Proceedings of the 1998 Winter Simulation Conference
D.J. Medeiros, E.F. Watson, J.S. Carson and M.S. Manivannan, eds.

INTRODUCTION TO SILK^{TM} AND JAVA-BASED SIMULATION

Kevin J. Healy
Richard A. Kilgore

ThreadTec, Inc
P. O. Box 7
Chesterfield, MO 63017, U.S.A.

ABSTRACT

Silk is a general-purpose simulation language based on the Java programming language. Silk, however, is more than just another simulation language. It is a new standard for creating usable and reusable object-oriented simulation components and models. The Silk modeling language is ***usable*** because it represents a unique marriage of familiar process-oriented modeling constructs and the powerful object-oriented features of a general purpose programming language. Silk models are ***reusable*** because they are created using professional Java visual development environments that support graphical assembly of self-contained Silk modeling components based on the JavaBeans component architecture. Java's adoption as the standard language for computation on the World Wide Web also means that Silk models can be deployed in ways that exploit this new medium.

1 INTRODUCTION

The Silk language was designed to be different things to different people. To some users, Silk is a rich set of Java class libraries that can be creatively assembled into a variety of modeling constructs. To others, Silk is a process-oriented simulation language that provides the power and flexibility to program within a standard programming language and industry standard development environments. To others, Silk is a visual modeling environment where Silk-based modeling components can be graphically assembled to quickly create simulation applications. Perhaps most importantly, Silk is the first practical implementation of a tool for building self-contained, reusable modeling components and domain-specific simulators.

Silk is not so much a language in itself as it is an extension of the Java language. The power, flexibility and scalability of Silk derive directly from Java and the seamless way in which the simulation-related extensions that constitute Silk 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 Java language. 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 Java's built-in support for multi-threaded execution, which is essential to representing in a natural way the inherently concurrent flow of entities in process simulation models. Still another is JavaBeans, a simple convention for defining software components that automatically make known their functionality and interoperability when incorporated into any of the sophisticated visual development tools that support the JavaBeans standard. By incorporating the Silk modeling environment into Java, users also have direct access to Java's native support for standard Internet communication protocols, database connectivity and graphical user interface development. Java's platform neutral design also means that Silk models can be developed and executed on practically any combination of computer hardware and software platforms.

This paper serves primarily as an introduction to the language level features of Silk which are the basis for developing reusable simulation components and higher level domain-specific simulators. A series of articles dealing with other important aspects of the Silk language are listed in the References section. Section 2 contains a brief overview of object-oriented modeling. Section 3 demonstrates the implementation of these concepts in Silk by way of an example that also serves to illustrate other important features of the language and their use. Section 4 provides the software requirements for building and executing Silk models. Section 5 contains a general overview of the entire Silk feature set. A brief overview of Silk-based JavaBeans modeling and animation components is given in Section 6. Section 7 contains concluding remarks.

2 OBJECT-ORIENTED MODELING

Silk is a wholly object-oriented modeling formalism. An appreciation for object-oriented methods requires 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 (perhaps derived from others) plus added functionality in the form of new characteristics and/or behaviors.

These concepts can be applied in beneficial way to traditional process-oriented simulation, where the prevailing frame of reference is through the transactions of transient system entities. In this case, encapsulation is used to make explicit the association between the representation of an entity and its sequence of processing steps as it moves through the system. Most commercial simulation software is restricted to this push-type assembly-line view of the world where the focus is on transient entities whose aim is to complete a series of process steps, each of which may require a set of passive, unintelligent, capacity-limited resources. For simplistic models, this approach is often sufficient. However, in real-world engineered systems there is almost always a need to model active, intelligent resources who from a modeling standpoint are more naturally seen to control decision making within the system rather than the transient entities being acted upon. For this situation, a pull-oriented description of activity from the resource's point-of-view yields a more natural and efficient model of the system dynamics. In some cases, this resource-pull world-view can be modeled within an entity-push framework through the creation of executive entities whose purpose is to manage the interplay between the transient entities and static resources. However, this approach is never ideal and almost always less than satisfactory.

The Object-oriented design of Silk is ideally suited to combining entity-push and resource-pull process logic within the same model. Classes can be created to represent entities that lend themselves to push-type descriptions and other classes can be created that define the state and behaviors of each of the types of intelligent resources in the systems. There also exists within Silk a variety of mechanisms to coordinate the communication that takes place between these types of submodels.

