

## TECHNICAL NOTES

## Discrete Event Programming with Simkit

Arnold Buss

Operations Research Department, Naval Postgraduate School  
Monterey, CA 93943-5000 U.S.A.**Introduction**

This paper is a brief introduction to the use of Simkit, a software package for implementing Discrete Event Simulation (DES) models. Simkit is written in Java and runs on any operating system with Java 2™ installed.

Simkit adopts DES as its fundamental world view and does not directly implement other world views such as process/resource. Although this makes certain simple models slightly more complex, a pure DES world view provides more flexibility and modelling power than a pure process-oriented world view. Event Graph methodology is sufficiently powerful by itself to represent any model that can be captured by the DES framework. In particular, every model that can be represented in the process world view can also be represented in a pure DES world view; the reverse is not true.

The remainder of this paper is organized as follows. First we will discuss Simkit's implementation of the Event List, then the primary templates for constructing simulation components, the SimEntity interface and the SimEntityBase class. Next, we show how Simkit starts and stops simulation execution, followed by a simple example. Following a brief description of the listener patterns used in Simkit, we present more examples. Then we show how Simkit implements two advanced features of Event Graph modeling, cancelling edges and passing parameters to events, with illustrative examples for each. Finally, Simkit's random variate generation framework is briefly discussed.

**Event List Implementation**

All DES frameworks require an implementation of a Future Event List (FEL) to operate.

Simkit implements a FEL in a class called simkit.Schedule that consists entirely of static methods and variables. The Schedule class has a variable representing the FEL using a Java class called java.util.SortedSet, which contains objects of type SimEvent. Each SimEvent object contains data on which event it represents and what time it is scheduled to occur. The SortedSet object uses a Comparator based on a sequence of criteria, the first being the

scheduled time. In cases of ties, the SimEvent object can be given a priority.

Simkit attempts to hide the details of the FEL from the simulation modeller. Instead of directly placing events on the FEL, the programmer invokes the waitDelay() method on an instance of simkit.SimEntityBase, as described in the following section. The execution of the event consists of a callback from Schedule to the SimEntity instance that originally scheduled it.

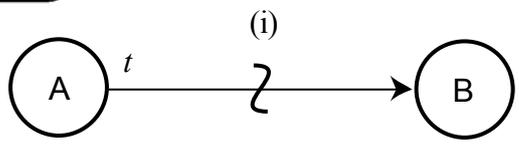
**SimEntity and SimEntityBase**

Simkit provides an abstract class and an interface to help encapsulate the Future Event List activities (scheduling events and processing events).

The SimEntity interface specifies a set of methods that must be implemented by any class designed to interact with the FEL and with other simulation objects. SimEntityBase is an abstract class that implements most of the functionality for interacting with the FEL. Recall that there are just two constructs in Event Graphs: the event (node) and the scheduling edge (Schruben, 1983).

Each event in a Simkit model is implemented as a user-defined "do" methods in a subclass of SimEntityBase. A "do" method is simply a method starting with the string "do." Scheduling edges are executed using a method called "waitDelay()" that had various signatures. The simplest has signature (String, double), where the first argument is the name of the event without the "do" and the second argument is the amount of simulated time between when the event is scheduled and when it occurs, that is, the delay associated with scheduling that event. The boolean edge condition is implemented by wrapping the waitDelay() call in an "if" test.

For example, the basic Event Graph construct (Schruben, 1983; Buss, 2001), is shown in Figure 1.



**Figure 1. Basic Event Graph Construct**

The event graph in Figure 1 is interpreted as follows: “When event A occurs, then if condition (i) is true, event B is scheduled to occur after a delay of *t* simulated time units.” The Simkit code corresponding to Figure 1 is implemented in Simkit in the following code snippet:

```
public void doA() {
    <code to perform state transition for event A>
    if (i) {
        waitDelay("B", t);
    }
}
```

The order of execution in a “do” method is, by convention, identical to that in Event Graphs: first perform state transitions, then schedule events (if any).

Note that the first argument in the `waitDelay()` call above is a String that is the event name, not the method name. Simkit uses Java’s reflection to determine the corresponding method. When `waitDelay()` is invoked, `SimEntityBase` creates a `SimEvent` and adds it to the FEL. Each `SimEvent` contains a reference to the object that created it. When that event “occurs” the Event List invokes a callback method on the object that scheduled it called “`handleSimEvent(SimEvent)`.” Normally, the programmer does not have to deal with this method, however. The `SimEntityBase` class implements the `handleSimEvent()` method to invoke the “do” method indicated by the data in the `SimEvent`.

When the `SimEntityBase` instance receives the event, it attempts to find a matching “do” method for that event based on its name. In the example above, when the event with name “B” is received from the Event List, the scheduling object prepends the string “do” and tries to find a method called “`doB()`”. If such a method is found, it is invoked. If no such method is found, then `SimEntityBase` returns to the FEL algorithm with no error.

Event Graphs have only one “keyword”—the event called Run. The Run event is analogous to the main method in C or Java programs. The Run event is placed on the event list at the start of the simulation run. That way the FEL always starts the run in a non-empty state.

