2004

Simkit Analysis Workbench for Rapid Construction of Modeling and Simulation Components

Buss, Arnold

Monterey, California: Naval Postgraduate School.

Simkit Analysis Workbench for Rapid Construction of Modeling and Simulation Components

Arnold Buss
MOVES Institute
700 Dyer Road
Naval Postgraduate School
Monterey, CA 93943
abuss@nps.edu

Keywords: Discrete Event Simulation; Event Graphs; Simulation Components

***ABSTRACT**: A recurring dilemma in the use of simulation models for analytic support of decision-making has been the length of time required to build the simulation model. Although emerging simulations have improved over legacy models, the problem persists. It is particularly difficult to create a simulation model using existing tools that captures only the desired elements affecting the performance measures to be studied. Additionally, there is often a lack of rigorous methodology underlying the model's design. Simkit is an Object-Oriented API for creating Discrete Event Simulation (DES) models in Java. Based on proven Event Graph methodology, Simkit has been used to quickly create models in a wide range of areas, including logistics and operational support, undersea models, and models that evaluate algorithms for allocation of weapons and sensors to targets in ground combat. Simkit's component-oriented approach facilitates the composition of models using some pre-built and some custom simulation components. This work demonstrates a Graphical User Interface (GUI) for the creation and analysis of Simkit models. It utilizes XML to represent the components, so there is built-in interoperability with many other tools. Specifically, simulation components and models designed in this manner will be capable of interacting with models with Extensible Modeling and Simulation Framework (XMSF) capabilities. In component design mode, a new component is created by drawing the Event Graph and filling in parameters, so that the simulation modeler need not be a sophisticated programmer. In component construction mode, components are hooked together to create a model. In analysis mode, the models are exercised and run according to the desired experimental design. The workbench also has a number of exemplar models that have been extracted from recent NPS Master's Theses.*

1. Introduction

The use of simulation models for analysis is becoming increasingly important. Many important questions of interest require a model that represents features challenging, if not impossible, for analytic or algorithmic approaches ([1]). Discrete Event Simulation (DES), described in the following section, is often used as the simulation methodology of choice for analysis. One of the challenges to using DES in analysis is the amount of time and effort devoted to creating the model in the first place.

Simkit is a Java-based package that enables faster development of DES programs. Simkit is oriented towards Event Graph Methodology, first proposed by Schruben in 1983 ([2]), which is an ideal balance between expressiveness and simplicity for representing DES models. Simkit has demonstrated its efficacy in implementing DES models by the more than thirty Masters theses at the Naval Postgraduate School which have used Simkit. Simkit is also utilized fro its core DES functionality in the U.S. Army's COMBAT^{XXI} simulation.

However, Simkit requires a fair amount of Object-Oriented programming expertise, specifically in Java. It was desirable to reduce this dependence on programming, which would also allow the modeler to focus more on the structure and content of the model being developed rather than on programming issues. This paper discusses a workbench for creating Simkit models using a graphical user interface. It focuses particularly on Viskit, a component of the workbench for creating DES models by drawing Event Graphs and filling in forms. Viskit thus has the potential for supporting even more rapid development of models than can be accomplished using the current Simkit practice of writing Java code.

The remainder of this paper is organized as follows. First Event Graph methodology will be briefly covered, followed by a description of Simkit. The Simkit Analysis

Workbench components will be presented, starting with an XML representation of Event Graph models, followed by the two components of Viskit, the Event Graph editor and the Assembly editor.

2. Event Graph Methodology

Discrete Event Simulation (DES) methodology is a way of modeling a situation in a certain stylized manner ([1]). Two elements of DES are noteworthy. First, DES models advance time according to the Next Event rule. A list of future events (the "Event List") holds the pending list of scheduled future events at any time point. Rather than advancing time in discrete, uniform increments, the simulation time is advanced to that of the next scheduled event. The second identifying element is that the state variables (defined below) stay constant between events, and at events change value according to a predefined state transition function for the occurring event. This state transition occurs instantaneously in simulated time units.

Event Graph methodology is a way of formally representing DES models ([2]). An Event Graph model consists of four elements: A collection of parameters, a collection of state variables, a collection of events (or state transitions) and a collection of scheduling relationships between events.

