Proceedings of the 2002 Winter Simulation Conference
E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, eds.

COMPONENT BASED SIMULATION MODELING WITH SIMKIT

Arnold Buss

MOVES Institute
Naval Postgraduate School
Monterey, CA 93943, U.S.A.

ABSTRACT

This paper demonstrates how to use Simkit to create Discrete Event Simulation models using a component framework. The component framework is based on a listener design pattern especially useful for simulation models. The objects created are called Listener Event Graph Objects, so the component framework is called LEGO. Simkit is an Open Source package written in Java.

1 INTRODUCTION

The LEGO component framework is powerful, flexible, and promotes designing generic, reusable simulation models (Buss and Sanchez, 2002). Simkit is an implementation of the LEGO framework that supports all its key concepts. This type of component simulation modeling is distinct from the commercial process-oriented modeling environments because of the loose-coupling between components brought by the two listener patterns in the methodology. In addition to bringing much more flexibility to creating models, the loosely-coupled component approach supports substantially more reuse of developed modules (components) than traditional approaches. A particularly useful feature of the approach is the ability to decouple the model dynamics from all uses of the simulation data.

Simkit is based on an Event Graph world view (Schruben, 1983, 1992; Buss, 2001a). Event Graphs are the simplest and most natural way to represent Discrete Event Simulation (DES) models. For a general introduction to DES modeling, see Law and Kelton (2000).

Simkit is a programming toolkit that supports this kind of component-based modeling. In its current form, the simulation modeler interacts with Simkit at the Application Programmer Interface (API) level, in contrast to commercial Graphical User Interface (GUI) environments. A GUI for more intuitive model building in Simkit is currently under development.

Simkit is platform-independent, written in the Java^{TM} programming language, and will run on any reasonably modern operating system. Simkit is copyright under the GNU public license. Simkit is an Open Source package, and may be downloaded from the internet at `<http://diana.gl.nps.navy.mil/Simkit/>`.

2 EVENT GRAPH MODELING

Simkit is designed with a pure discrete event world view. The most natural way to specify Discrete Event Simulation (DES) models is using Event Graphs. We will therefore present a brief review of Event Graphs. More detailed information about Event Graphs may be found in Schruben (1983, 1992) and Buss (2001a).

The defining feature of DES models is that they have state variables whose trajectories in simulated time are piecewise constant. State transitions only occur at discrete time epochs, which are designated as events. The Event List is responsible for determining which events occur and that the appropriate state transitions are executed. The occurrence of an event may trigger the occurrence of other events at later times. These future occurrences of events are implemented in a DES model by placing the appropriate scheduled events on the Event List. The Event List algorithm sorts the events in ascending temporal order and executed the simulation by always

Specification of a DES model therefore consists of

- Defining the state variables
- Defining the state transitions corresponding to events
- Defining the scheduling relationships between events.

The prototypical Event Graph construct is shown in Figure 1. Its interpretation is: When Event A occurs, then if condition (i) is true, Event B is scheduled to occur at the current time + *t*.

There are only two reserved terms in Event Graph methodology: the Run event and a variable representing the current value of simulated time called *simTime* (although Schruben (1992) call it 'clk'). The only special property of the Run event is the fact that it is placed on the Event List at time 0.0.

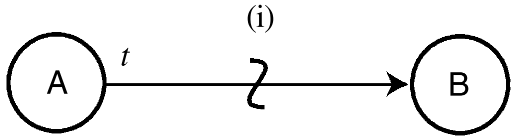

Figure 1: Basic Event Graph Construct

The simplest nontrivial Event Graph is the Arrival Process. The most generic version of the Arrival Process has parameter consisting of a stream of non-negative inter-arrival times $\{t_A\}$ and state consisting of the cumulative arrival count *N*. There is one event (in addition to the Run event) that will be labeled Arrival whose state transition is to increment the arrival count. The Arrival Event has a single scheduling edge that schedules another Arrival event after a delay of t_A. The Arrival Process Event Graph is shown in Figure 2.

