Proceedings of the 2007 Winter Simulation Conference
S. G. Henderson, B. Biller, M.-H. Hsieh, J. Shortle, J. D. Tew, and R. R. Barton, eds.

COMPOSABILITY AND COMPONENT-BASED DISCRETE EVENT SIMULATION

Arnold Buss
Curtis Blais

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

ABSTRACT

This work presents a framework and a Graphical User Interface, Viskit, for the creation and analysis of component-based Discrete Event Simulation models. Two primary elements of the tool are discussed. 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 (assembly) mode, components are hooked together to create a model. In analysis mode, the models are exercised and run according to the desired experimental design.

1 INTRODUCTION

Discrete Event Simulation (DES) methodology is a way of modeling a situation in a stylized manner. 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 (Schruben 1983). 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 variables 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 concurrent or 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 directed 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.

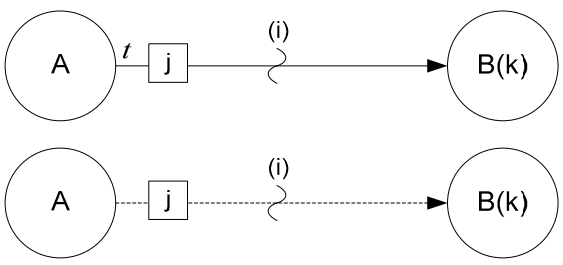

Figure 1: Basic event graph constructs

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 representation 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 COMPOSABILITY AND COMPONENT-BASED DES MODELING

Although Event Graph methodology can be used to build any Discrete Event Simulation model, by itself the methodology does not facilitate composability of such models. A component framework enables such composability, which in turn allows substantially more re-use and scalability of models.

The key to composing DES models is to treat an Event Graph as being the specification of each component. Such components have been called "Listener Event Graph Objects" or "LEGOs" (Buss and Sanchez 2002) because of the prominence of the Listener pattern as a means of loosely connecting them. Each LEGO is an instance of an Event Graph for which the parameters and state variables are encapsulated. Each LEGO is configured to hold its own distinct parameters and is responsible for the events and state transitions that modify its state variables and produce its state trajectories. Since each LEGO component is designed to work in conjunction with other LEGO components, it is not necessarily complete in itself. Like a hardware component, a LEGO may require a number of other LEGOs to trigger its events. The Listener Pattern is how this communication between components occurs (Buss 2002, Buss and Sanchez 2002), and is described next.

2.1 The Listener Pattern

The Listener Pattern provides the primary mechanism by which simulation components communicate in this framework. Two types of components are involved with a listener pattern: the listener component and the Event Source component. The listening component registers interest in another component's events and waits for the other component to execute the event. When the event occurs in the simulation component, the software notifies all registered listeners of the event. Note that the term "event" as used here is different than the simulation events that come off the event list. No matter how many of these events are fired, no simulated time passes. Indeed, the firing of these events can technically be considered to be part of the state transition function for the current simulation Event.

Three entities are involved with every implementation of the Listener pattern: The Event, the Listener, and the Event Source. The same component can serve as a Listener to some components and be an Event Source to other components. The Event that is fired should contain enough information for the Listener to be able to decide what to do without a callback to the Event Source. This no-callback property is a critical one for maximizing the looseness of the coupling between components since such a callback requires the listener to have knowledge of the event source object. Indeed, this feature distinguishes the Listener pattern from the Observer pattern (see Gamma et al. 1995), since the latter typically does require a callback to the event source. For maximum flexibility the Listener should be implemented as an interface consisting of just the single notification method with a signature consisting of a reference to the dispatched event. The Event Source component has three tasks: to register Listener components, to unregister Listener components, and to fire the Event at the proper time. Note that the use of an interface to implement the Listener pattern is critical to its extensibility. Implementing a Listener as a class, whether concrete or abstract, restricts all further Listeners to be subclasses. In fact, there is an Interface design pattern that is appropriate here (Gamma et al. 1995). The Interface pattern is easily implemented using a Java interface, enabling disparate classes without any *is-a* relationship whatsoever to be first-class participants as Listeners.

