# Discrete Event Simulation Modeling

Arnold Buss
Research Associate Professor
MOVES Institute
Naval Postgraduate School
700 Dyer Road
Monterey, CA 93943

abuss@nps.edu

# Table of Contents

# 1. The Basics: States, Events, and the Event List

We begin with a minimal description of the elements in a Discrete Event Simulation (DES) model. These elements are:

1. States

2. Events

3. Scheduling relationships between events

It is furthermore necessary to have an understanding of how time advance works in DES, namely the *next event* form of time advance. This is implemented using the *future event list*. A small extension to the minimal DES description is the inclusion of *parameters*, which enable many models to be specified with a single description.

## 1.1. States

A *state variable* in a DES model is one that has a possibility of changing value at least once during any given simulation run. The collection of all state variables for a given DES model should give a complete description of the simulation model at any point in time. The collection of all state variables is called the *state space*. The value of a given state variable over time is called as *state trajectory*. DES state trajectories are not arbitrary, but must be piecewise constant, as shown below in Figure 1-1. That is, the state must stay constant in a given value for a certain period of time, then instantaneously change value to something else and stay at that value for another period of time. Thus DES state trajectories are extremely simple.
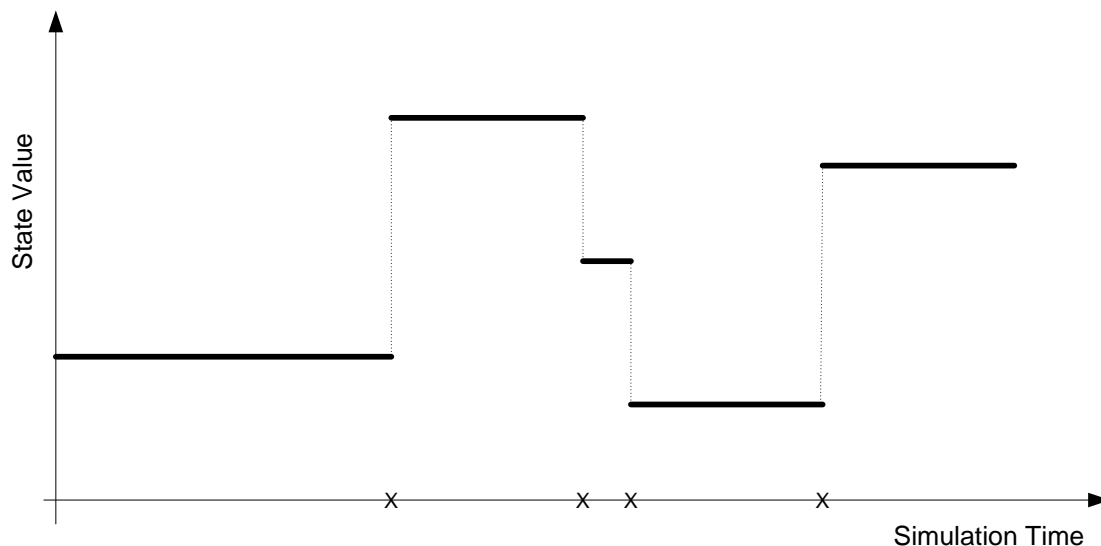


**Figure 1-1. A Typical DES State Trajectory**

However, not every possible variable is capable of being represented as DES state variable because of this restriction. Any variable whose state trajectory is not piecewise constant cannot be represented as a DES state. Figure 1-2 shows several state trajectories for variables that cannot be represented as DES states.
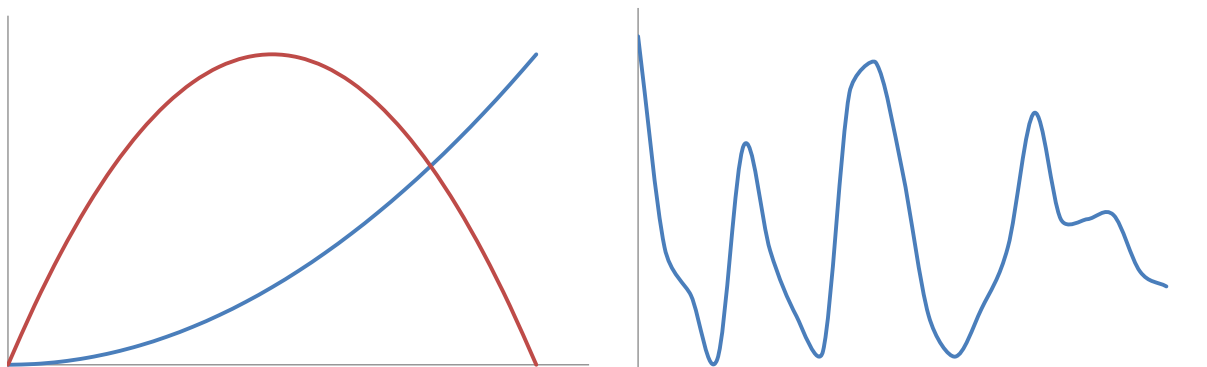
**Figure 1-2. State Trajectories for Non-DES States**

The inability to represent state trajectories such as those shown in Figure 1-2 may seem to make Discrete Event Simulation a very restrictive methodology. However, it turns out that restricting state variables to have piecewise constant trajectories as in Figure 1-1 is not as much of a limitation as might appear. It turns out that variables that change continuously *can* be modeled in DES – the modeling of such states requires a bit more effort, and will be discussed later. For now, the reader will have to take it on faith that DES is a sufficiently flexible approach to modeling many varied systems and that the restriction of DES state variables is not such a big liability. It turns out that the benefits can outweigh the limitations.

One advantage of this restriction is that the focus of the model can be on the rules by which each state variable changes value. Such a rule is called a *state transition function*, or more simply a *state transition*. Because each state transition occurs instantaneously (in simulated time at least), each state transition function can be identified with an instantaneous occurrence of an *Event*. The marks on the horizontal axis in Figure 1-1 show the occurrence of Events for that state variable. These Events are the building blocks of a DES model, and are one of the primary reasons why restricting possible DES states is a beneficial tradeoff.

The next step in the construction of a DES model is therefore to define each state transition as an event and give them each a name associated with that state transition. A particular simulation run thus consists of a sequence of events, whose state transitions result in state trajectories for each state variable.

## 1.2.   Events

A DES Event begins by defining its state transition, as described above. An Event can change a few state variables (possibly even none) or it may cause many state variables to change. Each state transition is a mapping from a model's state space into itself. For each possible state transition an Event must be defined. Everything else being equal, it is best to define models that have many simple state transitions than a few complex state transitions.

Defining the Events alone is not sufficient to fully describe how a simulation run will unfold. Also needed are rules that determine what the next Event will in fact be. The way a DES model describes this is by forming scheduling relationships between Events. That is, a particular Event may cause another Event to be scheduled sometime in the future. For example, if an Event is that a ball is thrown in the air, a second Event might be that the ball lands on the

ground.  The first Event (Throw Ball) would schedule the second (Ball Lands) at a time in the future that depends on how the ball was thrown.  Note that the trajectory of the ball itself might resemble the parabola in the left graph of Figure 1-2.  However, in many cases we might not care so much about the exact location of the ball at every point in time, but only in the way it changes things when it lands.[1]

Thus, each Event is completely defined by specifying its state transition function.  A complete specification of a DES model therefore has an event for every possible change of state that could occur, that is, for every possible state transition function.

The DES model is not yet complete, however, when all state transitions are defined.  The final element in the model description is to specify for Event every possible future Event it could directly "cause."  This is the *scheduling relationship* between events, and is seen as a directed relationship between the collection of Events.  As will be seen soon, these relationships can be expressed simply and intuitively as a graph.

## 1.3.  Time Advance

The method of time advance in DES models is termed *Next Event*.  Rather than advancing time in a regular, consistent manner, simulation time moves in typically unequal increments, jumping from the scheduled time of one Event to another; thus, the term *Next Event*.
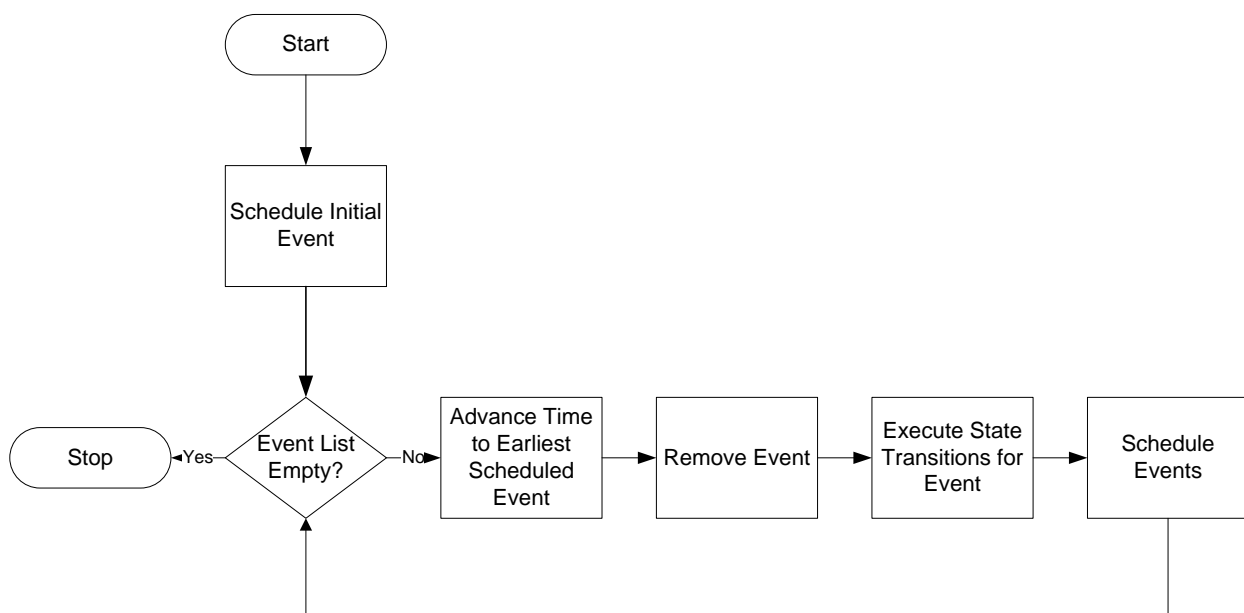


**Figure 1-3. Next Event Algorithm**

The Next Event algorithm is shown in Figure 1-3, and as can be seen is quite simple. However, each step does need some explanation.

1. **Schedule Initial Event.  An Event is placed on the Event List scheduled to occur at time 0.0.**  This special event (which we will call Run) is responsible for initializing all state

---

[1] If we are indeed interested in the exact location of the ball we can model that in DES as well, as it turns out.

variables as its "state transition" as well as scheduling any initial "real" events of the model. Otherwise, Run is treated like any other event.

2. **Is Event List Empty?** If the Event List is empty, then stop; otherwise proceed to the next step.

3. **Advance Time to Earliest Scheduled Event.** Simulation time is "advanced" to the time of the earliest scheduled event. Note that this could result in time being incremented by a very large amount or by a very small amount; it is possible that the "advance" may be 0.0.

4. **Remove Event**. The Event from the previous step is removed from the Event List.

5. **Execute State Transitions for Event**. All state transitions associated with the Event are executed, resulting in an instantaneous change of state according to the state transition function description.

6. **Schedule Events.** The scheduling of future Events by this Event is performed, if called for by the model. This scheduling consists simply of placing new Events on the Future Event List described in the next section. No other state changes occur. An Event may or may not schedule future Events; furthermore, whether it does or doesn't may be conditionally determined by the state. That is, whether another Event is scheduled may depend on the state of the simulation after that Event's state transitions are executed.

When all Events that are specified have been scheduled (if any), the process continues by returning to Step 2.

Figure 1-3 can also be summarized by the following pseudo-code.

```
Initialize:
      Set simulation time to 0.0
      Schedule the Run Event
While (There are pending Events):
      Advance time to next Event
      Remove Event from Event List
      Execute State transition for Event
      Schedule any Events as specified by the model
```

Understanding the Next Event algorithm is critical to effectively creating DES models. The precise mechanism for implementing it may vary, but the logic remains the same. Most important perhaps is the concept of the Future Event List, which is explained further in the following section.

A common implementation of the Initialize step is to identify a special Event that is always scheduled to occur at time 0.0. This Event, which we call Run, has state transition that sets each state variable to its initial value and schedules at least one other Event. The advantage of this approach is that the Run Event, once placed on the Event List, is simply processed like any other Event, so the only special treatment is the fact that the Event List bootstraps the model by simply scheduling Run to occur at time 0.0.

It should also be noted that there will be an additional step in processing simulation Events when the ability to cancel pending Events is introduced later. Thus, Figure 1-3 and the elaboration in this section should be seen as a basic template that will be expanded on subsequently.

## 1.4. Future Event List

The *Future Event List* (FEL) is the mechanism by which pending Events are held and managed. The FEL is responsible for holding information about every pending event and keeping them in order. More precisely, the FEL just needs to be able to identify the pending event with the earliest scheduled time. The contents of the FEL are sometimes called *Event Notices*, and they contain all the information necessary to process the Event. At the minimum, each Event Notice must contain an identifier for which Event to which it corresponds and the scheduled time for that occurrence of the Event (so the FEL can keep the Event Notices sorted).