Simkit implements the Run event by adopting the convention that every `SimEntityBase`, upon instantiation, is examined for the presence of a `doRun()` method. If the method is found, then a Run event is scheduled to occur at time 0.0. If no Run event is found, then there is no error. While Simkit encour-

ages the use of the Run event to initiate the simulation, it is not required. However, if the Run event is not used, the modeller must put events on the Event List “by hand” prior to the onset of the simulation run.

In Simkit, the Run event is only used to schedule the first events. Simkit requires an additional initialization method, `reset()`, that is responsible for initializing (or reinitializing) the state variables to their initial values. A call to `Schedule.reset()` just prior to starting the simulation run invokes `reset()` on every subclass of `SimEntityBase` that has been instantiated. Thus, each simulation object only has to be responsible for initializing its own state variables. Initializing state variables is separate from initial scheduling so that the first events that occur can reasonably assume that all objects have been set to their legitimate initial state.

### Starting and Stopping

The simulation run is controlled by the `Schedule` class, which also houses the FEL. `Schedule` initiates the run when there is a call to its `startSimulation()` method, which executes FEL.

The simulation continues executing until the FEL is empty. There are essentially four ways in Simkit by which this can occur: (1) the FEL empties naturally of its own accord; (2) there is an explicit call to `Schedule.stopAtTime(double)` before `Schedule.startSimulation()` is invoked; (3) there is a call to `Schedule.stopOnEvent(String, Class[], int)` before `Schedule.startSimulation()` is invoked; and (4) there is a call to `Schedule.stopSimulation()` anywhere in the program

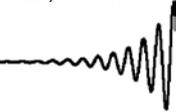
If the FEL empties of its own accord, then the simulation ends. The `Schedule.startSimulation()` method returns to where it was invoked. Typically, report methods are then invoked, depending on how the simulation run was configured. The value of simulated time (`simTime`) stays the same as last executing event, but can be set back to time 0.0 with a call to `Schedule.reset()`;

A call to `Schedule.stopAtTime(double)` schedules a Stop event, that just invokes `Schedule.stopSimulation()`. For example, `Schedule.stopAtTime(10.0)` stops the simulation at time 10.0 and leaves time at 10.0 when the run is over.

A call to `Schedule.stopOnEvent(String, Class[], int)` or `Schedule.stopOnEvent(String, int)` causes the simulation run to end after there have been a certain number of events of the specified name occurring. For example, invoking `Schedule.stopOnEvent("Arrival", 10)` will cause the simulation to end after the 10th Arrival event has occurred. Similarly, `Schedule.stopOnEvent("Arrival", new`

TECHNICAL NOTES

Issue 32



Class[] { Job.class }, 10) stops the simulation after the 10<sup>th</sup> Arrival(Job) event has occurred.

### Example: The Arrival Process

The Arrival Process is the simplest nontrivial Event Graph, essentially the “Hello World” of Event Graphs.

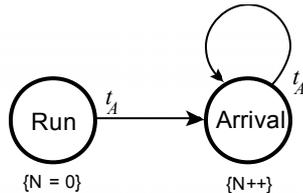


Figure 2. Arrival Process Event Graph

The initialization of a Simkit simulation run breaks the Run event into two parts: the initialization of state variable values and the scheduling of initial events on the Event List. The initial state values are set in a method called reset(), while the scheduling of initial events is done in the doRun() method. Recall that the Run event is placed on the Event List at the start of the simulation run.

The reset() method is typically not directly invoked, but rather is invoked by a call to Schedule.reset(). This has the effect of invoking reset() on all SimEntityBase instances that have been created. Thus, Schedule.reset() is like a big “reset” button on the simulation model. The modeler does not have to keep track of every SimEntityBase that has been created.

The Simkit implementation of the Arrival Process is shown in Figure 3 (the complete code for all examples is in the Appendix).

```
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 3. Simkit Code Snippet for Arrival Process

The Arrival Process class maintains a single state variable, numberArrivals, that counts the cumulative number of arrivals since time 0.0. The state variable is incremented by one upon the occurrence of each Arrival event. The reset() method therefore initializes numberArrivals to zero and the method doArrival() increments the value and schedules the next Arrival event, after a random delay. The doRun() method simply schedules the first Arrival event.

When the state variable numberArrivals is incremented, a PropertyChangeEvent is fired. That is, an instance of a PropertyChangeEvent is dispatched to all objects that have registered as PropertyChangeListener to the ArrivalProcess instance. This feature is useful for separating the code to model the state of the system from the code to compute statistics, graphs, or any other output-related task.

Running a Simkit model thus consists of the following tasks:

1. Instantiate desired objects
2. Register SimEventListener objects
3. Register PropertyChangeListener objects
4. Set stopping time or stopping criteria
5. Set the mode of the run (verbose/quiet, single-step/continuously running)
6. Reset all SimEntityBase instances
7. Start the simulation

Depending on the simplicity or the complexity of the model, some of these steps may be omitted. For the Arrival Process model, we will implement these tasks in a pure execution class—that is, a class consisting of only a main() method.