The power of the Listener pattern stems from the fact that the Event dispatching can be implemented generically, with the Event Source having to know only that the receiving component implements the Listener interface. The interface for a Listener typically consists of a single method with one argument, a reference to the dispatched Event. The event source uses this method to make a callback to each listener when the Event is dispatched. Thus, the interface for the event source consists (at a minimum)

of methods for registering and unregistering Listeners and at least one method to trigger an Event dispatch.

We will now discuss the two Listener patterns that have proved very useful for Discrete Event Simulation Modeling: the SimEventListener and the PropertyChangeListener. The presentation will be oriented towards its implementation in Simkit, since that forms the underpinnings of Viskit, the visual tool for creating such component-based models.

2.2 SimEventListener Pattern

The SimEventListener pattern involves an event that has been executed by the Event List. It consists of the source of the event (the SimEntity that scheduled it) multicasting the same SimEvent to registered SimEventListeners. The callback method from the Event List for a SimEntity is handleSimEvent(SimEvent), which simply invokes the processSimEvent(SimEvent) method defined by SimEventListener. The SimEvent contains data (in the form of a String) about which method is to be invoked and optionally a parameter list (in the form of an array of Objects) to be passed to the method. Java's reflection mechanism is used to find the desired method and to invoke it. The invoked method is determined by prepending "do" to the event name and matching a method of that name with a signature consistent with the parameter list. When processSimEvent() returns, then notifyListeners(SimEvent) is called, thus multicasting the SimEvent to all registered SimEventListeners. The SimEventListener interface defines just the processSimEvent(SimEvent) method, thus making it very easy for components to define different ways to respond to SimEvents. For example, instead of the slower (but flexible) reflection used by SimEntityBase, Simkit's default SimEntity base class, the desired method could be invoked using a switch-type statement based on magic numbers. Another example occurs when a base class that is not a SimEventListener has already been identified. The class has only to declare that it implements SimEventListener and then actually implement the processSimEvent(SimEvent) method. This is typical of the way Java implements polymorphism and is an alternative to multiple inheritance.

A SimEntity can only multicast a SimEvent it has previously scheduled; a heard SimEvent is not dispatched to its listeners. This enables two SimEntities having the same Event to listen to each other without generating an infinite loop. Of course, it is always possible to programmatically create cycles of scheduled events, but each new event must be explicitly scheduled.

2.3 PropertyChangeListener Pattern

The PropertyChangeListener pattern specifically involves components changing a property value and notifying interested listeners about that change. The Java language provides support for this pattern with the PropertyChangeEvent and the PropertyChangeListener interface, part of the "JavaBeans" conventions. A PropertyChangeEvent instance contains the property's name, references to both the old and new values, and a reference to the source of the PropertyChangeEvent to support callbacks.

Simkit adopts the convention of firing PropertyChangeEvents whenever state variables change value. The PropertyChangeListener interface has a single callback method, propertyChanged(PropertyChangeEvent) that is invoked when a property is fired. The PropertyChangeSupport class has methods for registering and unregistering PropertyChangeListeners and for firing PropertyChangeEvents. An object can delegate the management of the PropertyChangeListener pattern to an instance of PropertyChangeSupport.

The PropertyChangeListener pattern is more useful than a SimEventListener when the listening component is primarily interested in the state changes rather than the occurrence of a particular event. The property itself could in fact be present in more than one simulation component; and a PropertyChangeListener could be registered with all components managing a particular property. Furthermore, a component only concerned with the state variable would have to make a callback to the source if it used the SimEventListener pattern to hear the property changes. A PropertyChangeEvent, in contrast, contains all the necessary state information for that variable.

3 VISKIT

Viskit is a graphical front end for creating, editing, and composing DES simulation models using Event Graphs and the LEGO framework. Viskit currently implements all the basic functionality required to create the kind of DES models described in previous sections. This section will provide an overview of some of the basic features of the Viskit tool and its capabilities.

3.1 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 2.