The ability to make explicit the decomposition and encapsulation of entity-push and resource-pull logic is important to managing the complexity of simulation models. It is also a fundamental requirement for creating reusable simulation modeling components

3 AN EXAMPLE

In this section, we present a simple example containing a batch service operation that is characteristic of certain stages of semiconductor fabrication processes. The example demonstrates how easily the entity-push and resource-pull viewpoints can be combined in the same model using Silk's object-oriented framework. The example also serves to demonstrate other important features of the language and their use.

The example model consists of a tandem arrangement of two servers fed by a single stream of arriving entities. The two servers must process each arriving entity in order. Input queues in front of both servers accommodate those entities that must await the availability of the server before initiating service. The first server has the capability to process only a single entity at a time. The second, however, processes customers in batches that vary in size from 10-20. The dynamics of the batch sizing decision are as follows. The server always waits until there are at least 10 entities in the preceding queue before beginning a batch service. If, after finishing a batch, there are more than 10 in the preceding queue, then the server will take as many entities as it can up to a limit of 20. The introduction of min and max batch size thresholds results in a surprisingly complex tradeoff between the wait times of entities and the utilization of the server capacity that is well-suited to analysis by simulation.

The dynamics of the first stage of service are easily described from the viewpoint of arriving entities in the form the queue-seize-delay-release sequence that is characteristic of traditional process simulation languages. This same approach, however, is ill suited to describing the dynamics of the second stage of where the batch sizing decision is more naturally coordinated by the server rather than the arriving entities.

Figure 1 contains a Java class definition named `MyEntity` that models the representation of the transient entities in the system and the behavior associated with their first stage of service. The class definition begins with declarations of the data that represent the state of the entity types. Each instance of the `MyEntity` class is assigned an attribute that stores its time of arrival to the system. The interarrival and processing time generators, the passive resource that models the server, its associated queue, and the statistical accumulators also constitute part of the state of the `MyEntity` class. The `static` qualifier that precedes each of these declarations means that these data are shared by all instances of the `MyEntity` class rather than being unique to each entity that is created. The remaining components of the state consist 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. It is significant that to note that each of the simulation specific data types (`Exponential`, `Queue`, `Resource`, `Observational`, and `TimeDependent`) are themselves implemented as objects in the Silk formalism. Access to them from the `MyEntity` class is enabled by the inclusion of the `import` statement.

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

In Silk, there is no implicit status delay logic associated with the `queue( )` method, nor is it linked directly with the `seize( )` and `release( )` methods which merely toggle the state of a specified passive resource between idle to busy. Delays based on the state of the system are modeled more generally by the powerful and flexible `while(condition( ))` construct which delays an entity as long as the prescribed condition is true. As a result, an entity can reside simultaneously in any number of queues. This is a particularly useful feature for modeling certain types of dynamic behavior or for decomposing a wait time or queue length into constituent parts. An entity departs a queue only after dequeueing itself or after being removed by another entity.

Support for multithreaded execution is an essential 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 an unconditional wait at a `delay( )` method or a status delay, the most general of which is modeled by the `while(condition( ))` construct. An executive thread running in the background 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.

After completing the first stage of service, instances of `MyEntity` join the input queue to the second stage of service and then activate the `halt( )` method which unconditionally delays the entity until it is explicitly reactivated. In this case, the entity that coordinates the reactivation is an instance of the class `BatchServer` defined in Figure 2.