```
//1. Instantiate objects
    ArrivalProcess arrival = new ArrivalProcess(...);
//4. Stopping criterion: at time 10.0
    Schedule.stopAtTime(10.0);
//5. Verbose mode on
    Schedule.setVerbose(true);
//6. Reset all SimEntityBase instances
    Schedule.reset();
//7. Start simulation
    Schedule.startSimulation();
```

Figure 4. Execution Code for Arrival Process Test

Note the use of the RandomVariate instance variable in the code in Figure 3. The code in Figure 3 does not show how an instance of interarrival is obtained. We will discuss Simkit’s approach to random variate generation below. For now, it is sufficient to note that an instance of RandomVariate has a method called generate(), and each invocation of the generate() method returns a new random variate having a particular distribution.

### Listener Patterns

Simkit uses two “Listener” patterns to implement its component interoperability. The SimEventListener pattern is used to connect simulation components (instances of SimEntityBase) in a loosely coupled manner. As described above, SimEvents are always invoked by a callback from the FEL to the scheduling object that ultimately invokes the corresponding “do”

method. The `SimEvent` is then dispatched to every `SimEventListener` that has explicitly been registered interest in that object's `SimEvents`.

A related pattern, the `PropertyChangeListener` pattern, comes into play whenever a state variable changes value. In that case, a `PropertyChangeEvent` is dispatched to registered `PropertyChangeListener` objects. The purpose of `PropertyChangeEvent`s is to support generic observation of the simulation state trajectories, as well as any function thereof.

### SimEvent Listener Pattern

The mechanism by which two simulation components are linked is the `SimEventListener` pattern. Every `SimEntity` implements the `SimEventListener` interface, that defines a callback method. An instance of a `SimEventListener` registers interest in hearing a `SimEntity`'s simulation events with the `addSimEventListener(SimEventListener)` method. Whenever a `SimEvent` occurs for the `SimEntity` instance, notification is dispatched to all registered `SimEventListeners` via the callback method `processSimEvent(SimEvent)`.

The behaviour of a `SimEventListener` as implemented in the `processSimEvent(SimEvent)` method can be completely customized to suit the simulation modeller's needs. Most of the time, the modeller will be content with the default behaviour as implemented in the (abstract) `SimEntityBase` class. That behaviour is that whenever a `SimEvent` is heard, the object attempts to find a matching "do" method. If one is found, then it is invoked. If none is found, then nothing happens.

The `SimEventListener` pattern is useful in implementing component-based simulation models (Buss, 2000). For our introductory purposes in this paper, we will not use the `SimEventListener` pattern.

### Property Change Listener Pattern

One capability provided by Java is the ability to fire `PropertyChangeEvent`s whenever certain instance variables change value. This capability is provided by classes in the `java.beans` package, part of the standard Java 2 environment. The `java.beans` package contains a class, `PropertyChangeEvent` that is dispatched to objects interested in the property, and an interface, `PropertyChangeListener`, that provides a common callback method from the source of the `PropertyChangeEvent`.

The `java.beans` package also has a helper class, `PropertyChangeSupport`, that can register and unregister `PropertyChangeListeners` and can act as a proxy for firing the `PropertyChangeEvent`s. `PropertyChangeEvent`s are different than `SimEvents` and do not directly interact with the FEL.

`SimEntityBase` maintains an instance of `PropertyChangeSupport` and provides a method `firePropertyChange()`, with various signatures, to dispatch `PropertyChangeEvent`s to its registered `PropertyChangeListener`s. The convention adopted by Simkit models is that every state change is accompanied by a corresponding firing of a `PropertyChangeEvent`. For example, in the Arrival Process above, instead of simply incrementing the `numberArrivals` state variable, the following code is typically used:

```
firePropertyChange("numberarrivals",
    numberArrivals, ++numberArrivals);
```

This is using the `firePropertyChange(String, int, int)` version. The `String` argument is the name of the property, the first `int` parameter is the "old value" of the property – the value before the state change was made, and the second `int` parameter is the "new value" of the property – the value after the property was changed. Notice that the variable as the increment operator as a prefix rather than a postfix, since the value of the expression is the incremented value in this case, which is what is desired.

If the Simkit program adopts the convention of firing a `PropertyChangeEvent` for every state change, then an effective decoupling occurs between the model and the observation of the model (graphing results, estimating measures of performance, etc.). The model itself does not need to estimate any statistics at all. Instead, separate `PropertyChangeListener` objects can be created that perform estimation, analysis, or plotting results.

A second type of `PropertyChangeListener` event is supported by Simkit, the `IndexedPropertyChangeEvent`. The `IndexedPropertyChangeEvent` is defined in Simkit, since Java beans do not support indexed `PropertyChangeEvent`s. The `IndexedPropertyChangeEvent` is useful whenever the state changing is indexed, as in an array. The index of the property that had changed is included with the `IndexedPropertyChangeEvent`. Since `IndexedPropertyChangeEvent` subclasses Java's `PropertyChangeEvent`, any `PropertyChangeListener` is able to "hear" it. An example of its use is in the model of a transfer line discussed in the following section.

## More Examples

### Multiple Server Queue

An Event Graph for the multiple server queue defines two state variables: `Q`=the number of customers in the queue and `S`=the number of available servers. The Event Graph for the Multiple Server Queue is shown in

Figure 5. Event Graph for Multiple Server Queue (see Buss, 2001).



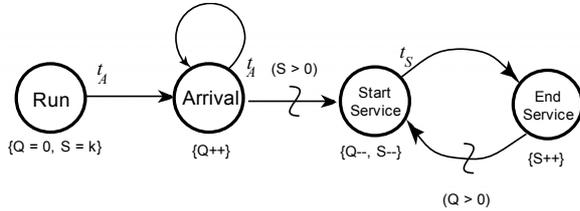


Figure 5. Event Graph for Multiple Server Queue

The Multiple Server Queue is implemented in Simkit by creating a MultipleServerQueue class that defines “do” methods corresponding to the events in

Figure 5. Event Graph for Multiple Server Queue: doRun(), doArrival(), doStartService(), and EndService(). The Simkit code for these methods are shown in Figure 6.