The FEL also needs to be able to add Events while maintaining time-sorted order and remove the next scheduled pending Event Notice. If canceling of Events is supported, it also needs to be able to find and remove the Event Notice corresponding to the cancelled Event.

These are of course minimal requirements for a FEL. In order to give effective support for model development, a FEL should also be able to run "verbosely" – that is, to print out each step it is taking in the algorithm of Figure 1-3.

For example, one very useful tool for verifying a DES model is for the FEL to print each Event as it is being processed together with the current value of simulation time and the list of pending Events.

## 1.5. Simulation Parameters

So far the only variables in a DES have been state variables. Recall that state variables are quantities that describe a portion of the system being modeled and that they have the possibility of changing value throughout a simulation run, resulting in a state trajectory.

A second type of variable is important, namely values that do *not* change during a simulation run. These are termed *Simulation Parameters* (or when there is no possibility of misunderstanding, simply *Parameters*).

An example of a simulation parameter might be the number of servers in a queueing system being modeled. Another might be the number of machines in a machine shop. Still another might be the maximum speed a vehicle can travel.

Often modeling sequences of random variables are utilized in a DES model; indeed, it is precisely the need to incorporate randomness that leads an analyst to utilize DES rather than an analytic technique. For purposes of specifying DES models, it is convenient to be as flexible as possible in defining such random sequences. Thus, such a sequence will typically be defined to be a "simulation parameter" even though it is actually a sequence. The "parameter" should be thought of as the specification of the sequence. Thus, a model might identify  as a sequence of random variables, and this sequence would be a parameter. When the model is actually executed, of course, the sequence must be specified, just as for the number of servers parameter the actual value must be given at run time. Also, when a DES model is specified using such a sequence as a parameter, then whenever the sequence variable appears in the model description, it is meant that the next random number in the sequence is generated.

## 1.6. Stopping Criteria (Terminating Conditions)

There are several ways in which a simulation run can be terminated. As noted in Figure 1-3, whenever the Event List is empty the simulation terminates. So ending a simulation run can be simply implemented by emptying the FEL whenever a terminating condition is met.

The simplest terminating condition is based on time: the run will end after a certain amount of simulation time has passed. This is simply implemented by a special event ("Stop") that doesn't affect any state, but simply empties the Event List.

Another terminating condition is based on how many times a particular Event has been executed. This condition is most effectively implemented within the FEL itself, since it requires keeping count of the given Event and emptying the FEL when the desired count has been reached.

Yet another termination condition is based on a certain state being reached; that is, a given state variable having a determined value. Implementing this terminating condition is most effectively done by the model itself: every Event that changes the value of the given state variable conditionally scheduling a Stop event if the desired value is reached. This can of course be extended to a collection of states entering a pre-determined region of the state space. The implementation is similarly done: every Event that changes the value of any state variable in the collection would conditionally schedule a Stop Event when the stopping region is entered.

Although every execution of a simulation run must have a stopping criterion, it will typically be omitted from the description of the DES model itself.

## 1.7. Defining a DES Model

Now that all the essential elements of a DES model have been presented as well as the execution of the Next Event processing, defining a DES model can be summarized by specifying four elements:

1. Define the parameters of the model (the variables that will not change during a single run).

2. Define the state variables of the model (the variables that will change in piecewise constant state trajectories) and for each state variable specify its initial value.

3. Define each Event by specifying its state transition and assigning a unique name to the Event.

4. Define the scheduling relationships between Events. For each Event that could schedule another, give the condition under which it will be scheduled and the amount of time in the future ("delay") the Event will be scheduled to occur.

Some simple examples will illustrate this.

## 2. Examples

Two simple examples of DES models will now be presented.

### 2.1.1. Arrival Process

The simplest non-trivial example of a DES has one parameter, one state variable, and one Event (besides the Run Event). It models a single Event that recurs periodically, where the time between occurrences of the Event are possibly random.

1. Parameter

    1. is the sequence of (possibly random) times between the occurrences of the Event.

2. State

    N is the number of times the Event has occurred. Its initial value is 0.

3. Events:

    Run will set N to its initial value of 0
    2. Arrival will be the name of the Event, and its state transition will be to increment the value of N by 1. More precisely, the state transition function is defined by: $N = N + 1$.

4. Scheduling:

    When the Run Event occurs, it schedules the first Arrival Event to occur at time units. When an Arrival Event occurs, it schedules another Arrival Event to occur at time units in the future.

### 2.1.2. Multiple Server Queue

Customers arrive to a service facility one at a time. There are a limited number of servers at the facility, and they can only serve one customer at a time. An arriving customer who finds all servers are busy waits in a line (queue) in the order of arrival. That is, the queue is first-come first-served.

1. Parameters:

    - $k$ is the total number of servers in the system
    - is the sequence of (possibly random) times between the arrival of customers to the system.
    - is the sequence of (possibly random) service times of each successive customer.
2. States:

    - Q is the number of customers in the queue; the initial value is 0.
    - S is the total number of available servers (between 0 and $k$). The initial value is $k$.
3. Events:

    - Run will set Q to its initial value of 0 and S it its initial value of k.
    - Arrival increments the value of Q by 1 ($Q = Q + 1$)
    - StartService decrements Q by 1 and S by 1 ($Q = Q - 1$; $S = S - 1$).

- EndService increments S by 1 (S = S + 1).
4. Scheduling:

- Run schedules the first Arrival Event to occur at  time units.
- Arrival schedules another Arrival Event to occur at $\{t_A\}$ time units in the future.  If there is an available server (S > 0) then a StartService is scheduled with a delay of 0.0
- StartService schedules an EndService with a delay of $\{t_S\}$ time units.
- When EndService occurs, if there is at least one customer in the queue (Q > 0) then a StartService is scheduled with delay of 0.0.

Note that the description of Events and Scheduling (Steps 3 and 4) tend to be somewhat verbose when compared to the underlying model being described.  This verbosity will be ameliorated when Event Graphs are introduced later.

## 2.2.    Running a Simulation By Hand

In order to illustrate the working of the Event List, a simple model will be run "by hand."

A single-server queue (that is, a multiple-server queue with $k = 1$) has interarrival times of {5.6, 2.9, 5.7} (these are the first few values of $t_A$) and service times of {4.7, 2.4} (these are the first few values of $t_S$ ).  Since we are interested in the number of customers in the system, an additional state variable L is added to the model.  At any time the value of L is the total number of customers in the queue plus those in service.  Note that L can be expressed in terms of the variables already defined in the model by $L = Q + k$ - S.

The simulation will be run until the second customer has completed service.  The current time and Event will be shown, together with the value of state variables and the status of the Event List.  Initially all simulation replications will start the same way after initialization of the Event List.  The current time is set to 0.0, the current event and the values of the state variables are undefined, and the Run event is on the Event List scheduled at time 0.0.  This is illustrated in Figure 1-3.

| Current Time | Current Event | Q | S | L | Event List |
|---|---|---|---|---|---|
| 0.0 | - | - | - | - | 0.0 Run |

**Figure 2-1. Initial Status of Simulation**

Execution of the Event List algorithm is ideally performed as mechanically as possible.  Each iteration, starts by advancing time to the earliest pending Event and removing it.  Next, the state transitions are performed, and finally any additional Events to be scheduled are put on the Event List.  In this case, time is "advanced" to 0.0 and the Run event is removed from the Event List.  Next, the values of Q and S are set to their initial values (0 and 1, respectively).  The additional state variable L, has its value computed from the new values of Q and S: $L = 0 + 1 - 1 = 0$.  Finally, Run schedules Arrival after a delay of $t_A$.  Since the first value in the sequence is 5.6, the Event List now has an Arrival Event scheduled to occur at time $0.0 + 5.6 = 5.6$.  The result of this step is shown in the last line of Figure 2-2.

| Current Time | Current Event | Q | S | L | Event List |
|---|---|---|---|---|---|
| 0.0 | - | - | - | - | 0.0 Run |
| 0.0 | Run | 0 | 1 | 0 | 5.6 Arrival |

**Figure 2-2. After the Run Event is Processed**

The process is now repeated. Time is advanced to 5.6 and Arrival is removed from the Event List. Q is incremented by 1 to 1 and L is updated to reflect the new values. Finally, since there is an available server ($S > 0$), a StartService Event is scheduled along with the next Arrival Event.

| Current Time | Current Event | Q | S | L | Event List |
|---|---|---|---|---|---|
| 0.0 | - | - | - | - | 0.0 Run |
| 0.0 | Run | 0 | 1 | 0 | 5.6 Arrival |
| 5.6 | Arrival | 1 | 1 | 1 | 5.6 StartService<br>8.5 Arrival |

**Figure 2-3. After Processing the First Arrival Event**

Note that *both* the Events scheduled by Arrival are put on the Event List. The StartService Event has a delay of 0.0, so its schedule time is $5.6 + 0.0 = 5.6$. The next Arrival Event has a delay of 2.9, so is scheduled to occur at time $5.6 + 2.9 = 8.4$. Note that because all scheduling does is place an Event on the Event List, the order in which Events are actually scheduled for any given Event is irrelevant. The Event List keeps the Events sorted in order of scheduled time.

Continuing, the final sequence of the hand simulation is shown in Figure 2-4.

| Current Time | Current Event | Q | S | L | Event List |
|---|---|---|---|---|---|
| 0.0 | - | - | - | - | 0.0 Run |
| 0.0 | Run | 0 | 1 | 0 | 5.6 Arrival |
| 5.6 | Arrival | 1 | 1 | 1 | 5.6 StartService<br>8.5 Arrival |
| 5.6 | StartService | 0 | 0 | 1 | 8.5 Arrival<br>10.3 EndService |
| 8.5 | Arrival | 1 | 0 | 2 | 10.3 EndService<br>14.2 Arrival |
| 10.3 | EndService | 1 | 1 | 1 | 10.3 StartService<br>14.2 Arrival |
| 10.3 | StartService | 0 | 0 | 1 | 12.7 EndService<br>14.2 Arrival |
| 12.7 | EndService | 0 | 1 | 0 | 14.2 Arrival |

**Figure 2-4. Complete Hand Simulation**

The Arrival at time 8.5 does not schedule a StartService Event because S = 0; that is, there is not an available server. However, the next Arrival event is scheduled. The EndService Event at time 10.3 schedules a StartService at time 10.3 because Q > 0 (i.e. there is a customer waiting in the queue). However, the EndService Event at time 12.7 does not schedule StartService because Q = 0 (i.e. there are no customers in the queue).

## 2.3. Gathering Statistics

The primary purpose of a DES simulation model is to learn something about the system being modeled that wasn't known previously. In order to accomplish this, statistics must be gathered from running the simulation. A complete discussion of statistical analysis of simulation models is beyond the purpose here. However, it is important to create DES models with an eye towards the analysis that will ultimately performed on them.

There are several important types of statistics commonly used in analyzing DES models: counts and rates, tally averages, and time averages. We will illustrate each of these using the small example from the previous section.

A count is simply the number of times something has occurred. In the queueing example of the previous section, at time 12.7 (the ending time of the simulation) there have been 2 arrivals. Counts can be used to estimate rates by dividing by simulation time. In this small

example, since there have been 2 arrivals at time 12.7, the average arrival rate of customers to the system is $2/12.7 \approx 0.16$.

A tally average is the average of a number of discrete observations. In our example, let's focus on the average time in the system. The data in Table 1 show the time each customer arrived and departed; the difference between the two is the time the respective customer spent in the system.

| Customer | Arrived | Departed | Time in System |
|----------|---------|----------|----------------|
| 1 | 5.6 | 10.3 | 4.7 |
| 2 | 8.5 | 12.7 | 4.2 |

**Table 1. Data for Customers' Time in System**

For this small amount of data, our estimate of the average time in the system would be $(4.7 + 4.2)/2 = 4.45$. Obviously no solid conclusions can be made based on so few observations, and this computation is shown for illustrative purposes only. There are additional issues, namely that the data from a DES model typically do not meet many of the "standard" statistical assumptions. For example, the two times in the system here are neither independent nor are they identically distributed. These, and other issues, will be dealt with when output analysis is covered.

The third type of statistic, the time-average, is computed differently than a tally average. A time average is computed by first finding the area under the state trajectory and dividing it by the simulation time. Time averages are common in DES models because of the nature of DES state variables.