Parameters are elements that do not change and do not have the possibility of changing in the course of a single simulation replication. Examples include the total number of servers in a multiple queueing system, the number of workstations in a serial production line, etc. For modeling purposes, a sequence of values, even a pseudo-random one, may be considered to be a single parameter. In that case, even though different values may be generated, the sequence as a whole stays unaltered.

A state variable is an element that changes, or at least has the possibility of changing, in the course of a single simulation replication. As mentioned above, the rule by which a state changes value is pre-specified by a state transition function, which occurs when the corresponding event "occurs" in the simulation run.

An Event is a way of labeling or identifying each state transition function. The collection of Events describes every possible change of value in that simulation model. State variable can only change value during the execution of an Event, and an Event always occurs in 0 simulated time. Thus, time only passes between events, never during an event.

Events are placed on the Event List for possible occurrence at some future scheduled time in the simulation. An Event Graph describes this scheduling relationship by specifying which Events (if any) are scheduled when each Event occurs. A second scheduling relationship involves removing a previously scheduled Event from the Event List. These scheduling relationships may be represented as a direction graph, an Event Graph, in which the Events are the nodes and the scheduling relationships form the edges. The two types of scheduling relationships are shown in Figure 1.

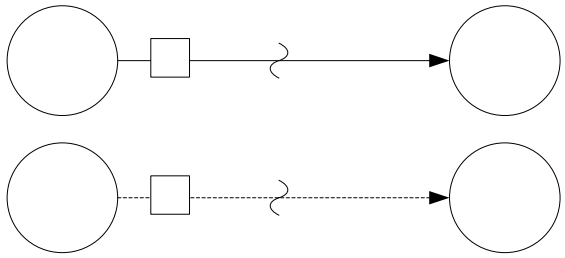

Figure 1 Basic Event Graph Constructs

The top construct in Figure 1 is a scheduling edge between Events A and B; the elements on the edge are a boolean condition (i), a time delay t, and a parameter expression j. Event B has an argument (k), which can be thought of in the same way as the list of formal parameters in a method definition. The scheduling edge reprehension has the following interpretation. When Event A occurs, then if boolean condition (i) is true, then Event B is scheduled to occur t time units in the future. When Event B occurs, then its argument(s) k are set to the value of the parameter(s) j at the time Event B was scheduled.

There is only one special event in Event Graph methodology, the Run event. Every Event Graph model has at least one Run event, and that event is assumed to be placed on the Event List at time 0.0. If there were no such construct, the Event List would start empty, and the simulation would immediately end. "Run" is analogous to a "main" method in C or Java programming, providing a starting place for the model to run. Once the Event List algorithm starts, the Run event is processed like any other event. Its state transition should set the initial values of all state variables, and it should then schedule whatever events are necessary, as determined by the specifics of the particular model.

2.1. Event Graph Examples

To reinforce the event Graph modeling paradigm, we will present several simple examples.

2.1.1. Arrival Process

The simplest Event Graph model has one parameter, one state variable, and a single event (in addition to the Run event). This model represents something of interest that occurs repeatedly throughout a run. Examples include the arrival of customers to a server , the arrival of jobs to a machine, or the arrival of calls for fire to an artillery battery. The parameter consists of a sequence of interarrival times $\{t_A\}$, which can be random or deterministic. The state variable N represents the cumulative number of arrivals that have occurred at any point in time. The state transition when the event of interest occurs is simply that the cumulative count gets incremented by 1. The Event Graph is shown in Figure 2.

Figure 2 Arrival Process Event Graph

2.1.2. Multiple Server Queue

Customers arrive to a service facility according to an arrival process and are served by one of k servers. Customers arriving to find all servers busy wait in a single queue and are served in order of their arrival. The parameters are: $\{t_A\}$ = interarrival times; $\{t_S\}$ = service times; k = total number of servers. The state variables are: Q = # of customers in queue; S = # of available servers. The Event Graph is shown in Figure 3.