```
public class MultipleServerQueue
    extends SimEntityBase {
    private int totalNumberServers;
    private RandomVariate interArrivalTime;
    private RandomVariate serviceTime;
    protected int numberArrivals;
    protected int numberInQueue;
    protected int numberAvailableServers;
    protected int numberServed;

    public MultipleServerQueue(int numberServers,
        RandomVariate iat, RandomVariate st) {
        totalNumberServers = numberServers;
        this.setInterArrivalTime(iat);
        this.setServiceTime(st);
    }

    public void reset() {
        super.reset();
        numberArrivals = 0;
        numberInQueue = 0;
        numberAvailableServers =
            totalNumberServers;
        numberServed = 0;
    }

    public void doRun() {
        firePropertyChange("numberInQueue",
            numberInQueue);
        firePropertyChange(
            "numberAvailableServers",
            numberAvailableServers);
        waitDelay("Arrival",
            interArrivalTime.generate());
    }

    public void doArrival() {
        firePropertyChange("numberInQueue",
            numberInQueue, ++numberInQueue);
        waitDelay("Arrival",
            interArrivalTime.generate());
        if (numberAvailableServers > 0) {
            waitDelay("StartService", 0.0);
        }
    }

    public void doStartService() {
        firePropertyChange(
            "numberAvailableServers",
            numberAvailableServers,
            --numberAvailableServers);
        firePropertyChange("numberInQueue",
            numberInQueue, --numberInQueue);
        waitDelay("EndService",
            serviceTime.generate());
    }
}
```

```
public void doEndService() {
    firePropertyChange(
        "numberAvailableServers",
        numberAvailableServers,
        ++numberAvailableServers);
    firePropertyChange("numberServed",
        ++numberServed);
    if (numberInQueue > 0) {
        waitDelay("StartService", 0.0);
    }
}
```

Figure 6. Code for MultipleServerQueue Class

The code in Figure 6 shows instance variables that correspond to the parameters and the state variables of the Event Graph model. By convention, parameters are defined to be private whereas state variables are defined with protected access. Thus, subclasses can change state variables but the data are still encapsulated. Parameters typically have both “setter” and “getter” methods, whereas state variables only have “getter” methods. For brevity, these methods have not been shown in Figure 6.

Edge Boolean conditions are implemented by wrapping the waitDelay() call inside an “if” block, with the Boolean condition on the “if” corresponding to the Boolean edge condition. This is illustrated in the Arrival event scheduling of the StartService event and the EndService event scheduling the StartService event.

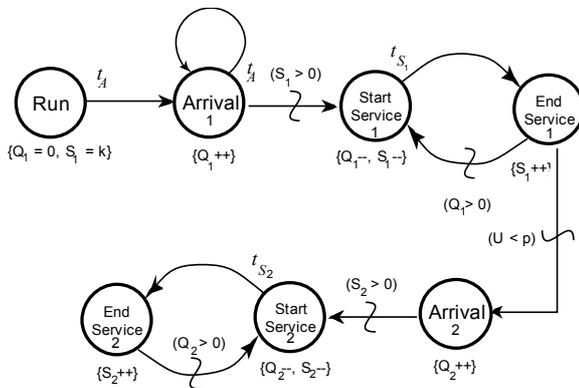


Figure 7. Tandem Queue Event Graph

Tandem Queue Model

A tandem queue model has two servers in series. All entering customers start service at the first server. Modelled as a multiple server queue. Customers completing service at the first server require service at the second server with probability p or leave the system with probability 1-p. The Event Graph for this system is shown in Figure 7 (see Buss, 2001).

Implementing this model in Simkit is a straightforward extension of the multiple server queue discussed

above. The state variables are now Q1 and Q2, the number in queue for the first and second station, respectively, and S1 and S2, the number of available servers at the first and second station.

The methods are likewise an obvious modification of those for the multiple server queue. The one change is in the EndService1 event, which adds the scheduling of the Arrival2 event with probability p. This functionality is shown in Figure 8.