Let's compute the average number of customers in the system for the queueing example of the previous section. A graph of the state trajectory for the number in the system in the queueing example is shown in Figure 2-5. From the graph, it should be clear that the "normal" way of computing the average would not be appropriate because it doesn't take into account the amount of time spent in each state.
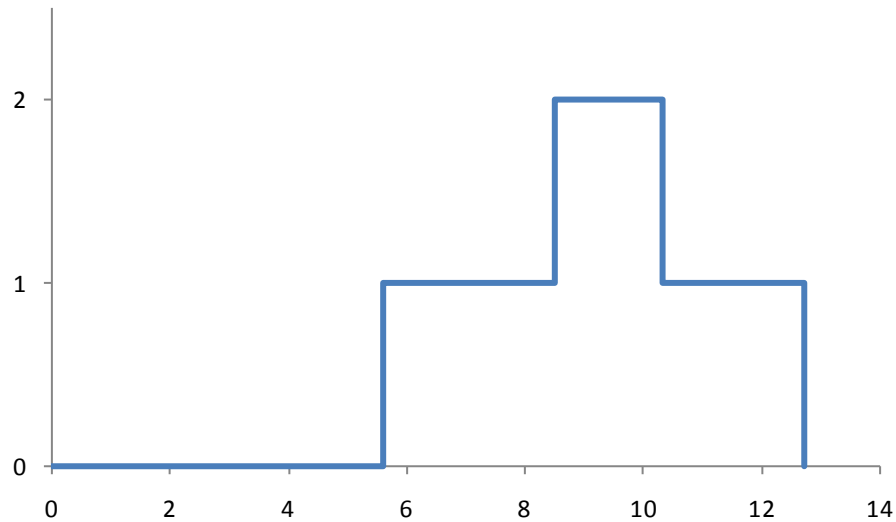
**Figure 2-5. State Trajectory for Number in System (L)**

There are several ways to compute the area under the curve. The most convenient way in a DES model is to keep a running total and update it every time the state changes value. This is illustrated in Table 2 below.

| Time | L | Area Under Curve |
|------|---|------------------|
| 0.0 | 0 | 0.0 |
| 5.6 | 1 | $0.0 + 0 (5.6 - 0.0) = 0.0$ |
| 8.5 | 2 | $0.0 + 1(8.5 - 5.6) = 2.9$ |
| 10.3 | 1 | $2.9 + 2(10.3 - 8.5) = 6.5$ |
| 12.7 | 0 | $6.5 + 1(12.7 - 10.3) = 8.9$ |

**Table 2. Computing the Area under the Curve for L**

The area is 0.0 at time 0.0, so the initial value always starts there. At time 5.6 the value of L changes to 1, but the area is still 0.0 because it had been 0.0 since the previous time of 0.0. At time 8.5 L changes again to 2, and this time the area is 2.9 since L has been 1 from time 5.6 to 8.5. At time 10.3 the area is incremented by 2.6, etc. When the simulation ends at time 12.7, the area under the state trajectory for L is 8.9. Therefore, the average number in the system at time 12.7 is $8.9/12.7 \approx 0.54$.

As with the average time in the system, there is obviously insufficient data to draw any reasonable conclusions about the system. In fact, the amount of data that can be obtained from running a simulation by hand this way will never be sufficient for analysis. That is why implementation and execution on the computer is so important.

2-12

An important time varying average is used as a measure of server performance in many queueing systems, namely the average amount of time a server is busy. This average is called *utilization*. The server utilization for this small example can be determined by first computing the time average number of available servers. The state trajectory for S is shown in Figure 2-6.
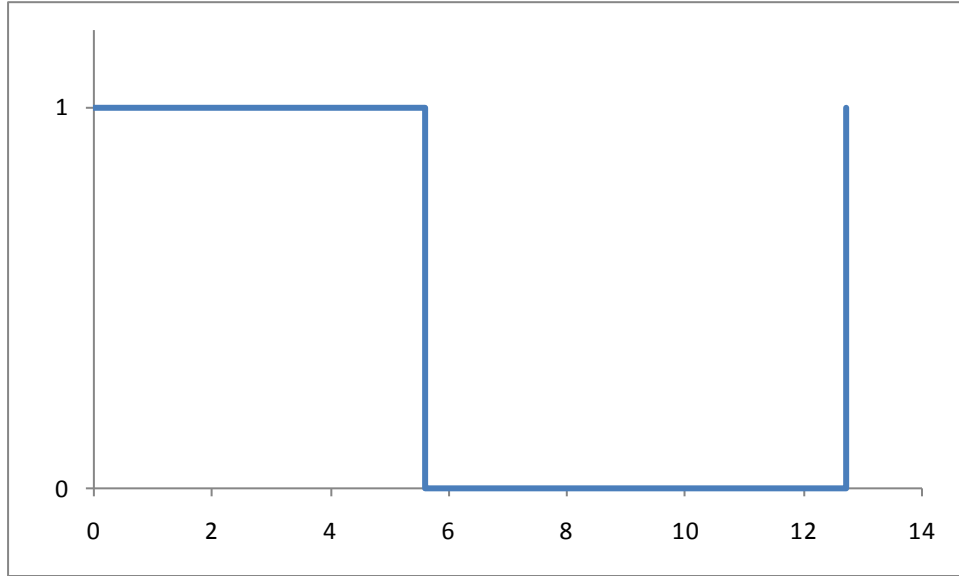


**Figure 2-6. State Trajectory for Number of Available Servers (S)**

The area under the graph is easily found to be 5.6, so the time varying average of S at the end of the run is 5.6/12.7 ≈ 0.44. Since the number of busy servers at any given time is k – S, the average utilization is therefore 1 – 0.44 ≈ 0.56.

## 2.3.1. Definitions

To summarize the different statistics discussed above, let us give more formal definitions for each of them.

1. **Count.** Simply the number of times something has occurred. Often this can be determined by observing a state variable, such as the number of arrivals, N, in the ArrivalProcess model described previously.

2. **Average Rate.** A count divided by simulation time. If $N$ is a count and T the value of simulation time, then the estimate of the average rate of occurrence is $\bar{\lambda} = N/T$.

3. **Tally Average.** The observations are discrete, and the tally average is simply the "standard" mean. If $X_1, \ldots, X_n$ are the observations, then the tally average is simply $\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$

4. **Time Varying Average.** The observations form a state trajectory; that is, there are no discrete observations, just the values of the state it changes over time. If $X(t)$ is the value of the state at time t, then the time varying average at simulation time $T$ is given by

$$\bar{X}(T) = \frac{1}{T}\int_0^T X(t)dt.$$

2-13

Note that this definition of a time average holds even if $X(t)$ is not a DES state trajectory. That is, regardless of the form of $X(t)$, the time average is given by the above integral. For complex curves it can be difficult to find the value and approximate methods must be used. Fortunately, since DES state trajectories are piecewise constant, the only "calculus" needed is the ability to compute the area of a rectangle.

5. **Average Utilization.** For a queueing system with $k$ identical servers and the number of available servers at time $t$ given by $S(t)$, the average utilization is $1 - \dfrac{1}{kT} \int_0^T S(t)dt$ .

## 2.3.2. Little's Formula

In computing the tally average time in system and time varying average number in system above, it may be observed that the former required much more knowledge about the simulation run than the latter. That is, to find the time each customer spent in the system it had to be "remembered" when each one first entered. In contrast, the average number in the system only required knowledge of the current state and the running total area. A consequence of this is that the DES model for the queueing system is not capable of directly estimating the average customer time in the system. This would seem to be a serious flaw in the model; however, there is a result called *Little's formula* that obviates the need for directly computing the average time in the system.

In this context, Little's formula can be stated as follows. Consider a "system" in which entities arrive, enter, stay awhile, and then leave. Let $\overline{L}$ be the average number of these entities in the "system" at a given time T, let $\overline{W}$ be the average time in the system for those entities, and let $\overline{\lambda}$ be the average arrival rate of the entities to the system. Little's formula is: $\overline{L} = \overline{\lambda}\overline{W}$ .

There are times when Little's formula holds exactly, times when Little's formula holds approximately, and times when Little's formula doesn't hold at all. It turns out that whenever the "system" is empty, then Little's formula holds exactly. This means that for these times, the value of $\overline{W}$ can be found as $\overline{W} = \overline{L}/\overline{\lambda}$ .

For a system that achieves some kind of "steady state" (which shall be left somewhat vague for now), Little's formula can be shown to hold approximately. For systems that do not reach any kind of steady-state, Little's formula cannot be said to hold in general or even approximately (but even for these systems, whenever they are empty Little's Formula does hold nevertheless). Finally, during the transient (early) period of a simulation when the system is not empty, Little's Formula has little to offer. Fortunately, neither of these instances is typically of interest.

Let's verify Little's formula in our small example. Recall that (using the notation in this section) we had: $\overline{L} = 8.9/12.7$, $\overline{W} = 8.9/2$, and $\overline{\lambda} = 2/12.7$, so $\overline{\lambda}\overline{W} = \left(\dfrac{2}{12.7}\right)\left(\dfrac{8.9}{2}\right) = \dfrac{8.9}{12.7} = \overline{L}$.

Note that the computations were done exactly, so the Little's formula does indeed hold exactly at time 12.7, since the system is empty at that time.

The word "system" was put in quotes because it applies to any sub-system as well. For example, the queue in our example can be considered a "system" and Little's formula holds for it

as well, under the same circumstances.  For example, when there are no customers in the queue, it holds exactly.  The reader should be able to easily show that the average number in the queue at time 12.7 is 1.8/12.7 and that the average delay in the queue is $1.8/2 = 0.9$.  The average arrival rate to the queue is identical to that of the system.

Note that at time 12.0 Little's formula for the overall system does *not* hold, but Little's formula for the queue *does* hold.

## 2.4.   Next Steps

Sufficient material has now been presented to create DES models.  However, the language for describing Events and scheduling relationships is wordy and can be unintuitive.  The next step is to introduce a compact and intuitive representation of a DES model.  This representation is called an Event Graph

## 3.    Basic Event Graph Modeling

The description of scheduling Events in the basic Discrete Event Simulation (DES) framework indicates a directed, binary relationship between the Event that is being "processed" and an Event that is scheduled. This suggests a way to represent this activity using a graph, called appropriately an *Event Graph*.

An Event Graph consists of nodes and directed edges.  Each node corresponds to an Event, or state transition, and each edge corresponds to the scheduling of other events.  Each edge can optionally have an associated Boolean condition and/or a time delay.  Figure 3-1 shows the fundamental construct for Event Graphs and is interpreted as follows: the occurrence of Event A causes Event B to be scheduled after a time delay of t, providing condition (i) is true (Note: the Boolean condition (i) is evaluated after the state transitions for Event A have been performed).  By convention, the time delay t is indicated toward the tail of the scheduling edge and the edge condition is shown just above the wavy line through the middle of the edge. If there is no time delay, then t is omitted. Similarly, if Event B is always scheduled following the occurrence of Event B, then the edge condition is omitted, and the edge is called an unconditional edge. Thus, the basic Event Graph paradigm contains only two elements: the event node and the scheduling edge with two options on the edges (time delay and edge condition).
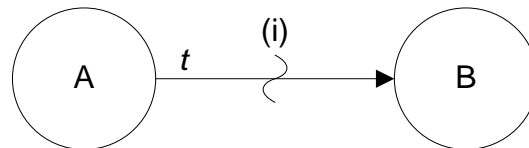


**Figure 3-1. Fundamental Event Graph Construct**

An Event Graph makes the relationships between Events clear and intuitive.  A complete Event Graph model consists of the parameters and state variables, as with a "vanilla" DES model, plus the Event Graph itself, which defines the Events with their state transitions and the scheduling relationships defined by the edges of the graph.

### 3.1.   Simple Examples

The two simple models describes previously will now be formulated as Event Graph models.

### 3.2.   The ArrivalProcess

**Parameter**

- $\{t_A\}$ is the sequence of (possibly random) times between the occurrences of the Event.

**State**

- N is the number of times the Event has occurred.  Its initial value is 0.

**Event Graph:**
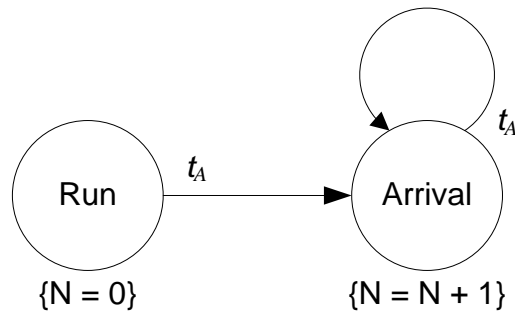


**Figure 3-2. ArrivalProcess Event Graph**

For simple Event Graphs like this one, it is convenient to put the state transitions beneath the Event nodes. For larger models this may make it more cluttered and less intuitive, and so may be defined separately. If the parameter given by the sequence of interarrival times $\{t_A\}$ is random, then the appearance of $t_A$ in Figure 3-2 is interpreted to mean that a new value is generated every time it is encountered. In that case, each value for the time delay when scheduling in Figure 3-2 would be a different value.

Note that the Event scheduling relationships in Figure 3-2 make it quite clear which Events cause others to occur: that Run schedules the first Arrival Event and each occurrence of Arrival schedules the next occurrence of Arrival.

## 3.3. The Multiple Server Queue

**Parameters:**

- $k$ is the total number of servers in the system
- $\{t_A\}$ is the sequence of (possibly random) times between the arrival of customers to the system.
- $\{t_S\}$ is the sequence of (possibly random) service times of each successive customer.

**States:**

- Q is the number of customers in the queue; the initial value is 0.
- S is the total number of available servers (between 0 and $k$). The initial value is $k$.