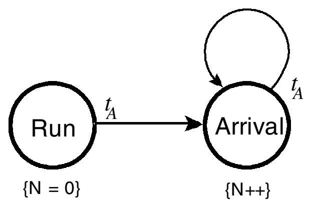

Figure 2: Arrival Process Event Graph

More complicated models can be created using Event Graph methodology (Schruben 1992).

Another useful Event Graph model captures a multiple server queue. The parameters are $\{t_S\}$, the stream of service times, and k, the total number of servers. State variables are S, the number of available servers, and Q, the number of customers in the queue. The Event Graph for the Multiple Server Queue component is shown in Figure 3.

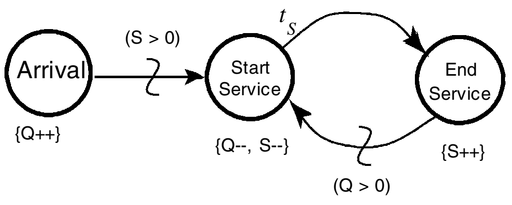

Figure 3: Multiple Server Queue Component

3 BUILDING SIMPLE MODELS IN SIMKIT

Simple DES models encapsulating a single Event Graph are implemented in Simkit by subclassing the `SimEntityBase` class, an abstract base class that implements most of the functionality required by a DES component. There is a direct mapping from an Event Graph model to this subclass.

Every parameter is mapped into a read/write property of the class and every state variable is mapped into a read-only property. At first glance this seems unintuitive, since parameters do not change value during a simulation run, whereas state variables do. State variables change value, but only inside the class itself according to the state transitions specified by the corresponding Event Graph. By making the parameters writable properties, tools for Java Beans may be used to configure the parameters of Simkit objects. In the Arrival Process of Figure 2, the parameter of interarrival times is a RandomVariate instance variable (see Section 6) and the state variable N is implemented as an instance variable of type `int`.

Every event is mapped to a corresponding method with the same name, but with a prefix of 'do.' Thus, a Simkit model corresponding to the Arrival Process of Figure 2 would consist of a single class (probably called ArrivalProcess) that had two methods `doRun()` and `doArrival()`.

State transitions are implemented as expressions that change the value of the corresponding instance variables according to the state transition function. Furthermore, in Simkit models, by convention, all state transitions are accompanied by a `PropertyChangeEvent` being fired. The reasons for this will be discussed in Section 4.2 below.

The final mapping between Event Graphs and a simple Simkit model is the scheduling edges. Each scheduling edge is implemented as a call to the `waitDelay()` method, which is implemented in the SimEntityBase class. The simplest form is `waitDelay(String, double)`. The first argument is the name of the Event and the second is the time delay. Thus, the method corresponding to the Arrival event in Figure 2 is shown in Figure 4.

```
public void doArrival() {
    firePropertyChange("numberArrivals",
        numberArrivals, ++numberArrivals);
    waitDelay("Arrival", iat.generate());
}
```

Figure 4: Source code for Arrival Event

In the code snippet of Figure 4, `iat` is a reference to a RandomVariate instance that generates the interarrival times.

Initialization must be treated specially. The state transitions in the Run event are actually the initialization os state variables. It is better for replications purposes to separate the state transitions and the scheduling in the `doRun()` method. The state transitions are therefore written in a separate method called `reset()`, and the scheduling is implemented using `waitDelay()` invocations in `doRun()`. Thus, the Run event in Figure 2 is implemented by the two methods, as shown in Figure 5.

```
public void reset() {
    super.reset();
    numberArrivals = 0;
}
public void doRun() {
    waitDelay("Arrival", iat.generate());
}
```

Figure 5: Initialization methods for Arrival Process

The programmer never writes code to directly invoke any 'do' method or `reset()`. Instead, every `SimEntity` instance is registered with the Event List class, called `Schedule`. Invoking the static `Schedule.reset()` method invokes the `reset()` method of every `SimEntity` that has been instantiated. That way, every SimEntity need only be responsible for initializing (and maintaining) its own state variables, a considerably smaller task than for the entire simulation.