```
protected double probToSecondServer;
protected RandomNumber rng;
. . .
public void doEndService1() {
    firePropertyChange("numberAvailableServers1",
        numberAvailableServers1,
        ++numberAvailableServers1);
    if (numberInQueue1 > 0) {
        waitDelay("StartService1", 0.0);
    }
    if (rng.draw() < probToSecondServer) {
        waitDelay("Arrival2", 0.0);
    }
}
```

Figure 8. Code Snippet for TandemQueue Class

### Modeling Cancelling Edges

Cancelling edges are used in Event Graphs to remove previously scheduled events from the event list. Cancelling edges are implemented in Simkit with the interrupt() method. By convention, cancelling edges for an event are executed after state transitions but before scheduling edges.

The interrupt() method in Simkit behaves slightly differently than in "pure" Event Graphs because of the object-oriented nature of Simkit. The interrupt() method applies to the instance on which it is invoked, rather than globally as in the Event Graph world view. This gives the simulation modeller finer-grained control over cancelling events.

The signature of interrupt is (String, Class[]), where the String argument is the name of the event to be cancelled and the Class[] array represents the arguments on the event – that is, the signature of the "do" method corresponding to the cancelled event. The second argument may be omitted if a zero-parameter event is being cancelled.

### Server with Failures

A model to illustrate cancelling edges is the server with failures (Buss, 2001). Here a single server operates continuously while processing jobs as they arrive. The server fails after a certain (random) time of operation (regardless of the time spent processing jobs) after which it immediately begins repair. After a (random) repair time, the server is available to process jobs again. It is assumed that a job in process when a failure occurs goes back to the queue and is issued a

new service time when the server becomes available again. The Event Graph for this model is shown in Figure 9 (Note that the Event Graph in Buss(2001) for this model has an error in its state transition functions).

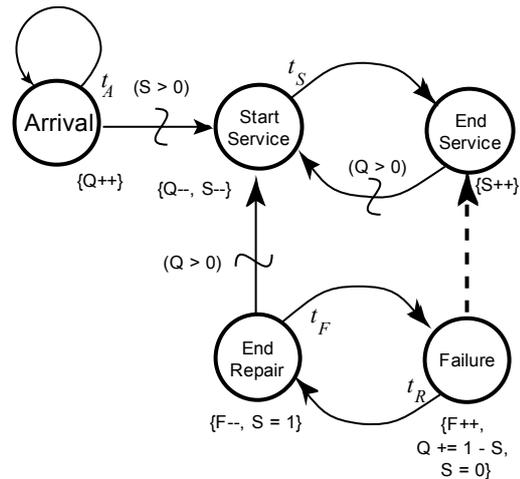


Figure 9. Server with Failures Event Graph

This model can be implemented in Simkit by subclassing the MultipleServerQueue class described above. The code for the Simkit program for the Server with Failures model is shown in Figure 10 (the interrupt call that implements the cancelling edge in Figure 9 is shown in bold for clarity). As before, the constructor and accessor methods are omitted for brevity.

```
public void doRun() {
    super.doRun();
    waitDelay("Failure",
        timeToFailure.generate(), 1.0);
}
public void doFailure() {
    int temp = this.getNumberInQueue();
    numberInQueue +=
        1 - numberAvailableServers;
    firePropertyChange("numberInQueue", temp,
        numberInQueue);
    temp = getNumberAvailableServers();
    numberAvailableServers = 0;
    firePropertyChange(
        "numberAvailableServers", temp,
        numberAvailableServers);
    failed = !failed;
    firePropertyChange("failed",
        new Boolean(!failed),
        new Boolean(failed));
    interrupt("EndService");
    waitDelay("EndRepair",
        repairTime.generate());
}
public void doEndRepair() {
    failed = !failed;
    firePropertyChange("failed",
        new Boolean(!failed),
```



```

        new Boolean(failed));
    numberAvailableServers = 1;
    firePropertyChange(
        "numberAvailableServers", 0, 1);
    if (numberInQueue > 0) {
        waitDelay("StartService", 0.0);
    }
    waitDelay("Failure",
        timeToFailure.generate(), 1.0);
}

```

**Figure 10. Simkit Code for Server with Failures**

In Figure 10, only the next pending EndService event that has been scheduled by that instance of ServerWithFailures will be removed from the event list. If there is no such pending event then nothing happens.

In the state transition for the Failure event (doFailure() method) firing the PropertyChangeEvents for the state variables is slightly more lengthy than in previous models. Since the state transitions cannot be done “in place” with the increment or decrement operators, the old value of the state variable is saved in a temporary variable and passed as the second argument in the firePropertyChange() method.

The waitDelay() statement to schedule the Failure event has a third argument that is the priority of the scheduled event. The default priority is 0.0, so setting the priority of the Failure event to 1.0 ensures that it will occur before any StartService or Arrival events scheduled to occur at the same time.

Note that subclassing MultipleServerQueue was made possible by the fact that the state variables in MultipleServerQueue were declared to have protected access rather than private.

**Passing Parameters on Edges**

An important feature of Event Graphs is the ability to pass parameters on scheduling edges. This enables information about the simulation’s state at a particular simulation time to be transmitted to a future event in a kind of “time capsule.” Parameters on edges are represented in Event Graphs by putting them in a box on the edge, as shown in Figure 11. The corresponding scheduling edge must have an argument that matches the parameters, and vice versa. Cancelling edges can “pass” parameters too, but the meaning is slightly different. When a cancelling edge has a parameter, then the next event that matches both the name *and* the value of the parameter is cancelled (that is, simply removed from the FEL).

Simkit passes parameters using a variant of waitDelay() that adds a third argument of type Object[]. This array of objects should have the values to be passed to the scheduled event so that the signature of the corresponding “do” method is matched. All primi-

tive arguments are wrapped in Java’s Object equivalents. That is, a double argument is passed as a Double object, an int argument is passed as an Integer object, etc. The Simkit code for the scheduling edge in Figure 11 is as follows (assuming that j and k are both primitive integers):