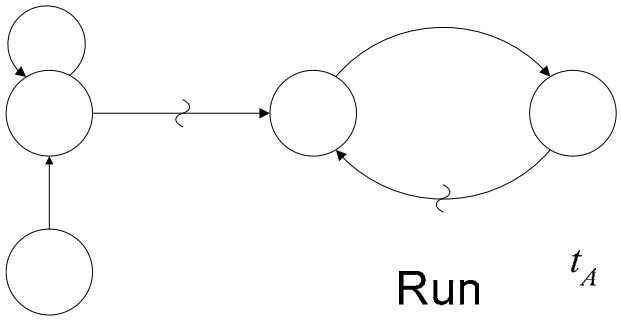

Figure 3. Multiple Server Queue

2.1.3. Server with Reneges

This model illustrates passing parameters on edges and canceling edges. Impatient customers arrive to a server and join a queue. Each customer is only willing to wait a certain amount of time in queue before leaving (the amount of time could vary from customer to customer). The parameters are identical to the multiple server queue model of the previous section, with the addition of a sequence $\{t_R\}$ of "renege" times. The state variables are: S = # available servers, N = customer number (unique to each arriving customer), q = a fifo container holding the unique customer numbers, R = total number of reneges. The Event Graph for just the server portion of the model is shown in Figure 4.

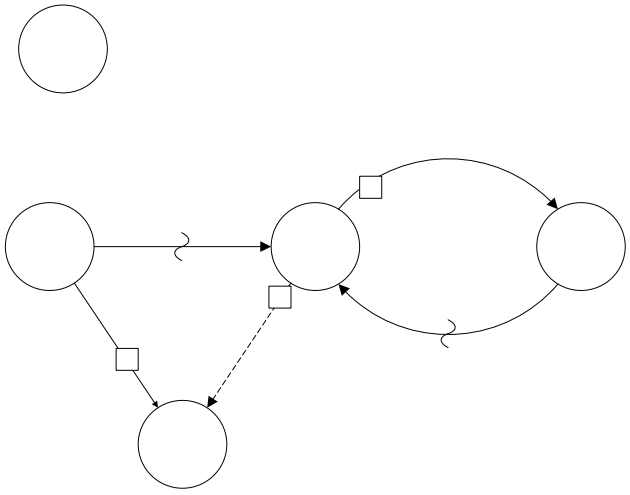

Figure 4. Server with Reneges Event Graph

2.2. Event Graph Components (LEGO)

As powerful as the basic Event Graph paradigm is, it does not scale up to very large models effectively. This is primarily because in their original form, Event Graphs are essentially monolithic, and as the number of state variables and events increase, the size of the Event Graph becomes larger and larger, ultimately becoming virtually impossible to understand, modify, or verify.

A component framework for Event Graphs has been developed based in the Listener software design pattern. The listener pattern enables simulation components to be created in which state variables and parameters are encapsulated in an object and the essential logic is represented by an Event Graph snippet. Since the components are connected using "listening", they have been dubbed "Listener Event Graph Objects", or LEGOs (see [4]).

There are two flavors of the Listener patter used in the LEGO framework: SimEventListener and PropertyChangeListener.

The SimEventListener pattern is implemented by having one simulation component register interest in "hearing" the SimEvents of another. Whenever an event scheduled by the source component "occurs" (that is, bubbles to the top of the Event List and is processed), then all SimEventListeners are notified and the event is passed to each one. The SimEventListener responds to this event in exactly the same way as if it had scheduled it. That is, state transitions are performed, followed by scheduling and canceling of events, as defined by the simulation component. This mechanism provides extremely loose coupling of components, and enables much larger models to be built by connecting components together. The SimEventListener relationship is illustrated in Figure 5.

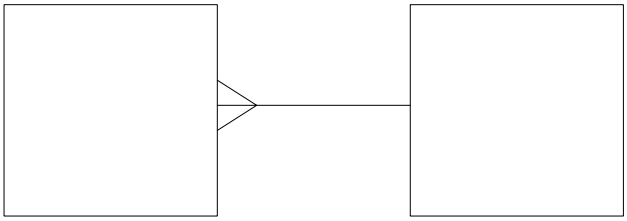

Figure 5. SimEventListener Pattern

The PropertyChangeListener pattern is similar. Instances of a PropertyChangeListener component register interest in "hearing" the PropertyChange events of a SimEntity component. When the source object fires the PropertyChange Event, all registered PropertyChangeListener objects are notified and passed the PropertyChange event. As with the SimEventListener pattern, the source neither knows nor cares what the listener objects do with the event, nor do the listeners care which object fired it. The primary difference between a SimEventListener and a PropertyChangeListener, as used in Simkit, is that one SimEvent is dispatched whenever an event occurs, whereas a PropertyChangeEvent is fired whenever a state variable changes value. A SimEvent carries information about the name of the event, its scheduled time (normally the current value of simulated time, and any parameters that had been passed when it was scheduled. A PropertyChangeEvent has only the name of the property, its new value, and (usually) its "old" value. Thus, a PropertyChangeEvent only has information about a particular state transition. In general, there may be many PropertyChangeEvents fired during when a particular event occurs. On the other hand, any event may occur without any state variables changing value. In such a case, a SimEvent would be dispatched, but no PropertyChangeEvent would be fired.

