements in the model. A thread is suspended when the corresponding entity encounters a deterministic or status delay modeled by the `delay` method and `while( condition( ))` constructs, respectively. There is also an executive thread running that coordinates the management of simulated time and the resumption of suspended threads whenever a corresponding entity's deterministic time delay expires or a system state change occurs that triggers the emergence of an entity from a status delay.

It is always intimidating to see a new modeling language or programming language for the first time, so before moving to the next example, consider the potential of what has been presented in this section. The objective of Silk is not a "better" approach to modeling the single-server queue. Every simulation language is successful in making simple models appear simple. What is unique in the design of *Silk* is the flexibility provided to the user to make complex models more manageable through a combination of programming and modeling extensions to the fundamental *Silk* classes. Users are encouraged to make full use of the Java programming language to write customized extensions to *Silk* that include more complex process classes such as "transport", "warehouse" or "schedule ". And users are encouraged to demonstrate and share these extensions with their internal and external clients who can browse and execute *Silk* models over the network using any Java-compatible browser on any hardware platform.

As an example of the potential of this extensibility and reusability, consider the possibility of creating a more generic "single-server" process. The goal is to encapsulate the generic behavior associated with a single-server queue. One approach is to implement the single-server sub-model as the inheritable method of the `Entity` parent class. However, in models where many different types of entities exist, only a subset of these entity types may utilize the new "single-server" process.

An alternative is to encapsulate the single-server sub-model in its own class that inherits from the predefined *Silk* class named `Process`. This approach is shown in Figure 3. The queue, server, and delay time distribution objects have been made properties of a new class named `SingleServer` whose `serve` method implements the generic process logic of a single server queue. The particular delay time distribution is specified when creating an instance of the `SingleServer` class which takes place when the Simulation class is automatically instantiated at initialization. Finally, the process logic associated with instances of the `Customer` class has been modified so that the sequence of steps involved in serving customers of this type is triggered by simply invoking the `serve` method on the desired single server.

The value of these types of extensions to the *Silk* language may be best appreciated by more experienced modelers who can better visualize the possible consolidation of the base *Silk* classes into a customized higher-level language. But the ultimate value may be to those new users who are attracted to visual modeling techniques where models are created through *point-and-click/drag-and-drop* interfaces with libraries of modeling components. This capability is simply another extension of the Java language known as JavaBeans.

```
import silk.*;

public class Simulation extends Silk {

   public SingleServer station  = new SingleServer( new Uniform( 2,4 ) );

   public void run ( ) {
      end_time = 100.;
      Customer first_one = new Customer( );
      first_one.start( );
   }
}


public class Customer extends Entity {

   double time_of_arrival;

   static Uniform inter_arv_time = new Uniform( 2, 5 );

   public void run ( ) {
      create ( inter_arv_time.sample( ) );
      time_of_arrival = time;
      station.serve( this );
      system_time.record( time – time_of_arrival );
      dispose ( );
   }
}


public class SingleServer extends Process {

   Resource server  = new Resource ( "Server" );
   Queue  in_q      = new Queue ( "Input Queue" );
   Distribution process_time;

   public SingleServer ( Distribution dist ) {
      process_time = dist;
   }

   public void serve ( Entity ent ) {
      ent.queue ( in_q );
      while ( ent.condition ( server.getAvailability( ) == 0 ) );
      ent.dequeue ( in_q );
      ent.seize ( server );
      ent.delay ( process_time.sample( ) );
      ent.release ( server );
   }
}
```

Figure 3: Example of Extending *Silk* to Include a Single-Server Process

5 JAVABEANS AND VISUAL MODELING

There is a clear trend in software development toward object-oriented techniques and component software environments that allow programmers to build applications in less time and with less money. The conventions that constitute JavaBeans bring the component development model to Java and *Silk*.

The component model is made up of an architecture and application programming interface. Together these elements provide a structure whereby program components can be combined to create an application. One of the important features of JavaBeans is that it does not alter the existing Java language. Instead, the strengths of Java have been built-upon and extended.