Scheduling edges that pass parameters to the scheduled are implemented by a form of `waitDelay()` with signature `(String, double, Object[])`. Cancelling edges are implemented by the `interrupt()` statement with either a `(String)` or a `(String, Object[])` signature, corresponding to interrupting the next scheduled event of a given name or to interrupting the next scheduled event with a given name and parameter list.

4 THE LISTENER PATTERN

Using the process described in the previous section, any self-contained Event Graph can be implemented as a Simkit model using a single SimEntity class (Buss, 2002b). Simkit's component model is best described using the concept of software design patterns (Gamma, et al, 1995).

Simkit also implements a simulation component framework that relies on two forms of the Listener Pattern, the SimEventListener and the PropertyChangeListener patterns. The Listener pattern allows the simulation modeler to create simulation components that encapsulate Event Graph logic, then connect the components together to create larger models of greater complexity. This component framework is called Listener Event Graph Objects (LEGO) and is described in Buss and Sanchez (2002). As the name suggests, the Listener pattern is a critical feature of the LEGO component framework. There are two types of Listener pattern used in LEGO: the SimEventListener pattern and the PropertyChangeListener pattern.

4.1 SimEvent Listener Pattern

The SimEventListener pattern is used when the occurrence of an event (SimEvent) in one SimEntity object should stimulate a corresponding SimEvent in another SimEntity object. The listening SimEntity must explicitly register interest in hearing another's SimEvents. The pattern is one that matches the names of the events. When a SimEvent is heard that matches an event of the SimEventListener, then that event is activated (i.e. its state transition is performed, then any scheduling or canceling edges are executed). If no corresponding event is found, then nothing happens.

The SimEvent Listener pattern is implemented in Simkit by the `SimEventListener` interface, which is implemented by a class intended to be a SimEvent Listener, and the `SimEventSource` interface, which is implemented by a class intended to be a source of SimEvents. Simkit's primary base class, SimEntityBase, implements both these interfaces, so instances cam be both sources and listeners of SimEvents.

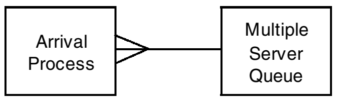

Figure 6: Queueing Component Model

Figure 6 shows an instance of MultipleServerQueue registered as a SimEventListener to an instance of ArrivalProcess, forming a complete queueing model. The occurrence of each Arrival event in the ArrivalProcess triggers the occurrence of the Arrival event in the MultipleServerQueue. This listening relationship is indicated by the line connecting the two components in Figure 6. The arrowhead-like portion near the ArrivalProcess component indicates the direction events are being dispatched, and the shape is also suggestive of a stethoscope, indicating that the MultipleServerQueue instance is listening to the ArrivalProcess for its SimEvents.

4.2 PropertyChangeListener Pattern

The PropertyChangeListener pattern is closely tied to the DES concepts of state and state transition. Every PropertyChangeSource object maintains a list of other objects that are interested in being notified whenever one of its properties changes value. Whenever a property does change value, all registered listeners are notified by a PropertyChangeEvent that is dispatched to each listener. In Simkit, every SimEntity instance can be both a source or PropertyChange events. The "properties" are in fact the state variables of the SimEntity instance. Every time a state changes value, a PropertyChangeEvent should be fired. This convention allows there to be important decoupling between key DES tasks. For example, one consequence discussed below is the fact that DES models constructed in this manner do not need to have any statistics collection built into the model, yet are capable of many more disparate types of data collection than if statistics were in fact built into the model.

The PropertyChangeSource does not know or care what the listeners do when they are notified of the property change. More important, the source object does not know or care what class a listener instance is. The only requirement is that the listener be an object that implements the PropertyChangeListener interface. Thus, there is very loose coupling between the source and the listener in this pattern.