The PropertyChangeListener pattern is used primarily in the Simkit implementation (discussed in the following section) to decouple the dynamics of a model from data collection and estimation of statistics. The convention is that whenever a state variable changes its value, the object in which it resides fires a PropertyChangeEvent. If the analyst wishes to collect statistics, for example, based on a set of state variables, then statistics PropertyChangeListeners are instantiated and registered with the appropriate SimEntity objects. Thus, data may be gathered in a non-invasive manner, and a LEGO component need not be modified in order for a different set of statistics to be gathered on it.

In fact, this mechanism virtually ensures that any performance measure possible can be estimated from a DES model implemented in this manner. Only state variables change throughout a given simulation run, and if each state transition can be observed, then all the information about state trajectories can be extracted, again without having to modify the LEGO components themselves. Since all measures for a DES are some function of its state trajectories, there is no feasible measure that cannot be estimated.

3. Simkit

Simkit is a collection of Java libraries that support implementing Event Graph and LEGO models. There is a direct correspondence between an Event Graph or a LEGO model and a Simkit program. Each LEGO template is represented by a subclass of SimEntityBase, and each of the four elements of an Event Graph are represented within the class by the correspondence shown in Table 1.

Event Graph	*Simkit*
Parameter	Private instance variable
State Variable	Protected instance variable
Event	"do" Method
Scheduling edge	Call to "waitDelay"

Table 1. Event Graph/Simkit Correspondence

Simkit makes it straightforward to implement Event Graph models for execution. The Event Graph and the SimEntity class contain exactly the same information, and the mapping in Table 1 makes it extremely simple to develop the program itself. For example, the Simkit code for the ArrivalProcess is shown below (comments and ancillary statements have been omitted for clarity):

```
public class ArrivalProcess extends SimEntityBase {
```

```
  private RandomVariate interarrival;
  protected int numberArrivals;
  public void reset() {
    numberArrivals = 0;
  }
  public void doRun() {
    waitDelay("Arrival", interarrival.generate());
  }
  public void doArrival() {
    firePropertyChange("numberArrivals",
        numberArrivals, ++numberArrivals);
    waitDelay("Arrival", interarrival.generate());
  }
}
```

Figure 6. Code for ArrivalProcess

Further information about building Event Graphs with Simkit may be found in [5].

Simkit implements the Listener patterns of the LEGO component framework by means of the SimEventListener and SimEventSource interfaces (for SimEventListener pattern) and PropertyChangeSourse and PropertyChangeListener interfaces (for the PropertyChangelistener Pattern). Note that the PropertyChangeListener interface is actually part of Java. The SimEntityBase abstract base class is both a SimEventSource and a SimEventListener, as well as a PropertyChangeSource. Thus, both SimEventListenersand PropertyChangeListeners can be registered with an instance of a subclass ofSimEntityBase.

4. XML Representation

The visual Event Graph tool requires a format for saving its components. Although Java source code files might be considered a natural choice, XML is a superior one because of its universality, the ability to be parsed by a plethora of software, its being a natural fit for Web Services, and for the ability to transform XML documents via stylesheet transforms into any one of a number of formats.

As with Simkit models, there is a natural mapping between Event Graph elements and Elements in an XML document. The key elements are shown in Table 2.

Event Graph	*XML*
Parameter	Parameter Element
State Variable	State Variable Element
Event	Event Element
Scheduling edge	Schedule Element

Table 2. Event Graph to XML Mapping

With a one-to-one correspondence between Event Graph components and Simkit classes, and between Event Graph Components and XML documents, it is straightforward to create executable programs from the XML files.