**Event Graph:**
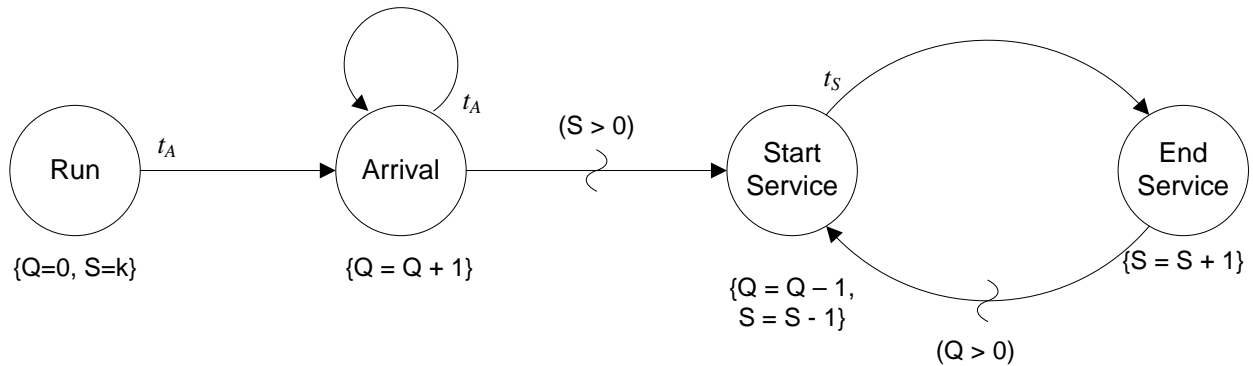


**Figure 3-3. Multiple Server Queue Event Graph**

For a slightly more complex model such as this one, the contribution of Event Graphs is even clearer than for the previous model. Compare Figure 3-3 with the verbal description of the same model previously. Both describe the same model, but the Event Graph is much easier to grasp and understand, once the meaning of the nodes and edges is known.

One caution however is to keep in mind that the nodes are Events and the edges scheduling relationships. Thus, when an edge is "executed," it has the sole effect of placing an Event on the Event List, nothing more. When reading the Events from left to right in Figure 3-3 it can be tempting to interpret it as a flow chart. This should be avoided, since that is not what the meaning is.

## 4. More Event Graph Modeling Concepts

Although the simple Event Graph methodology as presented so far can be very useful in designing DES models, there are two concepts that turn out to be extremely useful in creating simple and flexible models. These are the ability to cancel an Event after it has been scheduled and the ability to pass arguments on scheduling edges and have the values received by the scheduled event.

### 4.1. Cancelling Edges

With the basic DES methodology so far the only way Events can be removed from the Event List is by becoming the next event and being processed. Although it is not strictly necessary, it turns out that the ability to remove an Event from the Event List, that is to cancel it, is very convenient for helping to create parsimonious models. The cancelling edge triggers the removal of a single previously scheduled Event from the Event list and is indicated by a dashed arrow instead of the solid arrow of a scheduling edge, as shown in Figure 4-1. It can have an optional Boolean expression, just as with a scheduling edge, meaning that the Event pointed to is cancelled only if it evaluates to 'true.'



**Figure 4-1. Prototypical Cancelling Edge**

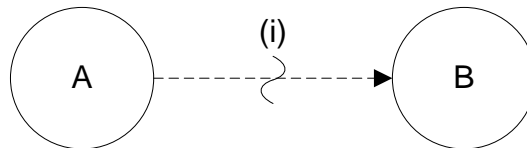Precisely, the interpretation of Figure 4-1 is as follows: whenever Event A occurs, then (following its state transition), if condition (i) is true, then the earliest scheduled occurrence of Event B is removed from the Event List. If no such Event had been previously scheduled, then nothing happens; it is not considered an error. If Event B had previously been scheduled multiple times, then only the earliest scheduled one is removed and the remaining ones are left on the Event List.

It is important to remember that the cancelling edge in Figure 4-1 has only one opportunity to remove Event B for any given occurrence of Event A. If the Boolean condition (i) evaluates to false, then that opportunity is "lost" – the cancelling edge will not "wait" for (i) to become true. Of course a subsequent occurrence of Event A may result in (i) being 'true' again, in which case the cancellation will in fact occur. Note that there is no time delay associated with a cancelling edge. If there is a desire to cancel an Even after a certain delay, then the modeler can first schedule one event with that delay, and when that Event occurs, the desired Event is cancelled.

### 4.2. Parameters on Events and Arguments on Edges

Another feature that is not strictly necessary but one that makes models more extensible is that of passing arguments on scheduling edges. In order for this to make any kind of semantic sense, the Event being scheduled in such a case must have formal parameters for each argument being passed. Conversely, it is often useful for data to be passed to an Event, much like an

argument being passed to a function or a method in computer programming. For the Event to have the appropriate data to work with, it must be passed. Since the scheduling edge is the mechanism by which an Event is "created," it is natural to insist that the data be passed on the scheduling edge.

This raises an issue about cancelling such Events. Event Graph methodology states that to cancel a previously scheduled Event with parameters, the cancelling edge must contain identical parameters as well.

A parameter on an Event is indicated by a list of variables in parentheses, similar to the syntax of parameters on a method in computer programming. The argument on the edge is a list of expressions that match the Event's parameters syntactically, just as with a strongly-typed programming language. The prototypes for a scheduling edge with arguments and an Event with parameters is shown in Figure 4-2. Note that although a single parameter/argument is indicated, multiple values may be passed as well.
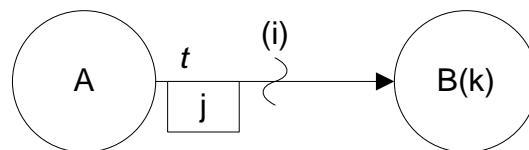


**Figure 4-2. Scheduling Edge with Arguments and Events with Parameters**

The interpretation of the scheduling prototype in Figure 4-2 is as follows. When Event A occurs, then if condition (i) is true, Event B is scheduled to occur (placed on the Event List) after a delay of $t$, and when it occurs its parameter k will be set to the value of the expression j at the time it had been scheduled.
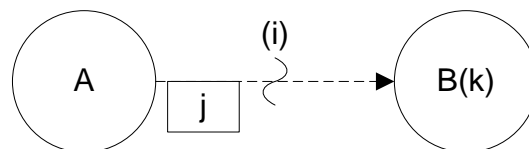


**Figure 4-3. Cancelling Edge with Arguments and Events with Parameters**

The cancelling prototype in Figure 4-3 is interpreted as follows: when Event A occurs, then if condition (i) is true, the earliest occurrence of a previously scheduled Event B whose parameter was passed value j is removed. If no such Events had been scheduled, nothing happens. If multiple Events meet the criterion, then the earliest of them only is removed. Any Event B that had been scheduled but with parameter *not* equal to j is not considered to be a match.

## 4.3.  Priorities on Scheduling Edges

Sometimes in extreme cases it is necessary to break ties for Events that are scheduled at exactly the same time. In most models for which times are continuous random variables this shouldn't occur. The modeler can specify a tiebreaker for Events that might occur simultaneously and for which the order matters. This is done by setting a priority on the scheduling edge. By convention a higher numerical value means higher priority. Figure 4-4 shows a scheduling edge with priority set to p.
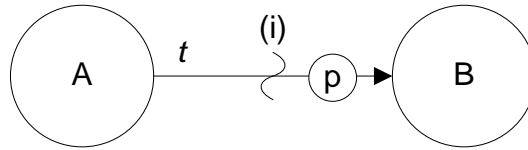
**Figure 4-4. Scheduling Edge with Priority**

If the priority is omitted, then the "default" priority is assumed. In most cases the use of zero-delay scheduling edges in the model when in fact there is a positive amount of delay in the actual system.

For example, in the multiple server queue Event Graph of Figure 3-3, with continuous interarrival and service times there will normally not be multiple Events on the Event List at identical times. However, if discrete random variates are used for the times then it is possible for an Arrival Event and a StartService Event to be simultaneously scheduled at exactly the same time.[2] If the value of S is 1 and the Arrival Event happens to be processed first, then it will schedule a StartService, with zero delay, thus having two StartService Events on the Event List. When the second one is processed, the value of S will then become -1, an impossible situation. The remedy is to schedule StartService with higher priority than Arrival so that even when they have identical scheduled times, StartService always is processed first.

## 4.4. Examples

### 4.4.1. "For" Loop

Event Graph methodology does not permit the use of "for" loops a common construct in computer programming. However, passing arguments on edges facilitates a simple Event Graph snippet that implements this functionality of a "for" loop by having a given Event be repeatedly scheduled with zero delay. For example, suppose it is desired to have an Event called Init(i) occur exactly m times at time 0.0, with . $i = 0, \ldots, n-1$ for the successive occurrences of Init. The Event Graph snippet in Figure 4-5
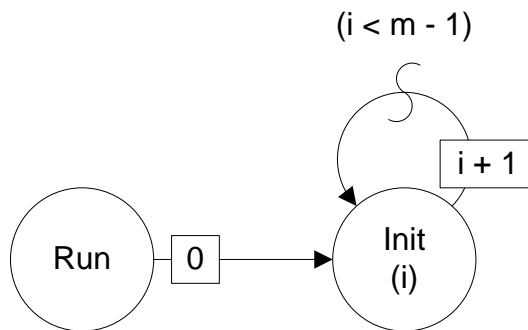


**Figure 4-5. Event Graph "For" Loop**

The reader should verify that Events Init(0), …, Init(m-1) will successively occur at time 0.0 when this is executed. Note that zero-based indexing is used here. This is because the most common simulation languages currently also use zero-based indexing for their arrays. The Event

---

[2] Note that when Arrival schedules a StartService they may occur at the same time, but they are never on the Event List at the same time.

Graph in Figure 4-5 could be modified to use one-based indexing; that is, for the successive values of $i$ to be 1,...,m. This simple modification is left as an exercise. The transfer line model below will use this construct to initialize its state variables.

## 4.4.2. A Simple Inventory Model with Backorders

A company is seeking to manage the inventory level for a single product. Customers purchasing the product arrive according to an arrival process. Each arriving customer attempts to purchase a random number of items, D. The inventory level at the company is reviewed periodically and a decision is made whether or not to place an order from its supplier. When the company places an order, it takes a certain amount of time ("lead time") for the order to arrive. Due to a variety of circumstances, the lead times are random. The company uses a $<s,S>$ ("little s big S") policy for its ordering decisions. If the inventory position at review time is below s, then an order is placed that is the difference between it and the number S. The inventory position includes the amount of the product on-hand and the amount of the product that is on-order (that is, has been ordered from the supplier but not yet received). Note that it is possible for there to be two or more outstanding orders from the supplier.

When a customer's order cannot be filled, the unfilled portion is put on backorder. When the company receives a shipment from its supplier, backorders are immediately filled and the remainder put in stock. For example, if there are 5 items in stock and a customer wants to buy 8 items, the customer is given the five items in stock and the remaining three are backordered.

Measures include the average amount of inventory on-hand, the average amount on backorder, and the average amount on-order, and the percentage of customers who get their orders filled immediately. The values of s and S are policy variables that can be chosen by the manager, who presumably wants to pick the "best" ones.

### Parameters

- $\{t_A\}$ = times between arrival of customers
- $\{D\}$ = number if items demanded by a customer
- $\{t_L\}$ = lead time for orders received by company
- $t_R$ = time between reviews. Note that this is will typically be deterministic, so it is not shown as a sequence
- $I_0$ = initial inventory
- S = order-up-to amount
- s = trigger amount

### States

- I = amount on-hand; initially $I_0$
- B = amount on backorder (i.e. owed to customers); initially 0
- T = total amount on-order (i.e. to be received); initially 0
- N = number items ordered ; initially 0
- NO = number orders placed; initially 0