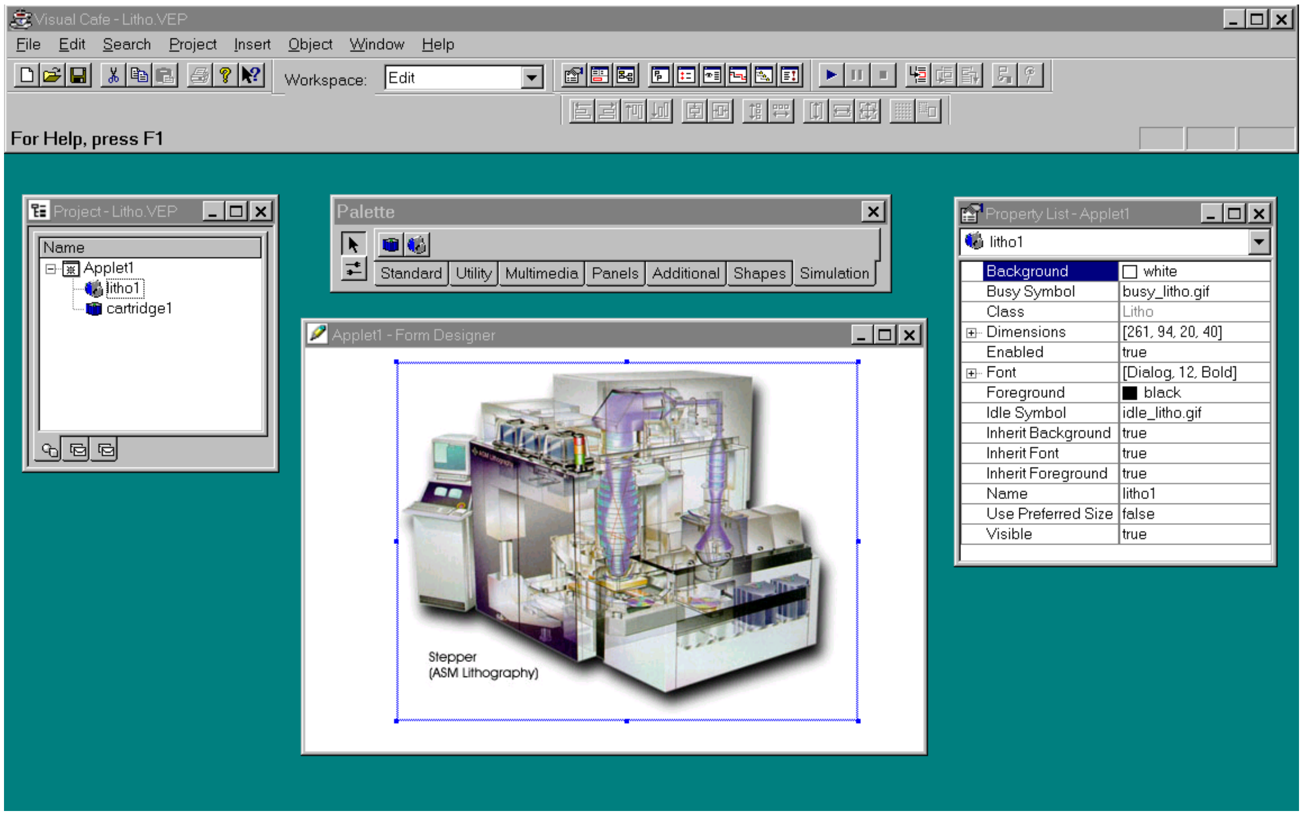
Since JavaBeans are built purely in Java, they are fully portable to any platform that supports the Java runtime environment. Visual programming is a key part of the JavaBeans component model. User-defined “beans” can be incorporated into any of the growing number of sophisticated visual development tools that support the open JavaBeans standard including Symantec's Visual Café, IBM's VisualAge, and Microsoft's J++.