```

public void doA() {
    <state changes for event A>
    if (i) {
        waitDelay("B", t,
            new Object[] {new Integer(j)});
    }
}
public void doB(int k) {...}

```

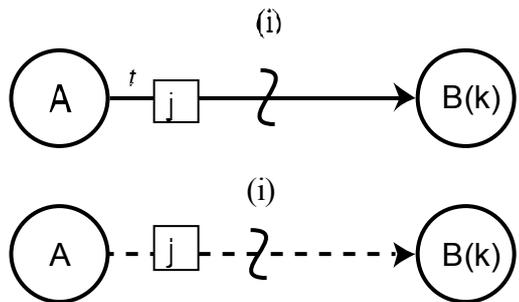
Similarly, the Simkit code for the cancelling edge in Figure 11 is as follows:

```

public void doA() {
    <state change for event A>
    if (i) {
        interrupt("B",
            new Object[] {new Integer(j)});
    }
}
public void doB(int k) {...}

```

Note the syntactic difference between j and k here in both the code and in the Event Graph. The value passed on the scheduling edge, j, is an expression, whereas k on the event B is a format parameter. Note also that the expression j must be computable at the event A. That is, j must be a function of state variables, model parameters, and parameters that may have been passed to A. Thus, event B can use k in



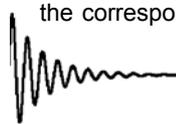
any expression it defines.

**Figure 11. Parameters on Edges**

The signature for the doB() method could be Integer (the object wrapper for the primitive int) instead of the int with the same effect. However, care must be taken to not overload “do” methods with the primitive and the corresponding object wrapper types.

**The Transfer Line Model**

The Event Graph model for a transfer line is as follows (Buss, 2001). Arriving customers are processed by n workstations in a series, each consisting of a



multiple-server queue. Upon completion of service at each workstation, a customer proceeds to the next workstations and departs the system when service at the last workstation is complete. The Event Graph is shown in Figure 12.

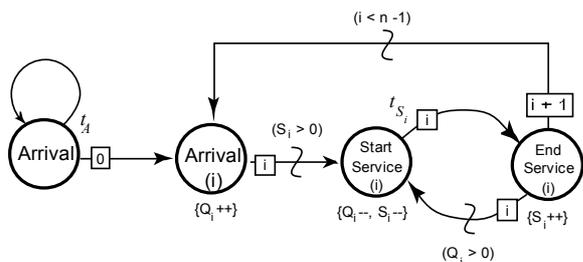


Figure 12. Transfer Line Event Graph

The Simkit implementation of the transfer line in Figure 12 is essentially identical to the multiple server queue model in Figure 6, except that the scalar state variables in the multiple server queue model are now replaced by arrays. Also, since the events have parameters, the corresponding “do” methods have signatures that match. These arguments correspond to the index of the workstation at which the event “occurs.” For example, doArrival(int i) means that a job arrives to workstation i.

Note that the Event Graph in Figure 12 has two Arrival events, one with no parameters and one with a parameter. These are implemented in the Simkit code by overloading the doArrival method, as shown in Figure 13.

```
public void doRun() {
    for (int i = 0; i < numberInQueue.length; i++) {
        fireIndexedPropertyChange(i, "numberInQueue", numberInQueue[i]);
    }
    for (int i = 0; i < numberAvailableServers.length; i++) {
        fireIndexedPropertyChange(i, "numberAvailableServers", numberAvailableServers[i]);
    }
    waitDelay("Arrival", interArrivalTime.generate());
}

public void doArrival() {
    firePropertyChange("numberArrivals", ++numberArrivals);
    waitDelay("Arrival", interArrivalTime.generate());
    waitDelay("Arrival", 0.0, new Integer(0) );
}

public void doArrival(int i) {
    fireIndexedPropertyChange(i, "numberInQueue", new Integer(numberInQueue[i]), new Integer(++numberInQueue[i]));
    if (numberAvailableServers[i] > 0) {
        waitDelay("StartService", 0.0, new Integer(i) );
    }
}
```

```
}
public void doStartService(int i) {
    fireIndexedPropertyChange(i, "numberInQueue", new Integer(numberInQueue[i]), new Integer(--numberInQueue[i]));
    fireIndexedPropertyChange(i, "numberAvailableServers", new Integer(numberAvailableServers[i]), new Integer(--numberAvailableServers[i]));
    waitDelay("EndService", serviceTime[i].generate(), new Integer(i) );
}