**Event Graph**
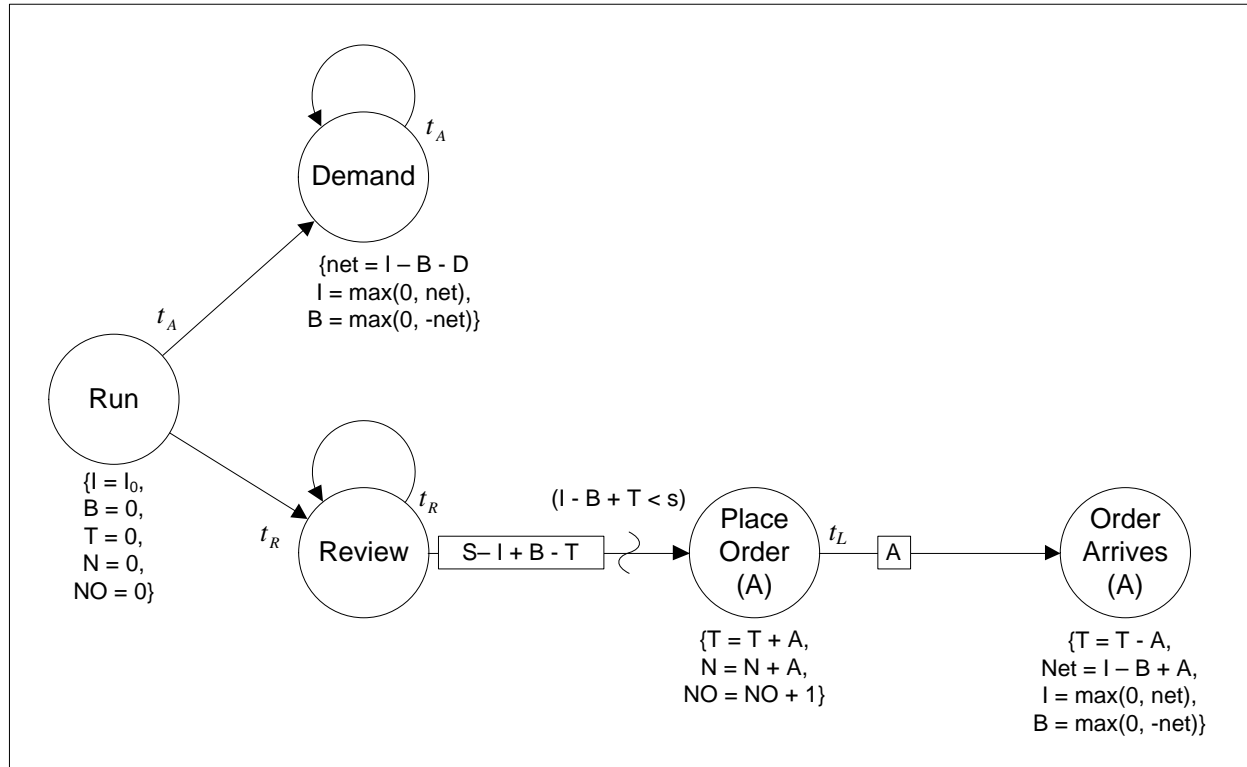


**Figure 4-6. Event Graph for Inventory Model**

Note that the only effect of a Demand is to update the net inventory position and then the two state variables, I (amount on hand) and B (amount backordered). Also, PlaceOrder is scheduled from Review with the condition $(I - B + T < s)$. The expression $I - B + T$ is the amount on-hand net the amount on backorder (owed to customers) and the amount on-order (owed to the facility). If T were omitted from the expression and a Review occurred before an outstanding order had arrived, then another order would be placed, which is not desired. The expression $S - I + B - T$ has the effect of making the inventory level at that instant, net everything that is owed, exactly equal to S. The use of a parameter on the scheduling edge and on the OrderArrives(A) Event is because the order quantities will typically vary from one order to the next. When an order does arrive, the amount on order (T) is decremented by the amount that has arrived (A), then nets out the inventory position variables in the same way as the Demand event. The state variables for the number of items ordered and number of orders placed are updated at the PlaceOrder(A) event.

## 4.4.3. Transfer Line

Jobs arrive one at a time according to n arrival process and are processed by n workstations in a series, each consisting of a multiple-server queue with infinite capacity. Upon completion of service at each workstation, a job proceeds to the next workstations and departs the system when service at the last workstation is complete.

**Parameters**

- $\{t_A\}$ = times between arrival of jobs
- n = # workstations
- $k_i$ = # machines at workstation $i$  $i = 0, \ldots, n-1$
- $\{t_{S_i}\}$ = service time at workstation $i$  $i = 0, \ldots, n-1$

**States**

- $Q_i$ = # in queue at workstation $i$  $i = 0, \ldots, n-1$
- $S_i$ = # available machines at workstation $i$  $i = 0, \ldots, n-1$

**Event Graph**
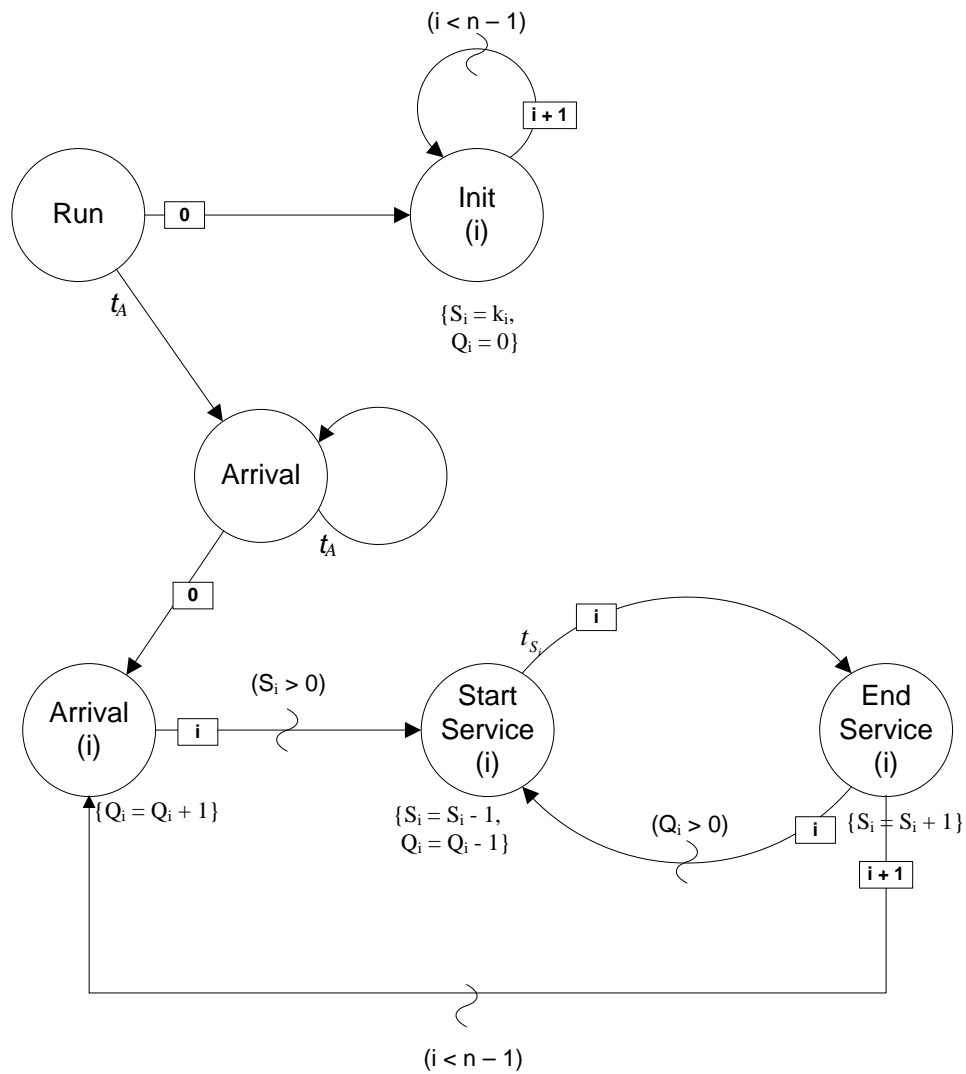


**Figure 4-7. Transfer Line Event Graph**

Note how the "for" loop is used to initialize $Q_i$ and $S_i$.  In general, whenever there are state variables that are arrays, a similar construct must be used to initialize their values.  Notice

4-9

how the basic structure of the model is very similar to the multiple server queue model above. By creating the model in this manner, with the number of workstations given as a parameter, the same model structure can be used regardless of the number of workstations in the system.

## 4.4.4. Multiple Server Queue with Impatient Customers

This model illustrates the use of a cancelling edge as well as passing arguments on edges. Consider a queueing system like the multiple server queue in which customers are "impatient" and will exit the queue if they are not served within a certain amount of time. A customer who joins a queue and subsequently leaves is said to have "reneged." Each customer's impatience time is from a given probability distribution. Additional measures include the percentage of customers who renege.

A simple approach to modeling this situation is to add state variables, on that counts the total number of arriving customers (N) and one that counts the number who have reneged. An event is needed that corresponds to the reneging of a customer. The state transition for a Renege is to decrement the number in the queue and increment the number of reneges that have occurred.

**Parameters**

- $k$ is the total number of servers in the system
- is the sequence of (possibly random) times between the arrival of customers to the system.
- $\{t_S\}$ = the sequence of (possibly random) service times of each successive customer.
- $\{t_R\}$ = the sequence of (possibly random) renege times

**States**

- q = FIFO container of (unique) customer id's (initial value = empty).
- S = the total number of available servers (between 0 and $k$). The initial value is $k$.
- N = the total number of customers who have arrived to the system (initially 0)
- R = the total number of customers who have reneged (initially 0).
- M = the total number of customers who have received service (initially 0)

**Event Graph**
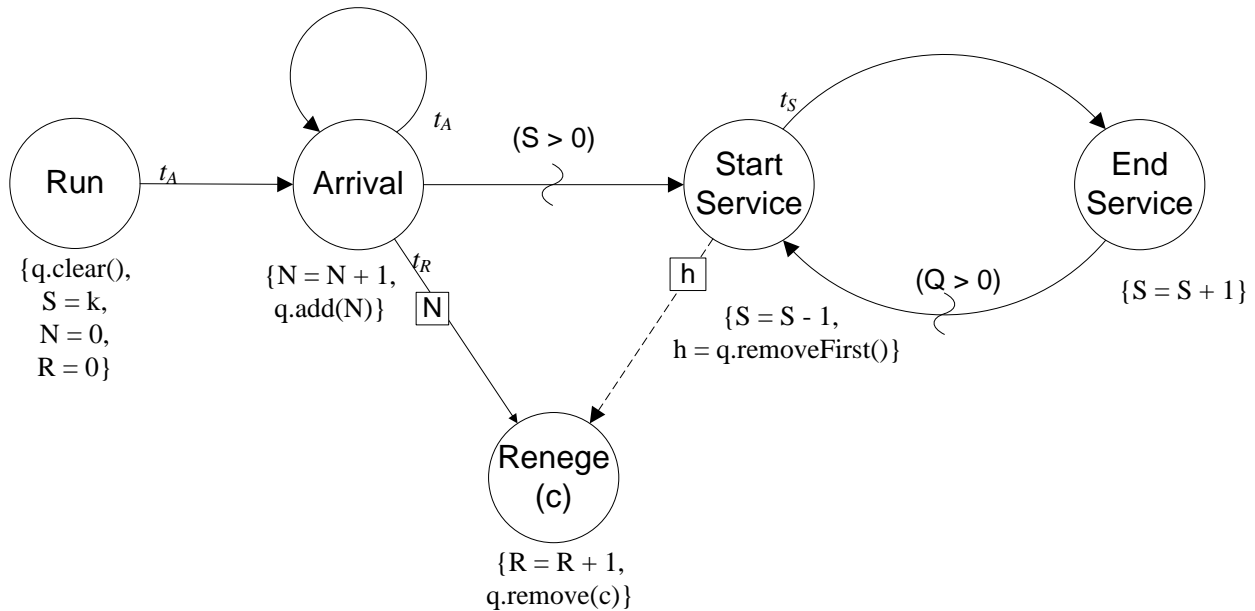


**Figure 4-8. Multiple Server Queue with Customers who Renege**

As shown by the Event Graph in Figure 4-8, an arriving customer schedules the Renege event upon arrival, so that whenever it occurs, the queue is decremented. If the StartService event occurs first, however, then the Renege event that corresponds to that customer is removed. Note that the expression $M + R$ that is on the cancelling edge from StartService to Renege gives the number of the customer who has just started service.

The state variables M, R, and N can be used to compute the proportion of customers who renege (R/N) and the proportion of customers who receive service (M/N). The time-varying averages of S and Q can be used to estimate the average number in the queue and the average utilization of the servers.

Note that Little's formula cannot be applied in this situation to estimate the delays in queue or the time in the system for customers who received service. This is because the queue count includes all customers and can't distinguish (before a Renege occurs) between those who will eventually renege and those who will receive service. Applying Little's formula would require two state variables, one for served customers and one for reneging customers. Since this cannot be done until after the fact, a model that explicitly computes these times is needed (see below).

## 4.5.   Containers

For certain situations it is convenient to model using containers to hold data. In Event Graph models a container can be thought of as a state variable with "values" that go beyond simple numerical variables.

### 4.5.1. Multiple Server Queue with Explicit Tally of Times in Queue and System

Suppose we wish to explicitly tally the delay in the queue and the time in the system for the multiple server queue model described above. As noted, this cannot be done explicitly by the previous model (although Little's formula can be used, as also discussed).

Instead of only keeping track of the number of customers in the queue, a first-in first-out container can be used to keep track of the respective arrival times of the customers. A global variable, which we will call simTime, is maintained by the Event List, as described earlier. The value of simTime when an Arrival Event occurs is the time when the corresponding customer arrived to the system, and the value of simTime when StartService occurs is the time when a customer started service. If the value of simTime when the corresponding customer arrived were known, the delay in queue could simply be computed by the difference. Similarly, the difference between the simTime when EndService occurs and when the corresponding Arrival occurred is the time in the system.

**Parameters**

- $k$ is the total number of servers in the system
- is the sequence of (possibly random) times between the arrival of customers to the system.
- $\{t_S\}$ is the sequence of (possibly random) service times of each successive customer.

**States**

- q is a fifo container holding the arrival times of each respective customer in the queue. Initially it is empty.
- S is the total number of available servers (between 0 and $k$). The initial value is $k$
- D = delay in queue for the last customer (initially undefined)
- W = time in the system for the last customer (initially undefined).