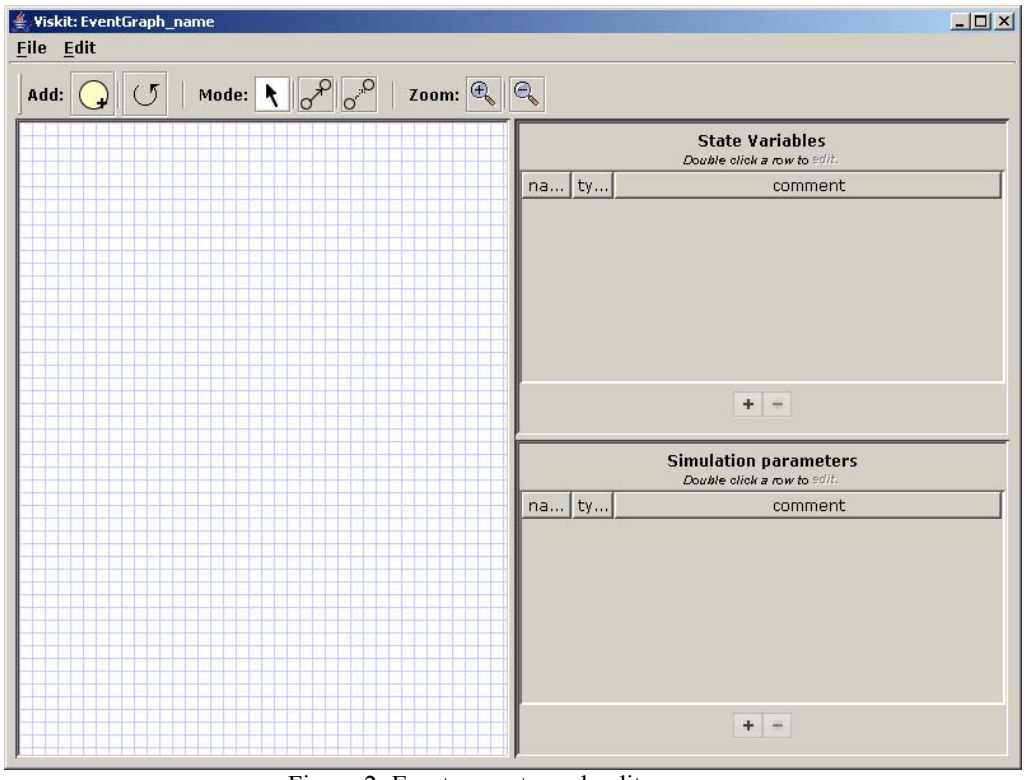

Figure 2: Empty event graph editor screen

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 event 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 3 (following pages) shows a completed Event Graph component.

Figure 4 (following pages) shows the Node inspector which is used to input, display, and edit the data associated with the node. 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 5 (following pages) shows the edge inspector, which is used to input, display, and edit information about the edges. 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 free-form expressions. As with the node inspector, Beanshell is used to verify all expressions entered in free form. 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 saves its components in XML. Simkit Java code can also be generated and saved for separate compilation, as shown in Figure 6 (following pages).