The PropertyChangeListener pattern is directly supported by Java, which defines a PropertyChangeEvent object and PropertyChangeListener interface, as well as supporting classes, such as PropertyChangeSupport, that acts as a proxy for registration of listeners and dispatching of PropertyChangeEvent.

The `SimEntityBase` class provides the convenience method firePropertyChange(), as shown in Figure 4. The general signature is `(String, Object, Object)`, with the name of the state (property), the old value, and the new value, respectively. Note that the name of the property as given need not coincide with the name of the state variable. This feature is important in distinguishing state changes made by different instances of the same SimEntity class.

One of the most important uses of the PropertyChangeListener pattern is in collecting statistics in a non-invasive manner. Simkit provides several PropertyChangeListener classes that estimate statistics on a given property by name. They each keep internal variables for running totals, counts, etc. When an instance hears a PropertyChangeEvent, it checks to see whether the property name is the one it is listening for. If so, it simply updates its variables, and if not then it does nothing. The code in Figure 7 shows two examples of this. The variable `queue` is an instance of a Multiple Server Queue component, as modeled by the Event Graph of Figure 3. The first statistics object (`niqStat`) listens for changes in a property called `numberInQueue`. and updates its statistical counters in a time-varying manner. The second (`diqStat`) listens for a property called `delayInQueue` and updates its statistical counters in a tally manner. The queue object is not aware that any statistics are being computed from its state transitions, it simply fires the appropriate PropertyChangeEvents whenever the state transitions occur.

```
SimpleStatsTimeVarying niqStat =
new SimpleStatsTimeVarying("numberInQueue");
SimpleStatsTally diqStat =
new SimpleStatsTally("delayInQueue");
. . .
queue.addPropertyChangeListener(niqStat);
queue.addPropertyChangeListener(diqStat);
```

Figure 7: Code for Statistics PropertyChangeListeners

The statistics objects are not aware of the fact that it is a queue they are collecting values from. Any PropertyChangeSource instance that fires PropertyChangeEvents of the given name will be acted upon.

5 BUILDING LEGO COMPONENT MODELS

The Listener patterns described in the previous section are the critical elements in Simkit's implementation of the LEGO component framework. In general, the way in which components are connected to each other is more a important feature than the design of the components themselves. The SimEventListener pattern to connect LEGO components is sufficient to obtain the required generality and flexibility.

For example, the queueing model shown in Figure 6 connects two components together using the SimEventListener pattern. Neither component needs to know any specifics about the other, apart from the fact that they are sources and listeners of SimEvents. Other SimEventListeners could be listening to the Arrival Process instance – instead of, or in addition to the Multiple Server Queue instance. Likewise, the Multiple Server Queue instance could be listening to other SimEvent sources – again, instead of or in addition to the Arrival Process instance.

The SimEventListener pattern allows the occurrence of an event in a simulation component to trigger the occurrence of an event of the same name (and signature) in a SimEventListener. However. situations arise in which the simulation modeler desires to have an event in one SimEntity trigger an event of a different name. This is accomplished without having to edit either class through the use of lightweight adaptor, or "bridge" components. The bridge component listens for a SimEvent and schedules another event with zero delay. A bridge component maintains no state, so there is never a need to listen to its PropertyChange events.

For example, a model of Tandem Queue can easily be created from instances of the Multiple Server Queue in Figure 3 using a bridge component, as shown in Figure 8.