**Event Graph**
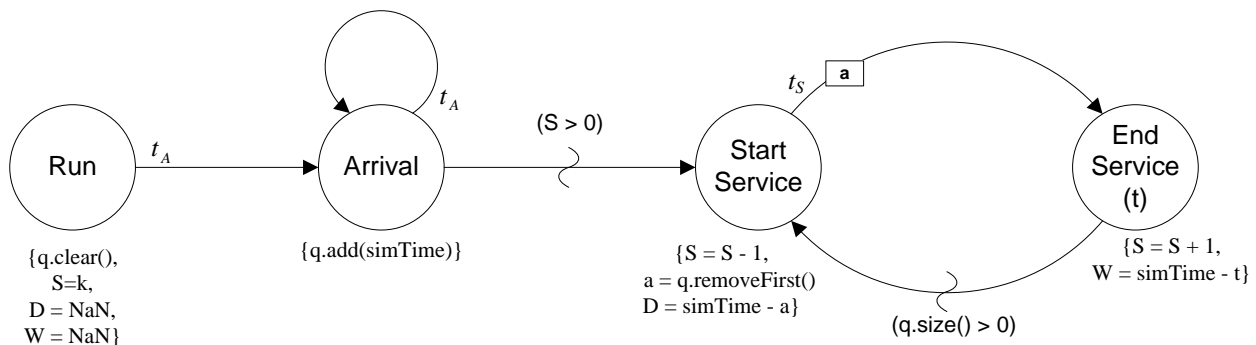


**Figure 4-9. Multiple Server Queue with Explicit Tally**

Note that two mechanisms are available for "remembering" the simTime of arriving customers: storing in a container and passing as an argument. Both are used in the model, as shown in Figure 4-9.

For the two state variables D and W, note that they are simply assigned values. Statistics for these variables will be collected in a tally manner, unlike S that is a time-varying variable. Note also that their initial values are 'NaN' which stands for "Not a Number." This is because at the start of the simulation there is no "last" customer of either type. As the simulation is running, there may be more observations of D at any given point in time than of W.

To estimate the average number in queue for this model it is necessary to collect a time-average on the *number* of items in q rather than the value itself, as in the basic model.

## 4.5.2. Multiple Server Queue with Impatient Customers – Explicit Tally of Delay in Queue and Time in System

In the queueing model with impatient customers, it was not possible to estimate the average delay in queue and average time in the system. One way to do this is to save the simTime in the container representing the queue instead of the index of the customers and to also pass that value to the Renege event. StartService will remove the first element from the container, which is the time that customer arrived. The corresponding Renege event for that time will be cancelled and the delay in queue calculated. The removed value will be passed to the EndService event for computing the time in the system. The Renege event itself will remove the corresponding time from the queue and compute the delay in queue for that customer. Thus, it will be possible to separately estimate the average delay in queue for customers who received service and for those who eventually reneged.

### Parameters

- $k$ is the total number of servers in the system
- is the sequence of (possibly random) times between the arrival of customers to the system.
- $\{t_S\}$ is the sequence of (possibly random) service times of each successive customer.
- $\{t_R\}$ is the sequence of (possibly random) renege times

### States

- q is fifo container of the arrival times; initially it is empty.
- S is the total number of available servers (between 0 and $k$). The initial value is $k$.
- N is the total number of customers who have arrived to the system (initially 0)
- R is the total number of customers who have reneged (initially 0).
- W is the time in the system for customers who are served(initially NaN)
- $D_S$ is the delay in queue for customers who are served (initially NaN)
- $D_R$ is the delay in queue for customers who are served (initially NaN)

**Event Graph**
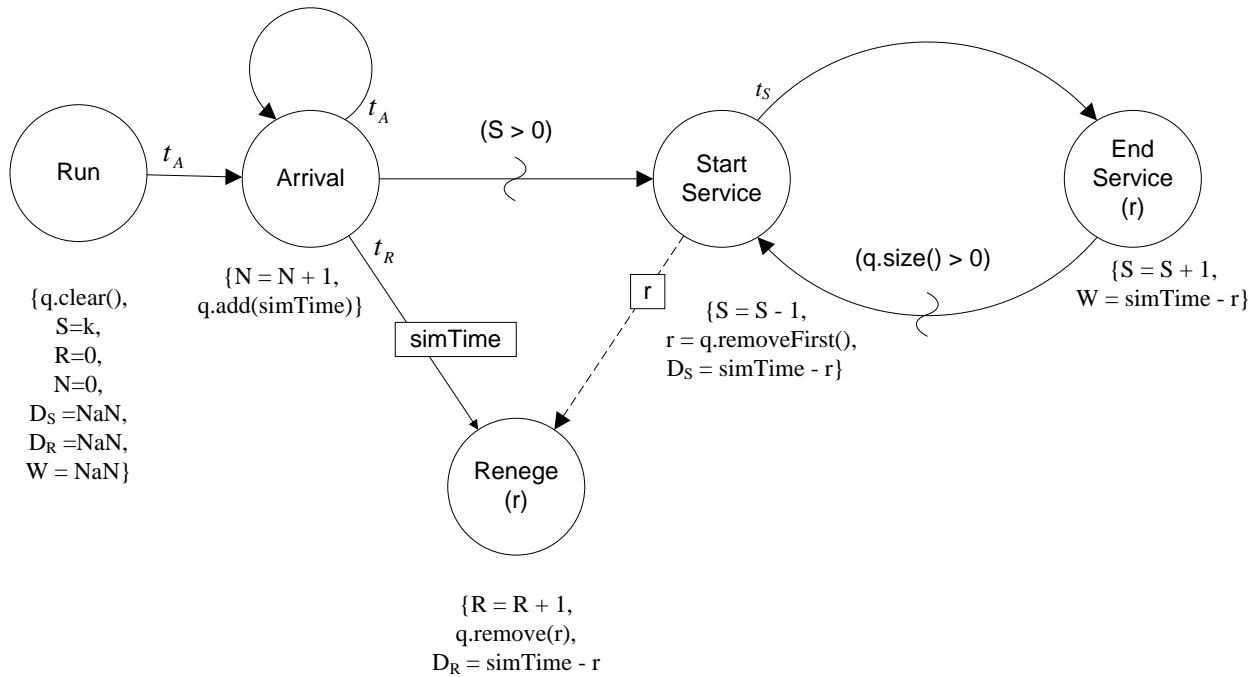


**Figure 4-10. Multiple Server Queue with Customers who Renege**

As with the previous model with impatient customers, the proportion of customers who renege can be computed as R/N. Note that the number of customers who have received service (or are receiving service) can be computed as $N - R - q.size()$.

## 4.6.    The Next Step

Event Graphs are an intuitive and powerful way to conceptualize DES models as well as a methodology for creating them. As models become more complex, the number of Events increases, and with it the number of nodes that are in the corresponding Event Graph. At some point, the number of Events can be come so large that the advantages of Event Graphs begin to diminish. In other words, there is a limit in the size of models that Event Graphs can continue to be an effective modeling tool. Although it is quite possible to build large models using only the Event Graph concepts that have been presented so far, it turns out that there is a way to use Event Graph methodology to create modular simulation components. Each component can be small and manageable, and these components can be connected to build much larger-scale models.

Thus, everything that has been presented so far can be utilized in a simulation component framework that is flexible, extensible, and scaleable

# 5.    Event Graph Components

Event Graph methodology as described in the previous section is an intuitive and effective way to design Discrete Event Simulation (DES) models.  However, as noted previously, if the entire model consists of a single Event Graph, large models – those with many Events - become difficult to understand and maintain.

One way that has been found to mitigate this problem is by defining Event Graph components.  An Event Graph component is simply an Event Graph "in miniature" – that is, an object that has its own parameters, state variables, and Events.  An Event Graph component is solely responsible for maintaining its own state variables, and is not at all responsible for the maintenance of any other component's state variables.

## 5.1.    Event Graph Components

An Event Graph component is simply a component whose description is given by an Event Graph model.  The description is a template for instances of the component, much like a class is a template for an object in object oriented programming. Thus, each Event Graph component encapsulates its own copies of parameters and of state variables.  However, all components share a common Event List.

Thus, each Event Graph component will have its own values of parameters that will stay fixed throughout a simulation run.  Each component's state variables will star the run at given initial conditions, defined by the component, and as the run unfolds, have its own state transitions that apply only to its state variables.  Even if two components are instances of the same Event Graph, they may have different parameters and the values of their state variables at any point in time depend only on those initial conditions and the Events that have occurred in that component only.

Since all components share a common Event List, the Event List needs to be able to keep track of which Event was scheduled by which component.  However this is done, whenever an Event occurs, the state transition for that Event is performed by the component that scheduled the Event.  That is, the state transition is applied to the values of the state variables in the component that scheduled the Event only – no other components are *directly* affected by the Event's occurrence.

However, for a DES model to have interesting and useful behavior, the components do need to have some kind of interaction.  That interaction is provided by the SimEventListener pattern and its relative the Adapter pattern, which are now discussed.

## 5.2.    SimEventListener Pattern

The SimEventListener pattern is the primary mechanism by which Events in one simulation component can affect the state of another.  A key element (and advantage) of simulation components is the fact that only Events for that component can make changes to its state. Thus, the only way for an external component to effect a change in state is for it to somehow cause an Event to be executed.

SimEventListening works as follows.  One simulation component shows "interest" in another's Events by explicitly being registered as a SimEventListener to it.  This relationship is shown in Figure 5-1.
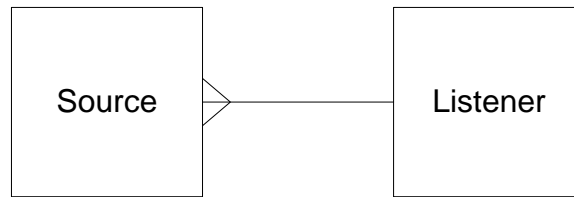
**Figure 5-1. SimEventListener Relationship: Component Listener "Hears" all of Component Source's Events**

If there is a listener relationship as in Figure 5-1, then whenever an Event from Source occurs, then after it has executed its state transitions and scheduled Events, the Event is sent to Listener. If Listener has an Event that is identical (in both name and signature) to the one it "hears" then it processes that Event as if it had scheduled it. That is, whatever state transitions and scheduling are defined for the listening component are executed. The only difference is that the listening component does *not* re-dispatch heard Events to its listeners, if it has any.

The one exception to SimEventListener is the Run Event, which is not dispatched to SimEventListeners when it is processed. That is because each component is expected to have its own Run Event that is processed once (and only once) per simulation run. Thus there is no need for it to be heard by any other components, since theirs will be processed anyway.

There is no theoretical limit to how many Listeners are connected to a given Event Graph component, nor is there a theoretical limit to how many Event Graph components a given one can listen to itself. The only limitations have to do with implementation, since each connection takes up space in a finite computer. Since a reasonable implementation involves very little space, and since computer memory is currently large and increasing, even this implementation constraint is not likely to be an issue.

Because listening is the key manner in which components interact with each other, this has been called the "LEGO Component Framework," where LEGO stands for **L**istener **E**vent **G**raph **O**bjects.

## 5.2.1. Examples

**ArrivalProcess**

The Arrival Process Event Graph can be thought of as a component as well as a stand-alone DES model. As a stand-alone model, it is not very interesting. However, the same pattern of a self-scheduling Event with a random delay has been observed in a number of DES models. Viewing the ArrivalProcess as a component simply means that whenever this functionality is needed in a model, henceforth the approach will be to instantiate an ArrivalProcess and have other components listen to it.

The ArrivalProcess Event Graph as a component is shown in Figure 5-2. The only difference appears to be the fact that it now has a box drawn around it.
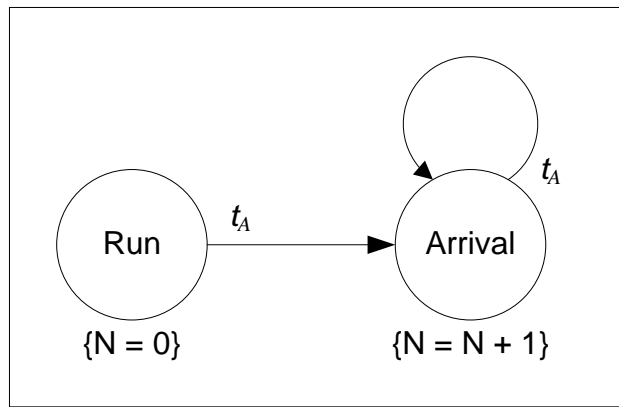
**Figure 5-2. Arrival Process as Event Graph Component**