The `BatchServer` class models the second stage of service from the perspective of the server as opposed to the arriving entities. When its `process( )`method is invoked, an instance of `BatchServer` initiates an endless loop of idle-busy cycles. The cycle starts with the server delaying until the minimum batch size of 10 entities are present in the queue preceding the second stage of service. If there are at least 10 in queue, the server takes as many as it can up to the maximum capacity of 20. The designated number of entities are removed from the input queue and inserted into a temporary queue where they reside while the server delays by the batch processing time. After delaying, the server removes the individual entities from the temporary batch queue and invokes the `activate( )` method on each one. This causes each of the activated entities to resume executing their own `process( )` method at the point following the call to `halt( )` where they were previously suspended. After resuming executing, the entities merely record their time in system, and dispose of themselves. At the same time, the `BatchServer` sets its status to idle, and loops around to recheck the condition for initiating another batch service.

```
import com.threadtec.silk.*;

public class MyEntity extends Entity {

  double attArvTime;
  static Exponential expInterArvTime = new Exponential ( 1.5 ),
                     expServTime    = new Exponential ( 1 );
  static Queue queStage1       = new Queue( “Stage 1 Input Queue” );
  static Resource resServer    = new Resource ( “Stage 1 Server” );
  static TimeDependent tdQueue = new TimeDependent(queStage1.length, “Stage 1 Input Queue” );
  static Observational obsSysTime = new Observational( “Time in System” );

  public void process ( ) {
     create( expInterArvTime.sample( ) );
     attArvTime = time;
     queue( queStage1 );
     while ( condition( resServer.getAvailability( ) == 0 ) );
     dequeue( queStage1 );
     seize( resServer );
     delay( expServTime.sample( ) );
     release( resServer );
     queue ( BatchServer.queStage2 );
     halt( );
     obsSysTime.record( time – attArvTime );
     dispose( );
  }
}
```

Figure 1: `MyEntity` Class Definition.

```
import com.threadtec.silk.*;

public class BatchServer extends Entity {

  int batchSize, IDLE=0, BUSY=1;
  IntStateVar isvStatus         = new IntStateVar( IDLE );
  public static Queue queStage2 = new Queue( “Stage 2 Input Queue” ),
                     queTemp   = new Queue( “Temporary Queue” );
  Exponential expServTime    = new Exponential( 20. );
  TimeDependent tdQueue      = new TimeDependent( queStage2.length, “Stage 2 Input Queue” );
  Observational obsBatchSize = new Observational( “Batch Size” );
  Entity entTemp;

  public void process ( ) {
     while ( true ) {
        while( condition ( queStage2.getLength( ) < 10 ) );
        batchSize = Math.min( 20, queStage2.getLength( ) );
        obsBatchSize.record( batchSize );
        for ( int i=1;  i<=batchSize;  ++i ) {
           entTemp = (Entity)queStage2.remove(1);
           queTemp.insert( entTemp );
        }
        isvStatus.setValue( BUSY );
        delay( expServTime.sample( ) );
        while( queTemp.getLength( ) > 0 ) {
           entTemp = (Entity)queTemp.remove(1);
           entTemp.activate( );
        }
        isvStatus.setValue( IDLE );
     }
  }
}
```

Figure 2: `BatchServer` Class Definition.

In addition to `MyEntity` and `BatchServer`, two other classes are needed to produce a working Silk simulation. The first is a class named `Simulation` whose `run( )` method serves as the starting point of execution for the model. Users can also define `init( )` and `finish( )` methods in the `Simulation` class which are called at the beginning and end of each run of the simulation, respectively. The `Simulation` class also serves as a place for users to define model specific data that is global to all instances of classes that are derived from Silk's predefined `Entity` class. The other required class is part of every Java program and defines the point-of-entry at which the program begins executing. 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 `begin( )` method. The two class definitions appearing in Figure 3 along with the `MyEntity` and `BatchServer` classes defined in Figures 1 and 2 constitute a complete model of the tandem server example.

```
import com.threadtec.silk.*;
public Class Example {

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

import com.threadtec.silk.*;
public Class Simulation extends Silk {

  public void run ( ) {
     setReplications( 1 );
     setRunLength(100);
     setControlConsole( true );
  }

  public void init( ) {
    MyEntity first = new MyEntity( );
    first.start( 0.0 );
    BatchServer only= new BatchServer( );
    only.start( 0.0 );
  }
}
```

Figure 3: Required Silk Simulation Classes.