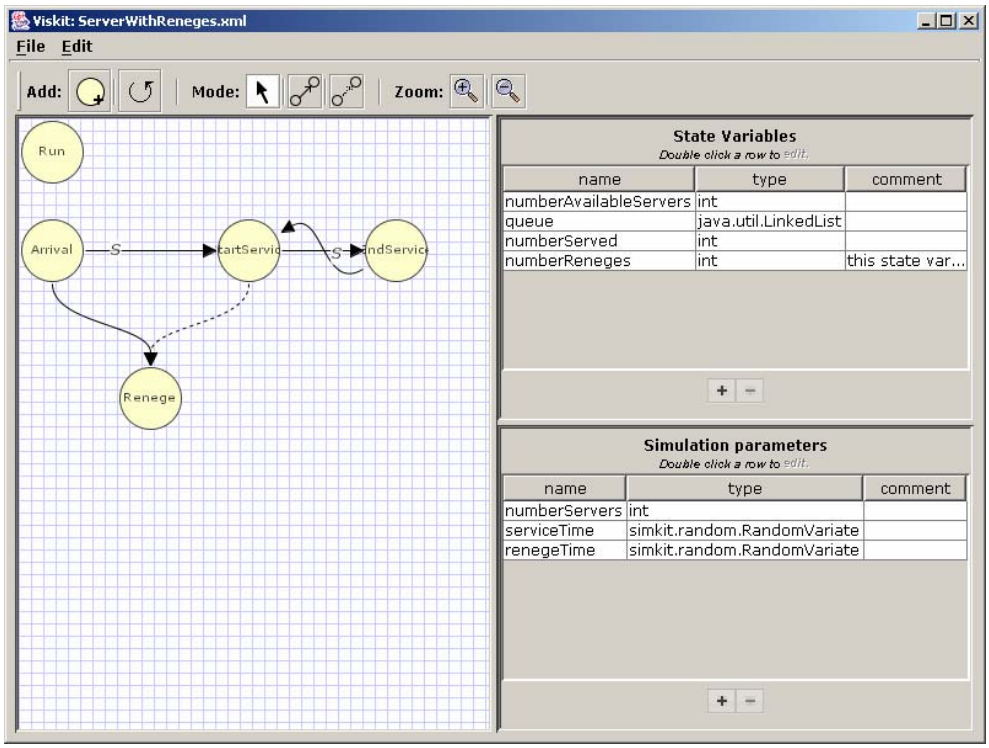

Figure 3: Server with reneges event graph

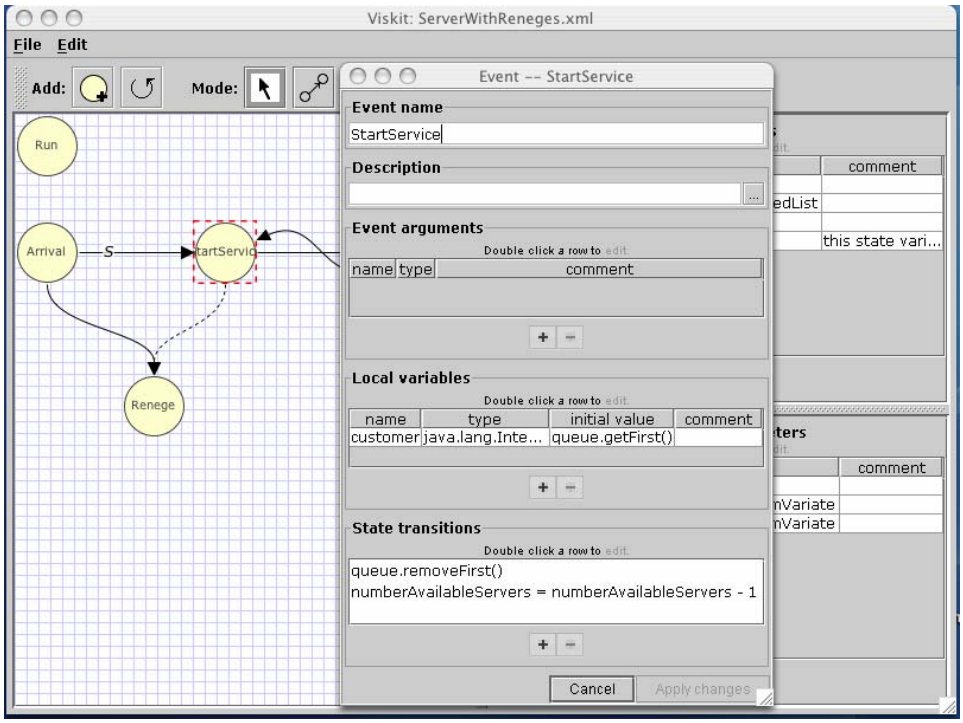

Figure 4: Event node inspector

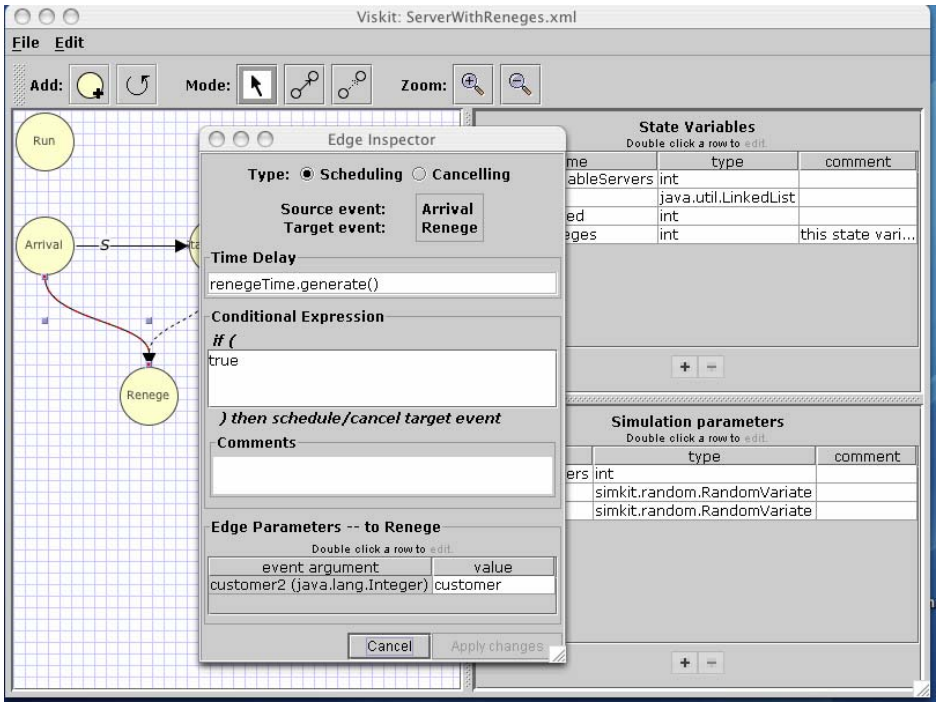

Figure 5: Edge inspector

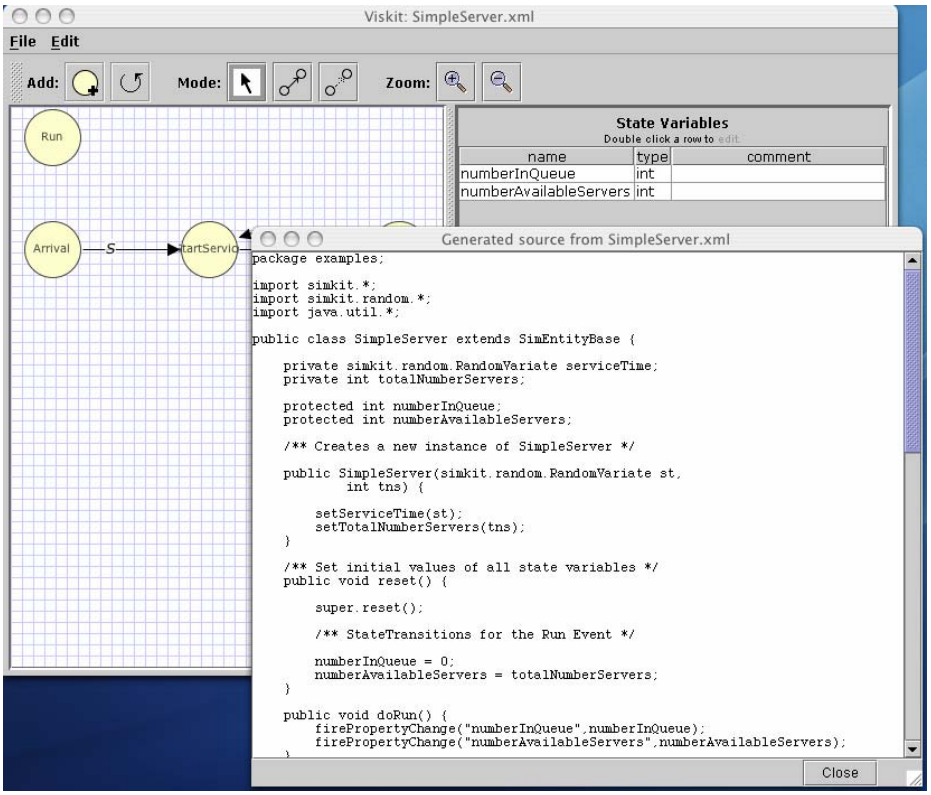

Figure 6: Generated Simkit code