## SimpleServer Component

The model of the multiple server queue described previously is one example of where there was an "embedded" ArrivalProcess in an otherwise monolithic model. With SimEventListening at our disposal, that aspect can be removed from the model, and what is left is just the portion that models the server logic. Figure 5-3 shows what the resulting Event Graph component looks like when this is done.
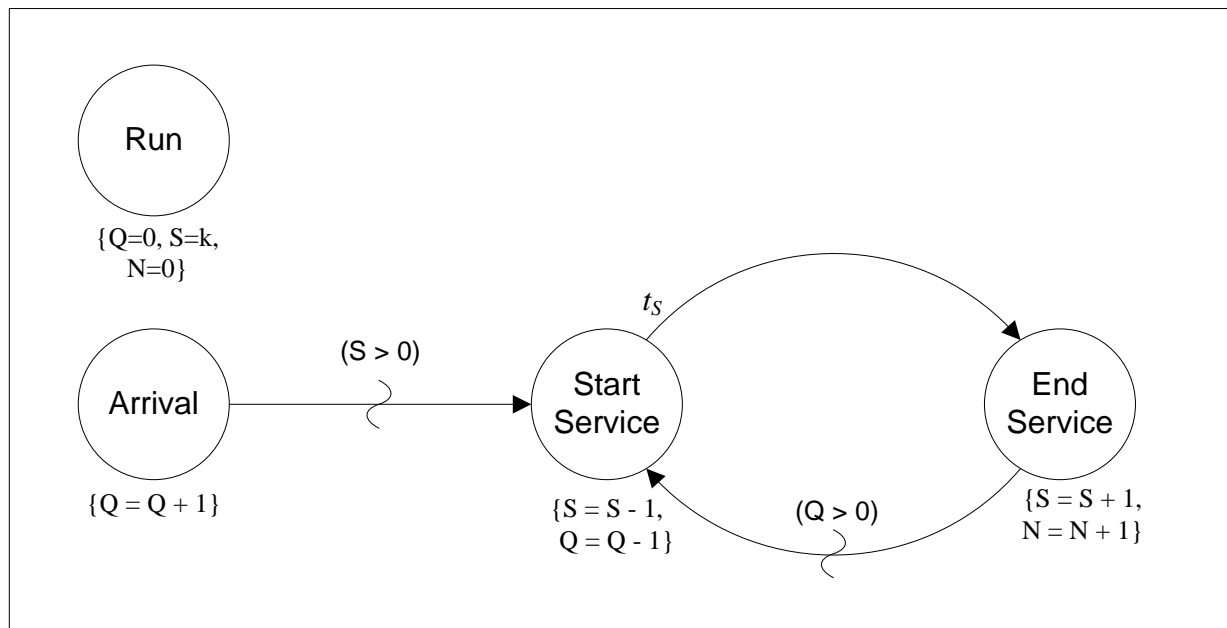


**Figure 5-3. SimpleServer Component**

Notice in Figure 5-3 that the Run event does not schedule any Events, but simply initializes the three state variables. This means that the SimpleServer component is not a complete model, and cannot produce anything meaningful by itself. It relies on some external Event Graph component to provide the Arrival Events that will cause its state transitions and other Events to be executed.

One way to do this is to have an ArrivalProcess that generates the Arrival events and have an instance of the SimpleServer component listen to it. This is shown in Figure 5-4.
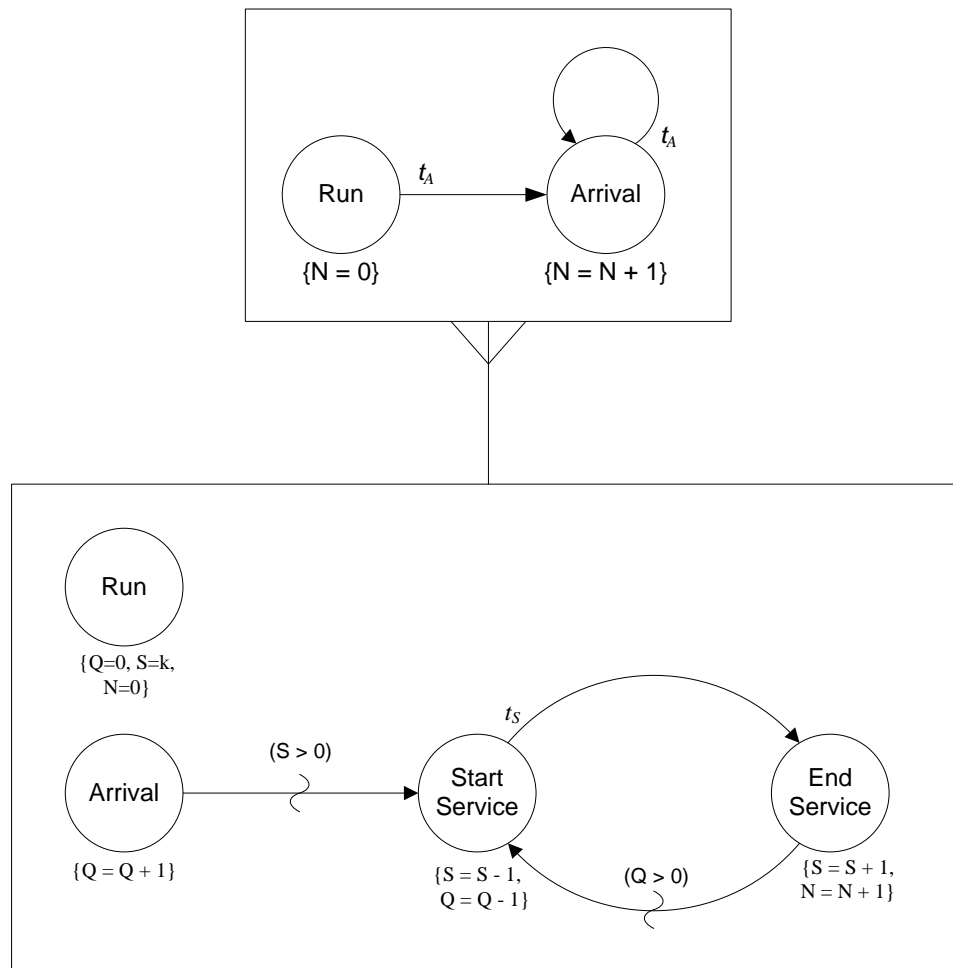


**Figure 5-4. Multiple Server Queue with Components**

**Tandem Queue**

Recall that to model the tandem queue in a monolithic manner required duplicating definitions of state variables and Events. A more parsimonious approach utilizes the fact that the primary functionality in the SimpleServer component is simply duplicated, and so using two instances of it should suffice.
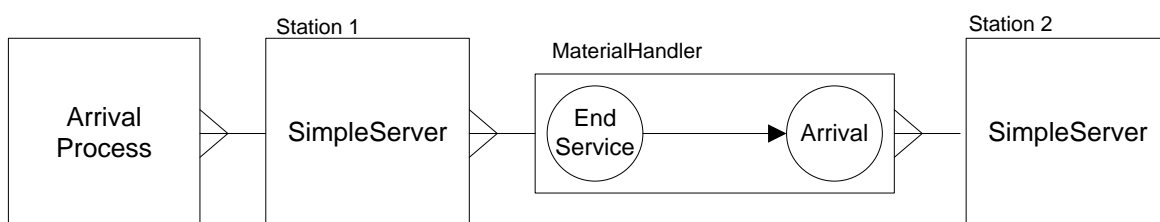


**Figure 5-5. Creating a Tandem Queue with Components**

The resulting model is shown in Figure 5-5.  Note that an additional Event Graph component has been inserted between the two SimpleServer instances.

This is necessary because to have a SimpleServer component listen to another would cause extraneous and undesired Events to be processed in the listener.  For example, in Figure 5-5, each StartService Event in the Station 1 SimpleServer would cause a StartService Event to also occur in the Station 2 SimpleServer.  Likewise, an EndService Event in Station 1 would cause an EndService in Station 2.[3]  This is highly undesirable because the start of service in the first station does not trigger the start of service in the second!

The dynamics occurring in Figure 5-5 are that when an EndService Event occurs in Station 1, it is heard by the "Material Handler" Event Graph in Figure 5-5, because it has an EndService Event as well.  That in turn schedules an Arrival Event with zero delay.  When that Arrival Event occurs, it is then heard by Station 2, which processes it as usual.

Although the MaterialHandler component will "hear" StartService events from Station 1, it will not respond to them, since it has no StartService event itself.  The MaterialHandler component is extremely lightweight – it has no parameters or state variables, and only serves to translate EndService events in Station 1 to Arrival events in Station 2.

The general pattern is that sometimes when an event occurs in one component (EndService in Station 1, in this case), the modeler wishes an event of a different name to occur in another component (Arrival in Station 2, in this example).  Since this is a highly desirable functionality, it is incorporated into the component framework as an *Adapter Pattern*, which is now discussed.

## 5.3.   Adapter Pattern

It frequently arises that there is a desire for an Event of one name in a component to cause another Event of a different name to occur in another component.  There was a specific example of this in the tandem queue of the previous section, in which the EndService Event in the first server component was to cause an Arrival Event in the second.  In general, if a "source" component has an Event A and it is desired to cause Event B in a Listener component whenever A occurs, then an adapter between the Source and Listener is created that "adapts" Event A to Event B.  This is illustrated in Figure 5-6.
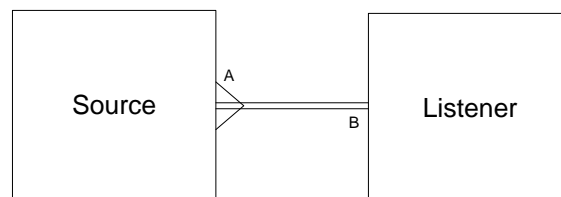


**Figure 5-6. Prototype Adapter: Event A in Source Causes Event B in Listener**

Unlike the Listener pattern, the Adapter works on a single Event only.  If a component is a SimEventListener to another, then *all* of the Events that are scheduled by the source component are heard by the Listener when they occur (with the exception of Run, as mentioned previously). If the Adapter is used, then only the Event specified by the adapter is heard (as the adapted Event).  That is, in Figure 5-6, only when Event A occurs in the Source component does the Listener "hear" anything.  If there is a desire to hear other Events, then additional Adapters must

---

[3] However, an Arrival Event will not be heard, because it itself has been heard from the ArrivalProcess.

be created to connect the components.  Note that both Events can have the same name – this would be the case if the Listener *only* wanted to hear a specific Event that it also had.

The Source Event and the Adapted Event must have identical signatures (parameter list) for the Adapter to actually work.  If there is a mismatch in signatures, then nothing will happen.  For example, in Figure 5-7 Event A is adapted to Event B just as in Figure 5-6, and this is a perfectly legal construct.  However, the Listener in Figure 5-7 will be expecting an Event that matches B(k), and it will hear an Event B with no parameters. Therefore, since there is no "match," the Listener in this case will do nothing.
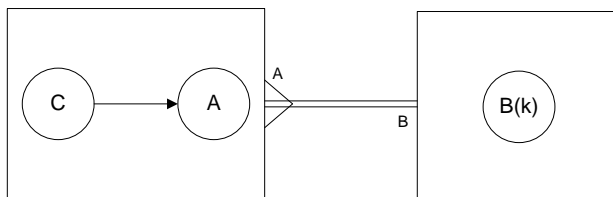


**Figure 5-7. Legal but Useless Adapter**

However, if instead the adapted Event does have the same signature, then it will actually cause the desired result.  Figure 5-8 shows an Event A(j) being scheduled in the Source component and being adapted to the B(k) Event in the Listener.
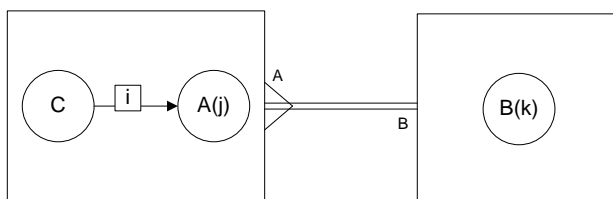


**Figure 5-8. Legal and Useful Adapter with Arguments**

Finally, although it is typically very dangerous to have two Event Graph components listening to each other, it is safer (and more common) to have them listen with adapters, especially if different Events are involved.  This is illustrated in Figure 5-9.
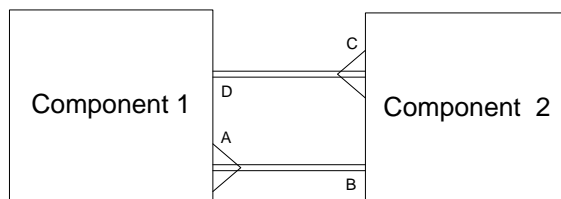


**Figure 5-9. Two Event Graph Components Listening to Each Other via Adapters**

In Figure 5-9, Event A in Component 1 is adapted to Event B in Component 2 and Event C in Component 2 is adapted to Event D in Component 1.  As with SimEventListeners, there is no theoretical limit to how many Adapters can be connected between Event Graph components.

### 5.3.1. Examples

**Tandem Queue with Adapter**

Instances of the simple server model of the multiple server queue can be strung together using the Adapter pattern. Instead of Figure 5-5, the EndService Event in Station 1 can be adapted to Arrival by an Adapter, as shown in Figure 5-10.
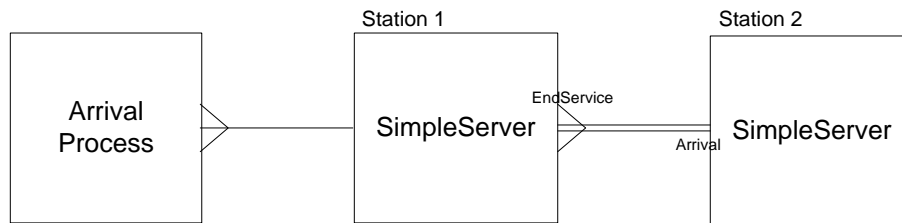


**Figure 5-10. Tandem Queue Using an Adapter**

Note that the connection between the Arrival Process and Station 1 in Figure 5-10 could be either the Listener, as shown, or the Arrival Event in the Arrival Process could be adapted to the Arrival Event in Station 1.