The XML shown in Figure 7 shows that the same information is being stored as in the Java source code of Figure 6.

```
<SimEntity name="ArrivalProcess" version="0.1"
xsi:noNamespaceSchemaLocation="http://diana.gl.nps.navy
.mil/Simkit/simkit.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <Parameter type="simkit.random.RandomVariate"
name="interarrivalTime"/>
    <StateVariable name="numberArrivals" type="int"/>
    <Event name="Run">
        <StateTransition state="numberArrivals">
            <Assignment value="0"/>
        </StateTransition>
        <Schedule priority="0.0"
delay="interarrivalTime.generate()" event="Arrival"/>
        <Coordinate y="60" x="40"/>
    </Event>
    <Event name="Arrival">
        <StateTransition state="numberArrivals">
            <Assignment value="numberArrivals + 1"/>
        </StateTransition>
        <Schedule priority="0.0"
delay="interarrivalTime.generate()" event="Arrival"/>
        <Coordinate y="60" x="250"/>
    </Event>
</SimEntity>
```

Figure 7. XML for Arrival Process SimEntity

The simulation model is described via a "SimkitAssembly" element, which specifies which objects are to be instantiated, what the parameter values will be, and establish the listener relationships. And example of a SimkitAssembly is shown in Figure 8.

```
<SimkitAssembly version="1.0" package="pkg"
name="MultipleServerQueue"
xsi:noNamespaceSchemaLocation="http://diana.gl.nps.navy
.mil/Simkit/assembly.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <SimEntity name="arrival"
type="simkit.examples.ArrivalProcess">
        <FactoryParameter
factory="simkit.random.RandomVariateFactory"
type="simkit.random.RandomVariate">
            <TerminalParameter value="Exponential"
type="String"/>
            <MultiParameter type="Object[]">
                <TerminalParameter value="1.5"
type="java.lang.Double"/>
            </MultiParameter>
            <TerminalParameter
value="simkit.random.CongruentialSeeds.SEED[0]"
type="long"/>
        </FactoryParameter>
    </SimEntity>
    <SimEntity name="server"
type="simkit.examples.Server">
        <TerminalParameter value="2" type="int"/>
        <FactoryParameter
factory="simkit.random.RandomVariateFactory"
type="simkit.random.RandomVariate">
            <TerminalParameter value="Gamma"
type="String"/>
            <MultiParameter type="Object[]">
                <TerminalParameter value="2.5"
type="java.lang.Double"/>
                <TerminalParameter value="1.2"
type="java.lang.Double"/>
            </MultiParameter>
            <TerminalParameter
value="simkit.random.CongruentialSeeds.SEED[1]"
type="long"/>
        </FactoryParameter>
    </SimEntity>
    <SimEventListenerConnection listener="server"
source="arrival"/>
</SimkitAssembly>
```

Figure 8. Simkit Assembly Example

5. Viskit

Viskit is a graphical front end for creating, editing, and composing DES simulation models using Event Graphs and the LEGO framework Viskit is very early in its development cycle, and it is anticipated that there will be many changes as experience is gained using the tool. Nevertheless, Viskit currently implements all the basic functionality required to implement the kind of DES models described in previous sections. This section will provide an overview of the Viskit tool and its capabilities.

5.1. Event Graph Editor

The Event Graph Editor is used to create Event Graph components by drawing the Event Graph on a palette and running inspectors to create parameters, state variables, and edit the event nodes and scheduling/canceling edges. An empty EventGraph editor is shown in Figure 9.

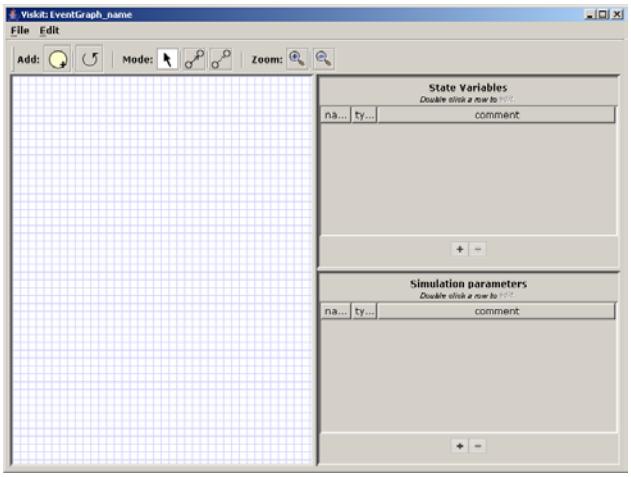

Figure 9. Empty Event Graph Editor

The Event Graph palette is on the left, and the two right panels are for defining state variables (top) and parameters (bottom). A new event is created by dragging the yellow node icon from the toolbar to the palette. Scheduling and canceling edges are created by selecting which type of edge to be drawn on the toolbar and then dragging the mouse from the scheduling event to the scheduled event. Figure 10 shows a completed Event Graph component corresponding to the server with reneging customers model in Figure 4.