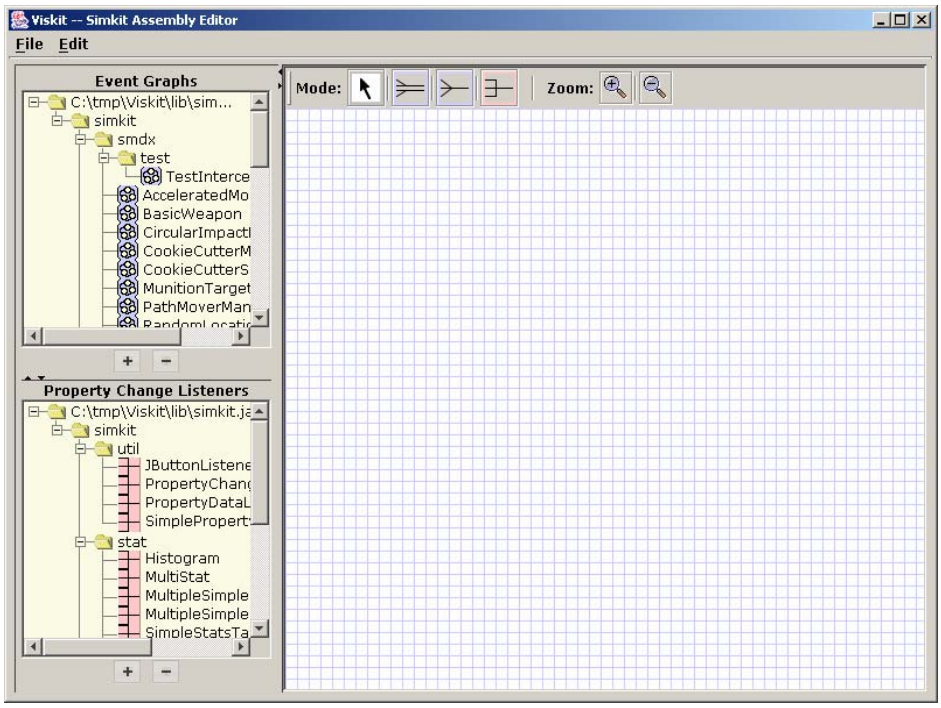

Figure 7: Empty assembly editor window

3.2 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 7.

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. An example of an Assembly is shown in Figure 8 (following page).

The blue icons in Figure 8 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 also saved in XML format. 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.

4 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. Increasingly, software applications utilize XML for data representations and processing, 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.