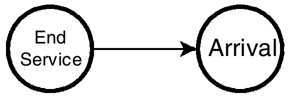

Figure 8: Bridge for Tandem Queue

The code for the bridge class in Figure 8 is essentially a single line, as shown in Figure 9.

```
public void doEndService() {
    waitDelay("Arrival", 0.0); }
```

Figure 9: Code for Tandem Queue Bridge

Instances of the bridge class can be used to transmit EndService events of one queue to Arrival events of the next queue. The model is configured as shown in Figure 10 below. Observe that the first queue in Figure 10 would have to be listening to s SimEventSource for its Arrival events, since it produces none itself.

```
MultipleServerQueue queue1;
MultipleServerQueue queue2;
Bridge bridge;
. . . // instantiation of objects.
queue1.addSimEventListener(bridge);
bridge.addSimEventListener(queue2);
```

Figure 10: Configuring a Tandem Queue Model

Simkit provides a class to provide simple Bridge functionality called `simkit.Bridge`. Objects of this class is instantiated with two strings, one for the name of the listened-to event and the second for the event that is scheduled. Thus the modeler can either write a custom bridge (as in Figure 8) or simply instantiate Simkit's `Bridge` class.

The fact that there are two instances of a Multiple Server Queue in the component Tandem Queue model introduces a potential difficulty regarding the state changes of the two instances. Although they are distinct objects and thus maintain two distinct sets of state variables, the PropertyChangeEvents fired by each object have identical names. This is not confusing to the model itself, since the PropertyChangeListener pattern is based on individual objects. However, there may be difficulty determining which instance is firing which event when log files or verbose output is being examined. To mitigate this difficulty, there is support in Simkit for PropertyChange renaming based on a simple namespace concept. Instances of PropertChangeNamespace listen for PropertyChangeEvents and when they hear one, prepend a name to the property name, then refire the event. Thus, a simple namespace hierarchy can be seamlessly incorporated into the model.

This is illustrated in Figure 11 for a Tandem Queue model. There are two levels: each instance of Multiple Server Queue is given a label (in this case an integer for its position in the line) and the name "`stationi`" is added. The Tandem Queue also has a PropertyChangeNamespace instance that adds the name "`tandem`." Thus, Figure 11 shows the `numberInQueue` state for station 1 of the tandem queue has just decreased from 1 to 0. Yet the instance of MultipleServerQueue simply fires a PropertyChangeEvent called "`numberInQueue`," unaware of the fact that additional names will be added to put it into context.

```
tandem:station1:numberInQueue:1 => 0
tandem:station1:numberAvailableServers:2 => 1
Time: 7.886  Current Event: StartService  [5]
 ** Event List --  **
8.387   EndService
9.729   EndService
10.801  Arrival
15.471  EndService
100.000 Stop
 ** End  of Event List --  **
```

Figure 11: Namespace PropertyChange Events

Components should be treated as "black boxes," meaning that they should be able to be used without knowledge of how they are internally constructed. Thus, LEGO components themselves can be constructed from other LEGO components in a manner transparent to their user. In other words, the simulation program should not be able to distinguish between two components implementing identical functionality, one of which was constructed monolithically and the other constructed from components.

Thus, we can create a LEGO component for a tandem queue by wrapping the above-described sequence of MultipleServerQueue and Bridge instances inside a component that exposes other events as the starting and ending points. So, using "Arrival to System" and "Exit System" as these two events, the tandem queue LEGO component is as shown in Figure 12.

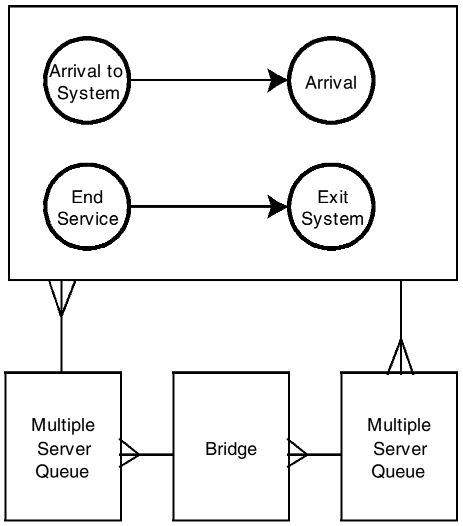

Figure 12: Tandem Queue LEGO Component (Two Queues)