It is a relatively simple matter to write self-contained, reusable JavaBean components that automatically make known their functionality and interoperability to these development environments. Within the environment, they can be incorporated into user-defined component toolboxes or palettes. Users can then assemble components visually into an application 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.

The combination of JavaBeans and the *Silk* package of process simulation extensions to Java is demonstrated in Figure 4 using Symantec’s Visual Café. The single-server process is associated with an icon and incorporated into Symantec’s component tool palette under a tab labeled Simulation. In this case, the server is a Lithography workstation which is part of a semiconductor fabrication processs. All of the properties of the Lithography workstation object are automatically exposed to the user and can be modified through a “properties” editor. The standard Java `paint` method was also added to the single server component to control its graphical depiction during design time and animation during the execution of the model.

6 SUMMARY

The *Silk* simulation language has been proposed as a breakthrough opportunity to encourage better discrete-event simulation through better programming by better programmers. Since the modeling language is integrated into the programming language, the full power and flexibility of the Java programming language is available and any Java environment (Symantec Visual Café, Microsoft J++, etc.) can be used for model building and debugging. And since a simulated entity is implemented as a separate program thread, there is a direct correspondence between an entity executing process steps in the process model and



Workspace: Edit

Name
- Applet1
  - litho1
  - cartridge1

litho1

Background	☐ white
Busy Symbol	busy_litho.gif
Class	Litho
Dimensions	[261, 94, 20, 40]
Enabled	true
Font	[Dialog, 12, Bold]
Foreground	■ black
Idle Symbol	idle_litho.gif
Inherit Background	true
Inherit Font	true
Inherit Foreground	true
Name	litho1
Use Preferred Size	false
Visible	true

Figure 4: Graphical Modeling Using *Silk*-Based JavaBeans.

the entity object processing program statements in the modeling program. These features encourage a host of future extension to visual modeling, distributed modeling and easily distributed libraries of modeling components.

Most importantly, the *Silk* language is an opportunity to make simulation more accessible without sacrificing power and flexibility. The *Silk* language offers the potential for industry-specific and company-specific modeling components which can be distributed and even executed through the Internet. Support for these models can utilize existing information system resources and computer science training. Finally, the simulation software industry can leverage commercially available programming environments and focus on building models instead of modeling environments that quickly become obsolete.

REFERENCES

Benson, D. 1996. Simulation Modeling and Optimization using Promodel. *Proceedings of the 1996 Winter Simulation Conference*.

Henriksen, J. O. 1996. An Introduction to SLX. *Proceedings of the 1996 Winter Simulation Conference*.

Jones, J. and S. Roberts. 1996 Design of Object-Oriented Simulations in C++, *Proceedings of the 1996 Winter Simulation Conference*.

Kirkerud, B. 1989. *Object-Oriented Programming with SIMULA*. Addison-Wesley.

Naughton, P. 1996. *The Java Handbook*. Osborne McGraw-Hill, Berkeley, CA.

O'Connel, M. 1995. Java: The Inside Story. In *SunWorld Online*. Sun Microsystems, Inc.

Pegden, C. D., R. E. Shannon, and R. P. Sadowski. 1995. *Introduction to SIMAN*. McGraw-Hill, New York, NY.

Schlender, B. November 1996. Sun's Java: The Threat to Microsoft is Real. *Fortune Magazine*.

Schwetman, H. 1995. Object Oriented Simulation Modeling with C++/CSIM17, *Proceedings of the 1995 Winter Simulation Conference*.

AUTHOR BIOGRAPHIES

KEVIN J. HEALY is the author of the Java-based *Silk* simulation language and a partner in Thread Technologies, a company involved in the development of Internet-based simulation capabilities. He received his Ph.D. in Operations Research from Cornell University. He served as the General Applications Track Coordinator for the 1994 Winter Simulation Conference and is an Associate Editor for the Proceedings of the 1997 Winter Simulation Conference.

RICHARD A. KILGORE is a partner in Thread Technologies. He has over 15 years of experience as a modeling consultant to Fortune 500 firms in a variety of industries. 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.