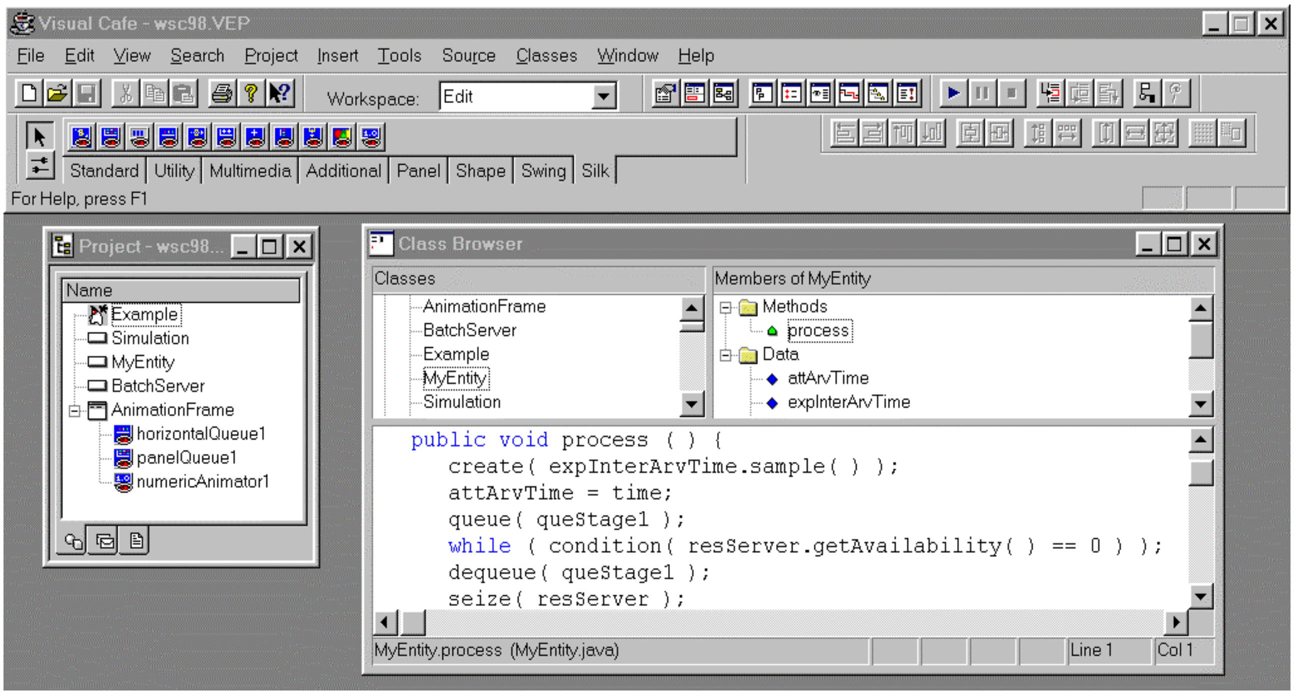
The model begins execution at the `run( )` method of the `Simulation` class which in this case initializes the simulation run length and number of replications and also activates Silk's `ConstrolConsole` – an interactive tool for parameterizing and controlling the execution of a Silk simulation. The `init( )` method, which is called at the beginning of each simulation, is used to instantiate the `BatchServer` and the first instance of the MyEntity class. Invoking the `start( )` method on each causes their `process( )` method to be invoked after the specified amount of simulated time. Subsequent instances of the `MyEntity` class are introduced into the model when an arriving entity explicitly creates its successor by invoking the `create( )` method. A standard statistical summary report is printed at the end of each simulation run which here summarizes the time-dependent statistics on the input queue for each server and the observational-based statistics on the batch size and system time of entities.

4 SOFTWARE REQUIREMENTS

The Silk simulation extensions to the Java language are themselves implemented entirely in Java. The only requirements for building and executing Silk simulation models are a Java language compiler and run-time Virtual Machine that are compatible with Sun's JDK 1.1 specification of the language. Most commercial simulation software vendors constrain users to a single proprietary and often cumbersome development environment. With Silk users can choose from a variety of professional, third-party Java Integrated Development Environments (IDE's) such as Symantec's Visual Café, Borland's Jbuilder, Sun's Java WorkShop, and Microsoft's Developer Studio. 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 Symantec's Visual Café development environment.

5 SILK LANGUAGE FEATURES

Silk was designed to be a small but powerful general purpose modeling capability that could be extended by users in a straightforward and unrestricted manner. Figure 5 contains a partial listing of the Silk class library and some of the more commonly used methods. What is unique about 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 Silk and the Java programming language to write customized extensions that include more complex process classes such as "transport", "warehouse" or "schedule". And users are encouraged to demonstrate



```
public void process ( ) {
   create( expInterArvTime.sample( ) );
   attArvTime = time;
   queue( queStage1 );
   while ( condition( resServer.getAvailability( ) == 0 ) );
   dequeue( queStage1 );
   seize( resServer );
```

Figure 4: The Visual Café Integrated Java Development Environment.