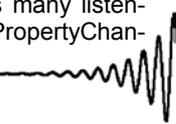
public void doEndService(int i) {
    fireIndexedPropertyChange(i, "numberAvailableServers", new Integer(numberAvailableServers[i]), new Integer(++numberAvailableServers[i]));
    if (numberInQueue[i] > 0) {
        waitDelay("StartService", 0.0, new Integer(i) );
    }
    if (i < getNumberWorkstations() - 1) {
        waitDelay("Arrival", 0.0, new Integer(i + 1) );
    }
}
}
```

Figure 13. Code for Transfer Line Model

### Collecting Statistics

Simkit uses the PropertyChangeListener pattern for collecting statistics from a simulation model. This pattern provides a great deal of flexibility for what gets collected, how it is collected, and which measures of performance are estimated. This approach also enables a clean separation between implementing the dynamics of the model and gathering data. The model can thus be created without any concern over which statistics are to be estimated, and the model classes themselves will not contain any code involved with statistics. All a model class must do is make sure it fires a PropertyChangeEvent whenever a state variable changes its value. This facility is provided by the SimEntityBase class by means of the firePropertyChange() method. The first argument in firePropertyChange() is the name of the property being fired, a String, and the second is the new value of the property, which can be a primitive, or an Object.

Data gathering is performed by classes that implement the PropertyChangeListener interface. This interface is part of the standard Java library in the package called java.beans. The PropertyChangeListener interface specifies a single callback method, propertyChange(PropertyChangeEvent). The PropertyChangeEvent object passed to the listener contains two key pieces of data: the name of the property that has changed in the source object (a String) and the new value of the property (an Object). What the listener does with this information is, of course, completely dependent on the implementation of the PropertyChangeListener class. Note that as many listeners can be registered with a source of PropertyChan-



Time	Event	Property	Value
[0.000	Run	numberInQueue	null -> 0
[0.000	Run	numberAvailableServers	null -> 2
[0.655	Arrival	numberInQueue	0 -> 1
[0.655	StartService	numberAvailableServers	2 -> 1
[0.655	StartService	numberInQueue	1 -> 0
[1.106	Arrival	numberInQueue	0 -> 1
[1.106	StartService	numberAvailableServers	1 -> 0
[1.106	StartService	numberInQueue	1 -> 0

Figure 20. Screen Capture of Property Change Frame

geEvents as desired (up to the limits of the Java virtual machine, of course). It is possible for a PropertyChangeListener to register for just a single property.

Two simple classes for data collection that are used in Simkit models are SimpleStatsTimeVarying and SimpleStatsTally. Instances of these classes compute summary statistics for a single property of the time-varying or tally type, respectively. The instance is registered as a PropertyChangeListener with an object that fires a PropertyChangeEvent with the given name.

When a SimpleStats object “hears” a PropertyChangeEvent, it checks to see whether the property name is identical to the one it is listening for. If so, then it updates its counters with the new property value it retrieves from the PropertyChangeEvent. This value must be an instance of Java’s Number class.

For example, the MultipleServerQueue fires PropertyChangeEvents for properties named “numberInQueue” and “numberAvailableServers” (see Figure 6). Since these are time-varying state variables, to collect statistics on them two instances of SimpleStatsTimeVarying are created. One is configured to listen for a property called “numberInQueue” and the second for a property called “numberAvailableServers.” Whenever a firePropertyChange() method is invoked in a MultipleServerQueue instance, a PropertyChangeEvent is dispatched to all registered listeners, in this case each of the two SimpleStatsTimeVarying objects. The code to instantiate the two SimpleStatsTimeVarying instances, and register them as PropertyChangeListeners to an instance of MultipleServerQueue is shown in Figure 14. This code can be written in either a main() method or in a simulation “executive” class.

```
MultipleServerQueue msg =
    new MultipleServerQueue(...);
SimpleStatsTimeVarying niqStat =
    new SimpleStatsTimeVarying("numberInQueue");
SimpleStatsTimeVarying nasStat =
    new SimpleStatsTimeVarying(
        "numberAvailableServers");
msg.addPropertyChangeListener(niqStat);
```

```
msg.addPropertyChangeListener(nasStat);
```

Figure 14. Code to Instantiate and Register Listeners

In this example two PropertyChangeListeners are listening to one object firing the PropertyChangeEvents. It is also possible for a state variable to change in more than one object. In that case, one listener can simply be registered with all the objects responsible for that property. When the simulation ends (or during the simulation run, if needed) basic sample statistics can be obtained from the SimpleStats objects using the appropriate “getter” method. For example, getMean() returns the sample mean, getVariance() the sample variance, etc. Running the multiple server queue with the two SimpleStats listeners as in Figure 14 can result in output like that shown in Figure 15.