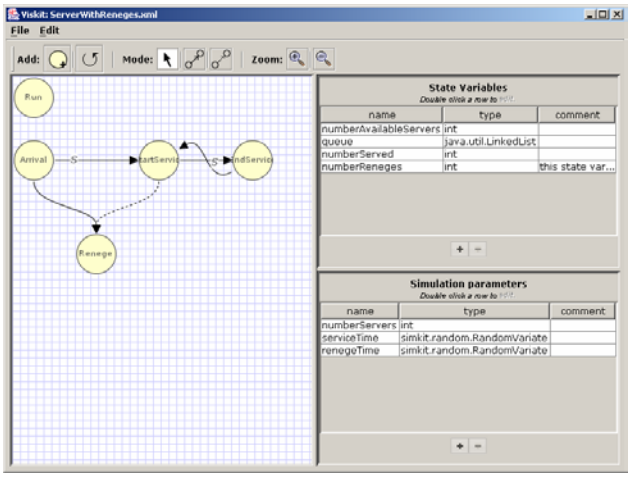

Figure 10. Server With Reneges Event Graph

shows the Node inspector which is used to input, display, and edit the data associated with the node.

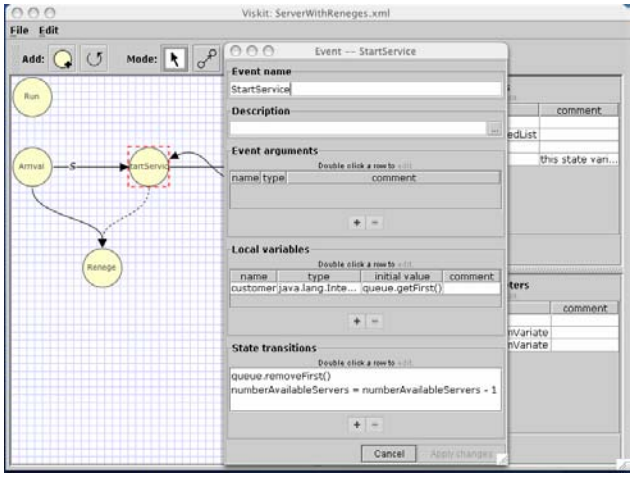

Figure 11. Event Node Inspector

The Node inspector can be used to change the name of the event and to define state transitions. The interface for state transitions ensures that only state variables can be modified. Variables which are local to the event may be defined for convenience. Finally, arguments to the event are also defined. An instance of the Beanshell interpreter is used to verify that user input consist of legitimate expressions, meaning that all variables have been specified (parameters, state variables, or local variables) and that all expressions are syntactically correct.

Figure 12 shows the edge inspector, which is used to input, display, and edit information about the edges.

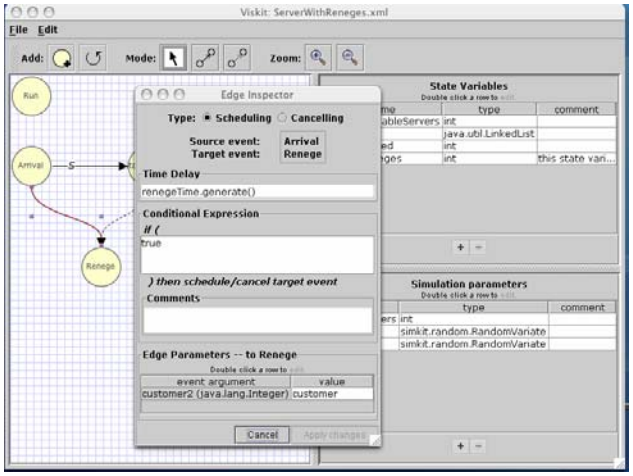

Figure 12. Edge Inspector

As with the node inspector, Beanshell is used to verify all expressions entered in free form. The source and target events are displayed, but cannot be edited from the edge inspector. The time delay and boolean conditions are filled in by the user as freeform expressions (which are verified, as described above). The possible edge parameters and associated types are filled in from the signature of the target event, ensuring that the signature of the edge matches the scheduled (or cancelled) event.