In Figure 12 an Arrival to System event in the Tandem Queue is converted to an Arrival event, which is heard by the first MultipleServerQueue instance. Similarly, the End Service event in the last queue is heard by the Tandem Queue and converted to an Exit System event. To a user of this LEGO, however, the component appears to only have two events, as shown in Figure 13

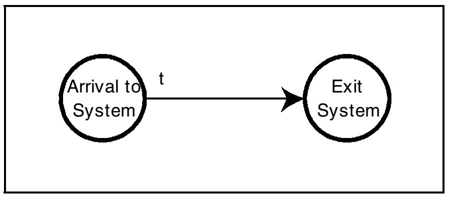

Figure 13: Tandem Queue LEGO

How the LEGO component schedules the Exit System event is internal and details of it should not be required for its use. The source code for the events in Figure 12 is shown in Figure 14.

```
public void doSystemArrival() {
  firePropertyChange("tandem:arrival",
    numberArrivals, ++numberArrivals);
  waitDelay("Arrival", 0.0);
}
public void doEndService() {
  waitDelay("SystemExit", 0.0);
}
public void doSystemExit() {
  firePropertyChange("tandem:exit",
    numberServed, ++numberServed);
}
```

Figure 14: Source code for LEGO Tandem Queue Events

The Tandem Queue LEGO component is implemented in Simkit so that the number of workstations in the line is data. The addWorkCenter() method ensures that the proper bridges and listeners are established, as shown in Figure 15.

```
public void addWorkCenter
    (MultiServerQueue workcenter) {
  PropertyChangeNamespace pcn =
    new PropertyChangeNamespace(workcenter,
      "station" + servers.size());
  workcenter.addPropertyChangeListener(pcn);
pcn.addPropertyChangeListener((PropertyChange
Listener) property);
  if (servers.isEmpty()){
    this.addSimEventListener(workcenter);
  }
  else {
    Bridge bridge =
      new Bridge("EndService", "Arrival");
    SimEntity last =
      (SimEntity)servers.getLast();
    last.addSimEventListener(bridge);
bridge.addSimEventListener(workcenter);
last.removeSimEventListener(this);

  }
  workcenter.addSimEventListener(this);
  servers.add(workcenter);
}
```

Figure 15: Source Code for Adding a Workcenter to Tandem Queue LEGO

6 RANDOM VARIATE GENERATION FRAMEWORK

The framework for generating random variates and random numbers in Simkit is of independent interest, not because of any innovations in the algorithms, but because of the degree of modeling flexibility afforded the simulator building models. This flexibility is a crucial factor in the LEGO component framework, since having a different component class for every conceivable random variable is undesirable. It is likewise unsuitable to only have a small handful of possible probability distributions from which to choose,. Any component that uses randomness must be able to be configured for any type of pseudo-random stream (including constants and correlated streams).

The objectives of the framework were as follows:

- To be capable of running a model with simulation components using different probability distributions without any "invasiveness" – that is, re-editing of the source code and recompilation.
- Changes should be done using input data only, and which probability distribution to us for a particular execution should be able to be decided at the last possible moment (i.e. at runtime);
- To enable a simulation component to use a random variate algorithm that was implemented subsequent to the component's creation, again without editing or recompilation.
- To be able to substitute a different random number generator for a given random variate algorithm without having to edit or recompile.
- To be able to implement a new random number generator and use it with any existing random variate algorithm, again without editing or recompiling the code that implements the random variate algorithm.

Some secondary goals that can be achieved include:

- The ability for a simulation component to be driven by either a random variate generating algorithm or by trace data without recompilation of the component.
- The ability for any part of a component modeled as a random to be also executed deterministically, again without recompilation.

The objectives described above are met in Simkit by defining two interfaces and two generic object factories to produces appropriate instances.

The objectives for generating random variates and using them in simulation components without recompilation is met by defining a `RandomVariate` interface that specifies a method called `generate()` that returns the generated random variate. Instead of a single class with many different methods for generating different types of random variates, a separate class that implements the `RandomVariate` interface must be written for each new random variate.