```
Multiple Server Queue with 2 servers

Service time distribution is Gamma (2.5, 1.2)

Arrival Process with Exponential (1.7) interarrival times

Simulation ended at time          1000.0000

There have been 614 customers arrive to the system

There have been 607 customers served

Average Number in Queue           4.0739

Average Utilization                0.9166
```

Figure 15. Example of Output from Multiple Server Queue Model

In the output in Figure 15, only the estimated averages were produced by the SimpleStatsTimeVarying object; the rest was simply custom-written report template for this model.

The loose coupling between the model’s state and the gathering of data enables a considerable degree of flexibility in what can be done with the model without editing or recompiling the simulation classes. For

example, suppose a plot of the trajectory for a given state variable is desired. A `PropertyChangeListener` class can be written that listens for the given state variable property and plots the next observation when the `PropertyChangeEvent` is heard. No invasive editing of the original class's source code is required to add substantially different features to the overall model.

To illustrate, suppose a more detailed trace of certain state variable is required for debugging purposes. Simple class called `PropertyChangeFrame` is a `PropertyChangeListener` that can be registered to listen to the `MultipleServerQueue` (in addition to the `SimpleStatsTimeVarying` instances already registered). The `PropertyChangeFrame` simply writes the event and the state change whenever it "hears" the `PropertyChangeEvent`. A screen capture of this is shown in Figure 20.

## Generating Random Variates

Simkit's design permits much flexibility for generating random variates used in the simulation models. The underlying design goal was to enable the modeller to change any random variate in a model to any desired probability distribution without having to recompile the model. This was to extend to classes generating random variates implemented *after* the compilation of the original model.

Simkit uses a combination of a `RandomVariate` interface and an abstract factory that is called to produce instances of the desired implementation using only "generic" data—that is, Strings, Objects, and numbers. A full discussion of Simkit's design for generating random will be presented elsewhere.

## Obtaining Simkit

The latest version of Simkit, including the source code, can be downloaded from the World Wide Web at <http://diana.gl.nps.navy.mil/Simkit/>. The source code for the examples presented in this paper may be obtained from the above URL as well. Simkit is copyright under the GNU Public License (GPL), which permits its use without any licensing fee.

## Conclusions

This article has presented a basic introduction to Simkit, an object-oriented, component-based platform that can be used to create discrete event simulation models using Event Graph methodology. Since Event Graphs can be used to represent any discrete event system, there are no theoretical limitations to the DES models that can be implemented in Simkit. The loose coupling in Simkit's component architecture facilitate a significant degree of reusability of simulation components. The Listener patterns used to implement the

loose coupling give the modeller a great degree of flexibility in adding new features to existing models without invasive changes to the source code. Simulation models using Simkit can be built and executed on any Java 2-enabled platform.

## References

- [1] 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.
- [2] 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.
- [3] Buss, A. 2001. *Basic Discrete Event Modeling*. *Simulation News Europe*.
- [4] Law, A. and D. Kelton. 2000. *Simulation Modeling and Analysis, Third Edition*, McGraw-Hill, Boston, MA.
- [5] Schruben, L. 1983. *Simulation Modeling with Event Graphs*, *Communications of the ACM*, 26, 957-963.
- [6] Schruben, L. 1995. *Graphical Simulation Modeling and Analysis Using Sigma for Windows*, Boyd and Fraser Publishing Company, Danvers, MA.
- [7] Schruben, L and E. Yücesan. 1993. *Modeling Paradigms for Discrete Event Simulation*, *Operations Research Letters*, 13, 265-275.
- [8] Schruben, L. and E. Yücesan. 1994. *Transforming Petri Nets Into Event Graph Models*. Proceedings of the 1994 Winter Simulation Conference, J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila, eds.

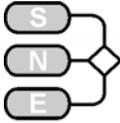
## Appendix

The complete source code for the `ArrivalProcess` class discussed above is shown below.

```
package examples;
import simkit.SimEntityBase;
import simkit.random.RandomNumber;
import simkit.random.RandomVariate;
import simkit.random.RandomNumberFactory;
import simkit.random.RandomVariateFactory;
import simkit.random.CongruentialSeeds;
public class ArrivalProcess extends SimEntityBase {
    private int numberArrivals;
    private RandomVariate interArrivalTime;

    public ArrivalProcess(String distribution,
        Object[] parameters, long seed) {
        interArrivalTime =
            RandomVariateFactory.getInstance(
                distribution, parameters, seed);
    }
    public void reset() {
        super.reset();
        numberArrivals = 0;
    }

    public void doRun() {
        super.reset();
        firePropertyChange("numberArrivals",
            numberArrivals);
        waitDelay("Arrival",
```



```
        interArrivalTime.generate());
    }
    public void doArrival() {
        firePropertyChange("numberArrivals",
            ++numberArrivals);
        waitDelay("Arrival",
            interArrivalTime.generate());
    }
    public void setSeed(long seed) {
        interArrivalTime.getRandomNumber().
            setSeed(seed);
    }
    public long getSeed() {
        return interArrivalTime.getRandomNumber().
            getSeed();
    }
    public static void main(String[] args) {
        SimEntityBase arrival =
            new ArrivalProcess(
                "simkit.random.ExponentialVariate",
                new Object[] {new Double(3.2)},
                CongruentialSeeds.SEED[0]);
        simkit.Schedule.reset();
        simkit.Schedule.setVerbose(true);
        simkit.Schedule.stopOnEvent("Arrival", 5);
        simkit.Schedule.startSimulation();
    }
}
```

To run the example, make sure that you have jdk1.2 or greater and that the simsystem.zip file is on the classpath. The source code should be in a subdirectory called examples. To compile for the command line, change to the directory just above examples and enter (if running on Unix or a Unix-like OS, change the slash to a forward slash).

```
javac examples\ArrivalProcess.java
java examples.ArrivalProcess
```