The Event Graph Editor normally saves its components in XML format, according to the schema described in Section 4. Simkit Java code can also be generated and saved for separate compilation, as shown in Figure 13.

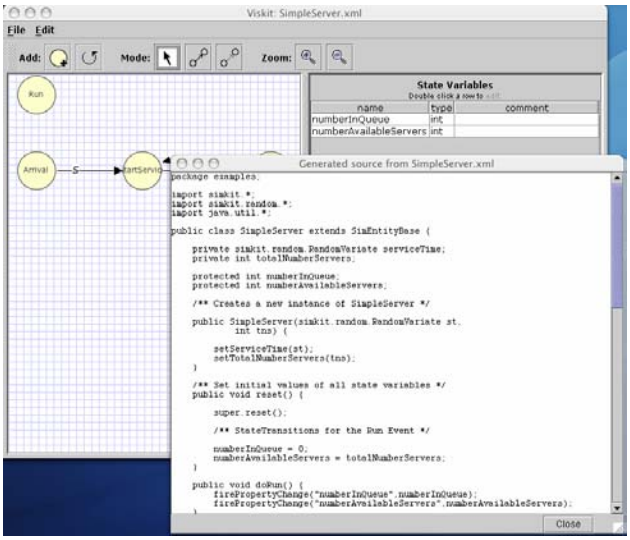

Figure 13. Generated Simkit Code

5.2. Assembly Editor

The Assembly Editor is used to compose DES models using Event Graph components. The Assembly Editor also uses a drawing palette and inspectors to populate the model, but the meaning of the nodes and edges are different. The Assembly Editor can utilize components created using the Event Graph Editor or compiled Java classes that have been created elsewhere. The Assembly Editor appears empty when first opened, as shown in Figure 14.

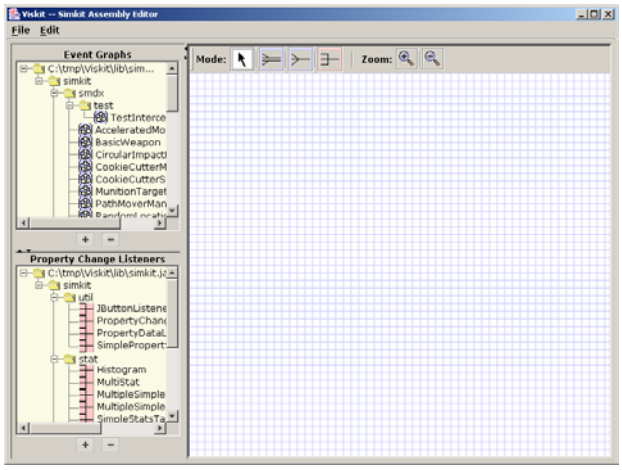

Figure 14. Empty Assembly Editor Window

The palette is on the right (to make it easy to distinguish whether one is using the Assembly or the Event Graph Editor) and the left panels are populated by Event Graph classes (top left) and PropertyChangeListener classes (bottom left). Additional classes may be added or removed by use of the '+' and '-' buttons. Dragging an item onto the palette signals an instantiation of an object of that type. Event Graph instances (LEGOs) are connected using the SimEventListener pattern described previously, and PropertyChangeListener instances listen to SimEntities using the PropertyChangeListener pattern, also discussed previously.