Since `RandomVariate` is an interface, it cannot be instantiated directly. Instead, instances of a `RandomVariate` class are obtained from the `RandomVariateFactory` using the `getInstance()` method. Arguments to `getInstance()` are a `String`, the name of the concrete `RandomVariate` class desired, an `Object[]` (array of objects) that are the desired parameters, and a `long`, the starting seed. These are generic data, and may be easily extracted from text input or from XML files. An example of obtaining a `RandomVariate` instance from `RandomVariateFactory` is shown in Figure 16.

```
RandomVariate rv =
  RandomVariateFactory.getInstance("Gamma",
    new Object[] { new Double(1.7),
      new Double(3.2) }, 12345L);
```

Figure 16: Obtaining RandomVariate instance

The ability of `RandomVariateFactory` to use generic data makes it particularly straightforward to use XML as input data. A single class is capable of converting XML input to RandomVariate instances. An example of some XML that cam be used this way is shown in Figure 17.

```
<RandomVariate>
  <class>simkit.random.GammaVariate</class>
  <parameter
    name="alpha" class="java.lang.Double"
             value="1.7"></parameter>
  <parameter
    name="beta" class="java.lang.Double"
             value="3.2"></parameter>
  <seed>12345</seed>
</RandomVariate>
```

Figure 17: Portion of XML Document to create RandomVariate Instance

The RandomVariate interface contains a method `generate()` that returns the next generated pseudo-random variate. This is an example of the Command Pattern (Gamma et al, 1995) and is what enables the generic configuration of LEGO components that use randomness. Since any class implementing `RandomVariate` contains the `generate()` method, the probability distribution can be changed simply by replacing the instance used. Although configuration files may have to be edited, the component itself does not. Examples of the generate() method have been shown in the code in Figure 4 and Figure 5

This structure for generating random variates makes LEGO components built with Simkit robust against limitations in Simkit's random variate classes. Classes implemented outside Simkit, as long as they implement the `RandomVariate` interface, can be used as "first-class citizens" in a simulation model. The `RandomVariateFactory` will happily generate instances of such user-defined classes and the LEGO components will likewise use them without complaint.

7 CONCLUSIONS

Simkit is an implementation of the LEGO component framework that enables the simulation modeler to create flexible, robust, and reusable components. These simulation components can be easily assembled into more complicated models. Since a component itself can consist of assembled sub-components, the framework is very scalable. Simkit is copyright under the GNU public license and may be downloaded from the web at `<http://diana.gl.nps.navy.mil/Simkit/>`.

ACKNOWLEDGEMENTS

The author thanks Paul Sanchez for many discussions and feedback. This work was supported by a grant from the Air Force Office of Scientific Research. This support is gratefully acknowledged.

REFERENCES

Buss, A. 2001a. Basic Event Graph Modeling. *Simulation News Europe*. 31:1-6.

Buss, A. 2001b. Event Graph Modeling with Simkit. *Simulation News Europe*. 32/33:15-25.

Buss, A. and Sanchez, P. 2002. Building Complex Models With Legos (Listener Event Graph Objects). In *Proceedings of the 2002 Winter Simulation Conference*, ed E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes. Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995 *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA,.

Law, A. and D. Kelton. 2000. *Simulation Modeling and Analysis, Third Edition*, McGraw-Hill., Boston, MA.

Schruben, L. 1983. Simulation Modeling with Event Graphs. *Communications of the ACM.* 26:957-963.

Schruben, L. 1992. *Sigma: A Graphical Simulation Modeling Program*. The Scientific Press. San Francisco, CA.

AUTHOR BIOGRAPHY

ARNOLD H. BUSS is a Research Assistant Professor in the MOVES Institute at the Naval Postgraduate School. He received a B.A. in Psychology from Rutgers University, his M.S. in Systems Engineering from the University of Arizona, and a Ph.D. in Operations Research from Cornell University. His recent work has involved Component-Based software design.