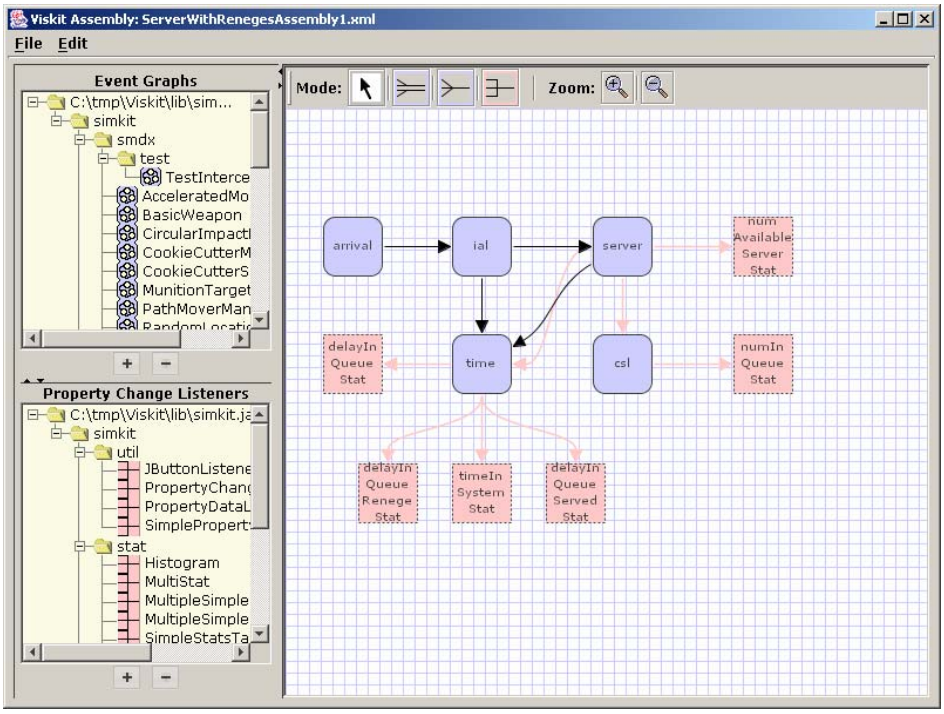

Figure 8: Example assembly

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. Other components of the Analysis Workbench not described here include a user interface for performing design of experiments and for launching complete Simkit or Viskit models. Viskit continues to be developed whil being employed in numerous research programs, Masters theses, and simulation instruction at the Naval Postgraduate School. Models developed using this graphical user interface include such diverse areas as maintenance and repair policies for aircraft engines, submarine tactics for negotiating a minefield, analysis of the dynamic allocation of networked fires and sensors (Buss and Ahner 2006), and perimeter security scenarios for waterside and landside anti-terrorism/force protection (Brutzman et al. 2006).

REFERENCES

Brutzman, D., C. Blais , and T. Norbraten. 2006. Modeling and 3D Visualization for Evaluation of Anti-Terrorism/Force Protection Alternatives Phase II Final Report. Technical Report NPS-MV-06-002. Naval Postgraduate School. Monterey, CA. 31 October.

Buss, A. H. 2001. Discrete Event Programming with Simkit. *Simulation News Europe*. 32/33:15-24.

Buss, A. H. 2002. Component based simulation modeling with Simkit. In *Proceedings of the 2002 Winter Simulation Conference*, ed. E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes.

Buss A. H. and P. J. Sanchez. 2002. Modeling very large scale systems: 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, 732-737. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Buss, A. H. and P. J. Sanchez. 2005. Simple movement and sensing in discrete event simulation. In *Proceedings of the 2005 Winter Simulation Conference*, ed. M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, 992-1000. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Buss, A. H. and D. Ahner. 2006. Dynamic Allocation Of Fires And Sensors (Dafs): A Low-Resolution Simulation For Rapid Modeling. In *Proceedings of the 2006 Winter Simulation Conference*, ed. L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, 1357-1364. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Free Software Foundation Web Site. <http://www.fsf.org>. [accessed June 2006].

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

Schruben, L. 1983. Simulation modeling with event graphs. *Communications of the ACM* 26:957-963.

AUTHOR BIOGRAPHIES

CURTIS BLAIS is a Research Associate at the Naval Postgraduate School (NPS) in the Modeling, Virtual Environments, and Simulation (MOVES) Institute. His principal research areas include agent-based simulation of non-traditional warfare and application of Web-based technologies for improving interoperability of Modeling and Simulation systems and Command and Control systems. Mr. Blais has a B.S. and M.S. in Mathematics from the University of Notre Dame and has advanced to candidacy in the MOVES Ph.D. program at NPS. His e-mail address is <clblais@nps.edu>.

ARNOLD BUSS is a Research Assistant Professor in the MOVES Institute at the Naval Postgraduate School. His research interests include discrete event simulation modeling and hybrid approaches incorporating multiple modeling methodologies His e-mail address is <abuss@nps.edu>.