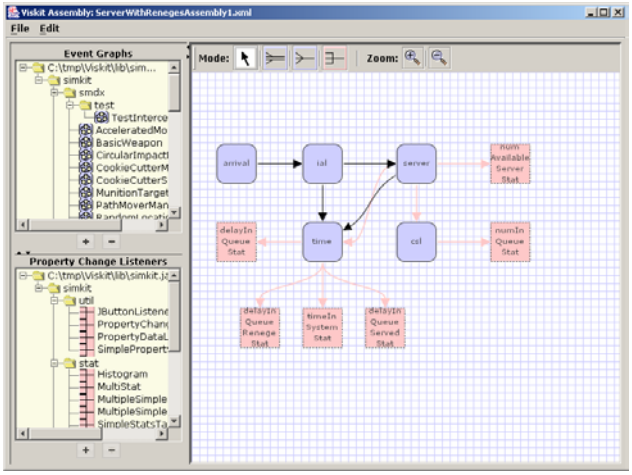

Figure 15. Server with Reneges Assembly

The blue icons in Figure 15 represent the Event Graph component instances (LEGOs) and the pink icons represent PropertyChangeListeners. The dark arrows represent SimEventListening and the pink arrows represent PropertyChangeListening.

An Assembly is normally saved in XML format, like that of Figure 8. As with the Event Graph Editor, the corresponding Java code can be generated, saved, and compiled separately. The Assembly editor can be used to create many different models from the same set of components.

A created Assembly can be run using the controls at the bottom of the window. The user can fill in the stop time for the run and check whether the run is to be in verbose or quiet mode. Verbose mode prints out each event along with the Event List after each event is executed.

6. Conclusions and Ongoing Work

The need for rapid development and implementation of DES models will be present for the foreseeable future. Effective tools are needed to support this. Simkit is a proven platform that supports rapidly implementing DES models in Java. The Analysis Workbench discussed in this paper incorporates tools for even more rapid development while reducing the dependency on programming expertise.

The use of XML as the "native" format has some interesting and useful implications for further work. More and more software utilizes XML for data, and the use of stylesheets allows XML data to be readily transformed from one form to another. XML is a key technology in Web Services, so the description of Event Graph components and Assemblies in XML can help support interoperability with web-based simulation services. In particular, the groundwork exists for interacting with models developed using the Extensible Simulation Modeling Framework (XMSF) ([7])

The Viskit component of the Analysis Workbench provides a user-friendly means of creating Event Graph components and DES simulation models by assembling components. The application is being tested and feedback from users will be incorporated into subsequent versions. One component of the Analysis Workbench not described here consists of a common user interface for launching complete Simkit or Viskit models. This component is currently under development and is being implemented to execute models based on some recent Masters theses written at the Naval Postgraduate School. These models include such diverse areas as maintenance and repair policies for aircraft engines, submarine tactics for negotiating a minefield, and analyzing the dynamic allocation of networked fires and sensors.

Acknowledgments

Rick Goldberg and Michael Bailey expertly implemented schema and graphical user interfaces resulting in the Viskit tool. Curt Blais, Don Brutzman, and Don McGregor gave valuable input and feedback. Comments on a previous draft by the SIW reviewer resulted in substantial improvements. This work was sponsored by the United States Navy OPNAV (N81); this support is gratefully acknowledged.

7. References

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

[2] Schruben, L. 1983. Simulation Modeling with Event Graphs, Communications of the ACM, 26, 957-963.Buss, A. 1996. Modeling with Event Graphs, Proceedings of the 1996 Winter Simulation Conference, J. M. Games, D. J. Morrice, D. T. Brunner, and J. J. Swain, eds.

[3] Buss, A. 2000. Component-Based Simulation Modeling, Proceedings of the 2000 Winter Simulation Conference, J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, eds.

[4] Buss , A. and P. Sanchez. 2002. Building Complex Models with LEGOs (Listener Event Graph Objects). Proceedings of the 2002 Winter Simulation Conference, E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, eds.

[5] Buss, A. 2001. Discrete Event Programming with Simkit. Simulation News Europe, 32/33, November,15-25.

[6] Schruben, L. 1995. Graphical Simulation Modeling and Analysis Using Sigma for Windows, Boyd and Fraser Publishing Company, Danvers, MA.

[7] D. Brutzman, M. Zyda, J. M. Pullen, and K. L. Morse: "Extensible Modeling and Simulation Framework (XMSF): Challenges for Web-Based Modeling and Simulation, Findings and Recommendations Report of the XMSF Technical Challenges Workshop and Strategic Opportunities Symposium," October 2004.

Author Biographies

ARNOLD BUSS is a Research Assistant Professor in the MOVES Institute at the Naval Postgraduate School. He received his MS in Systems Engineering from the University of Arizona and his PhD in Operations Research from Cornell University. His research interests include Discrete Event Simulation and component-based modeling.