Silk

SetControlConsole
setDebugLevel
setGlobalTrace
setReInitStats
setRunLength
setReplications
getReplication
setReInitStats
setReInitSystem
setSummaryReport
setAnimation
setSmoothness
setTimeScale
setStepMode

StateVar

getDoubleValue
getValue
setValue
setReInitVal
reInitAll
reInit

Queue

setLabel
getLength
setCapacity
addAnimator
removeAnimator
remove
insert
getContents
reInitAll
reInit

Resource

getAvailability
getNumBusy
setNumActive
setCapacity
setLabel
reInitAll
reInit

Entity

activate
condition
create
delay
dequeue
dispose
endSimulation
halt
process
queue
release
seize
start

TimeDependent

reSet
update
getArea
getAvg
getStdev
getMax
getMin
reSetAll
setLabel
summarize

Observational

reset
record
getSum
getAvg
getStdev
getMax
getMin
getCount
reSetAll
setLabel
summarize

Random

RandomStream
Uniform
Normal
LogNormal
Erlang
Exponential
Gamma
Bernoulli
Discrete

Figure 5: List of Silk Language Features.

and share these extensions with their internal and external clients who can browse and execute Silk models over the network on any hardware platform using a Java-enabled web browser.

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.

6 JAVABEANS AND VISUAL MODELING

Integrating components to create an application is preferable to developing applications from scratch. Component-based applications bring economies of speed in development and testing by capitalizing on previous successes. JavaBeans is a set of classes and programming conventions that constitute a component development model for the Java language.

Beans are designed to be manipulated graphically within visual development environments like Symantec's Visual Café. 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.

Component architectures that exist for other languages are designed primarily for graphical user interface components. JavaBeans, however, can be applied to any aspect of an application. In particular, it is a relatively simple matter to write self-contained, simulation modeling components based on Silk, that automatically make known their functionality and interoperability when incorporated into a JavaBeans visual development environment. Within the 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 not ensure reusability. To exploit the potential that beans have to offer, policies that define the functionality and modes of interoperability that allow beans to be reused must be developed and adhered to. For example, there exist a set of policies and supporting classes in Silk that define the ways in which components must interact with Silk 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 Silk 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 and JavaBeans, however, significantly facilitate the manner in which these issues can be approached - both from a design and implementation standpoint. In combination, they have the potential to raise component model development, interoperability, and reusability, to a new level.

7 SUMMARY

The Java language extensions that constitute 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, users also benefit from the growing number of commercially available professional Java development tools. Silk and JavaBeans also greatly advance both the state of the art and practice of visual modeling with reusable industry specific and company-specific modeling components. These language-level and component-level advances in combination with the ability to distribute and execute models via the Internet 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. 1998. Next Generation Simulation with Java. *Proceedings of the 1998 Winter Simulation Conference*, ed. D. Meideros, E. Watson, J. Carson, and M. Manivannan. IEEE, Piscataway, NJ.

Healy, K. and R. Kilgore. 1998. Java, Enterprise Simulation and the Silk^{TM} Simulation Language. *Proceedings of the 1998 International Conference on Web-Based Modeling & Simulation*, ed. P. Fishwick, D. Hill, and R. Smith. SCS, San Diego CA..

Healy, K. and R. Kilgore. 1997. Silk^{TM}: A Java-Based Process Simulation Language. *Proceedings of the 1997 Winter Simulation Conference,* ed. S. Andradóttir, K. Healy, D. Withers, and B.L. Nelson. IEEE, Piscataway NJ.

Healy, K., R. Kilgore and G. Kleindorfer . 1998. Silk^{TM}: Usable and Reusable Java-Based Object-Oriented Simulation. *Proceedings of the 12th European Simulation Multiconference*. SCS International, Ghent, Belgium.

AUTHOR BIOGRAPHIES

KEVIN J. HEALY is the author of the Silk simulation language and a partner in ThreadTec, Inc.. He received his Ph.D. in Operations Research from Cornell University. He served as Associate Editor for the Proceedings of the 1997 Winter Simulation Conference.

RICHARD A. KILGORE is a partner in ThreadTec, Inc.. 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.

*Silk is a registered trademark of ThreadTec, Inc.