**Transfer Line as Component**

The transfer line described previously can be modeled as a component, so that the manner of arrivals to the system is decoupled from an Arrival Process. This is shown in Figure 5-11.
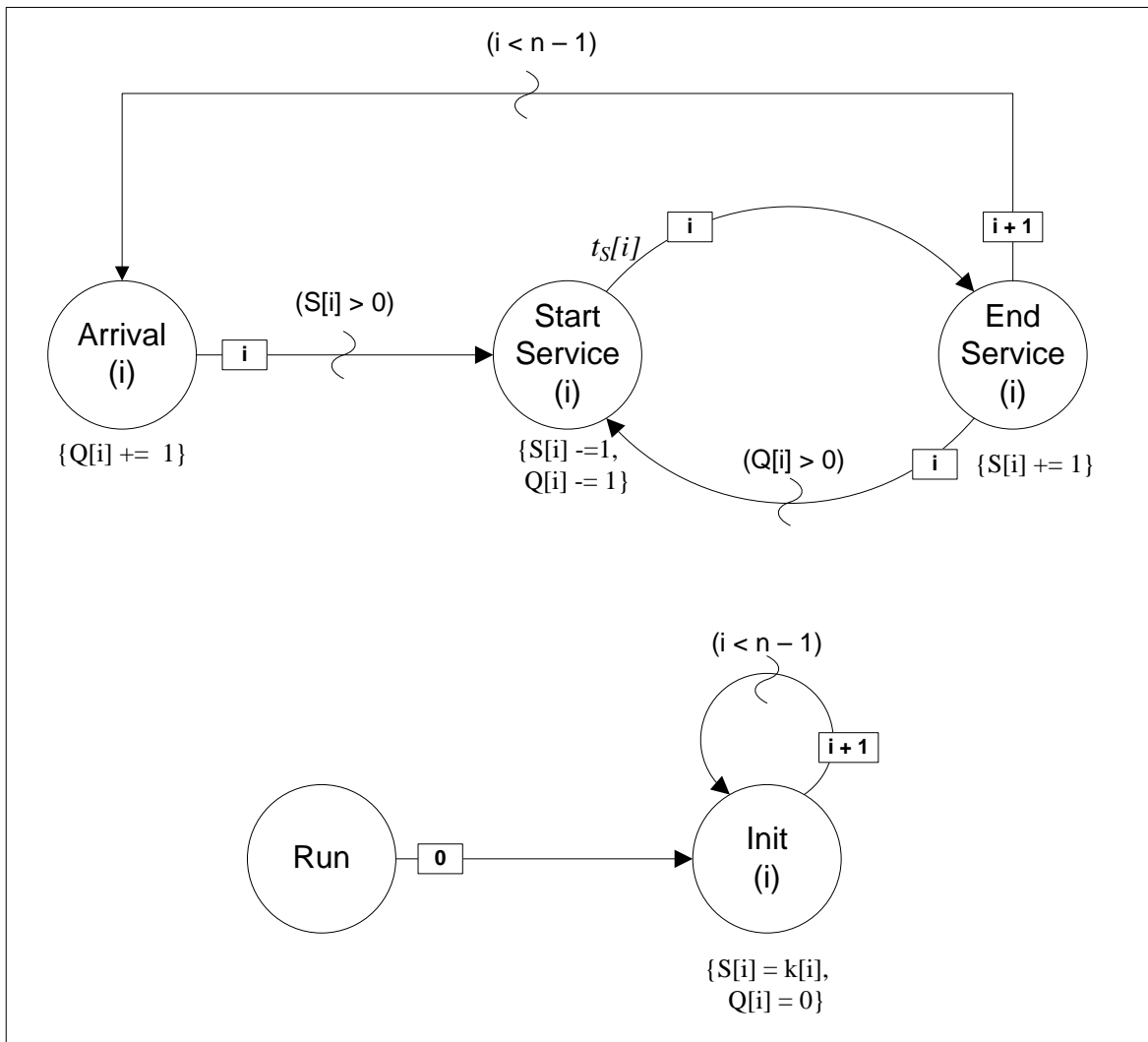
**Figure 5-11. Transfer Line Component**

As with the simple server component in Figure 5-3, the transfer line component in Figure 5-11 is not a stand-alone model, but must be connected to other component (or components) that will cause the Arrival(i) Event to occur. One example of this is shown in Figure 5-12.
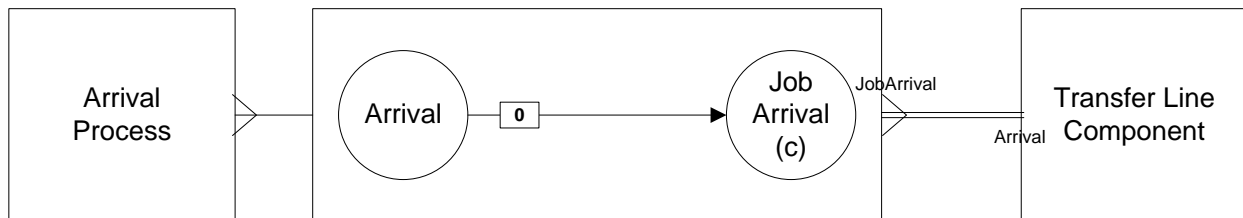


**Figure 5-12. Transfer Line Model Component Connected to Arrival Process**

The model in Figure 5-12 shows an Arrival Process being the source of arrivals to the transfer line system. From Figure 5-11, it is the Arrival(i) Event with argument of 0 that is needed to be heard. The arrival process cannot directly do this because the signature for the Arrival Event in the arrival process doesn't match the Arrival(i) Event in the transfer line component. This, in Figure 5-12 a third component is added to do this. That component simply

schedules a JobArrival(c) Event when it hears an Arrival Event and passes the number 0 as an argument.  The adapter connects the JobArrival(c) Event to the Arrival(i) Event in the transfer line component.

Although the middle component in Figure 5-12 bears a superficial resemblance to the material handler component in Figure 5-5 that motivated the Adapter, the key difference is that the signatures of the two Events are different for the component in Figure 5-12.  Although it might be possible to develop a generic implementation of this capability, that is stimulating Events in other components when the signatures don't match, it is not at all clear how that might be done.  For now, therefore, the modeler must be content with writing small adapter-like components like the one in Figure 5-12 when there is the need to change signatures of Events.

## 5.4.    Property Change Listener Pattern

A second listener pattern is available in the LEGO framework is the Property Change Listener pattern.  The idea is simply that whenever a state variable changes value, that information is dispatched in a Property Change Event.  A "PropertyChangeEvent" is completely different than a SimEvent; for example, a PropertyChangeEvent never interacts with the Event List, whereas a SimEvent does.

Special components called PropertyChangeListeners are created to listen to the PropertyChangeEvents that are fired by the LEGO components. These components are structurally considerably simpler than LEGO components.  They do not have any Events, nor do they interact in any way with the Event List, as already mentioned.  Their primary purpose is to do something in response to state transitions in certain designated LEGO components.

A PropertyChangeListener relationship is depicted in Figure 5-13.  The connector is similar to that of SimEventListener, except it is depicted with a small pitchfork-like end.
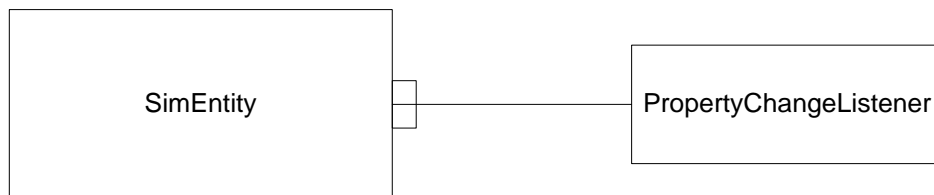


**Figure 5-13. PropertyChangeListener**

A PropertyChangeListener as shown in Figure 5-13 will "hear" all of the state transitions for the SimEntity component.  The information included is the name of the property (state), the old value (the value of the state variable prior to the state transition) and the new value (the value of the state variable after the state transition).  It is expected that every state variable that changes fire a separate PropertyChangeEvent.  If a PropertyChangeListener is designed to only hear one state variable at a time (as is the case with many of the statistics listeners), it is the listener's responsibility to filter out the "unwanted" PropertyChangeEvents.

This can also be mitigated by an alternate form of the PropertyChangeListener connection, so that the connection itself can do the filtering.  This is illustrated in Figure 5-14.
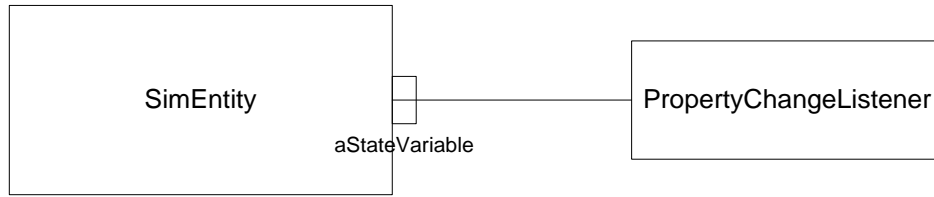
**Figure 5-14. PropertyChangeListener for Specific Property**

Here, the PropertyChangeListener component will only hear PropertyChangeEvents for the state

variable called "aStateVariable."

Just as with SimEventListeners, there is no theoretical limitation on how many PropertyChangeListeners can listen to a given LEGO, nor is there a theoretical limitation on how many LEGO components a given PropertyChangeListener can listen to.

Note that while LEGO components are both sources and listeners of SimEvents, a PropertyChangeListener is typically *not* a SimEntity. Thus there is an asymmetry between sources of PropertyChangeEvents and their listeners.

Two of the primary functions performed include a useful way of debugging and troubleshooting Event Graph components and collecting statistics. The first function is associated with a computer implementation of Event Graphs, and so is not relevant in a conceptual context. Two simple examples of the latter function will now be described.

### 5.4.1.  Examples

As previously discussed, there are two ways that statistics are collected in Discrete Event Simulation models: time-varying and tally. Recall that time-varying means are computed as the area under the state trajectory curve divided by the time, whereas a tally mean is the sum of discrete values divided by the number of the values observed. Since the formulas for computing these are fundamentally different, two distinct PropertyChangeListener components are created. The idea is that these be as lightweight as possible; specifically, only a minimal amount of data is stored, which typically will include counters and accumulators for the sum and sum of squares.

**Tally Statistics**

A tally mean is the numerical average of a discrete collection of observations. For discrete observations, the variance is likewise defined, as in Figure 5-15.

$$\overline{X} = \frac{1}{n} \sum_{i=1}^{n} X_i$$

$$\overline{S}^2 = \frac{1}{n-1} \sum_{i=1}^{n} \left( X_i - \overline{X} \right)^2$$

**Figure 5-15. Definition of Tally Mean and Variance for Discrete Observations**

One approach to estimate simple summary statistics for a simulation is to maintain a counter of the number of observations and a running sum and sum of squares. The minimum and maximum observations are easily kept as well. We will call this listener a *SimpleStatsTally*. It

consists of a counter, n, which contains the number of observations as well as running values of the mean, variance, min, and maximum values. Figure 5-16 shows the initialization of these values.

$$n = 0$$
$$\overline{X} = 0.0$$
$$\overline{S}^2 = 0.0$$
$$\text{Min} = \infty$$
$$\text{Max} = -\infty$$

**Figure 5-16. Initialization for SimpleStatsTally**

Now, whenever a state transition is heard, the counters are updated according to the logic in Figure 5-17.[4]

$$n = n + 1$$
$$\Delta = X_n - \overline{X}$$
$$\overline{X} = \overline{X} + \frac{\Delta}{n}$$
$$\overline{S}^2 = \begin{cases} 0.0 & , \quad n = 1 \\ \overline{S}^2 + \frac{\Delta^2}{n} - \frac{\overline{S}^2}{n-1} & , \quad n > 1 \end{cases}$$
$$\text{Min} = \text{Min} \wedge X$$
$$\text{Max} = \text{Max} \vee X$$

**Figure 5-17. Updating SimpleStatsTally Variables**

The reader may wonder why the expressions in Figure 5-17 aren't the obvious ones (i.e. maintaining running sum and sum-of-squares). The reason is that the expressions for $\overline{X}$ and $\overline{S}^2$ in Figure 5-17 will tend to stay closer to their respective values, whereas the running sum, and especially the running sum of squares, could easily overflow the possible values on the computer as more and more data are collected. The reader should verify that the expressions in Figure 5-17 yield the correct values.

Note that since simulation data tend to be correlated, interpreting the variance by $\overline{S}^2$ using this approach should be done with great caution. Specifically, this value should generally not be used when creating confidence intervals or conducting hypothesis tests.

**Time Varying Statistics**

Time varying statistics can be collected in a similar manner, by updating running values of the min, max, mean, and variance. Instead of the count, the time of the last state change, $t_{last}$ as well as the running value of $\Delta$ must be stored as well. Recall that the definitions of time-varying mean and variance are as in Figure 5-18.

---

[4] Recall that $a \wedge b$ is the minimum value of $a$ and $b$ and that $a \vee b$ is the maximum value of $a$ and $b$.

$$\overline{X}_T = \frac{1}{T}\int_0^T X_t\,dt$$

$$\overline{S}_T^2 = \frac{1}{T}\int_0^T \left(X_t - \overline{X}_T\right)^2 dt$$

**Figure 5-18. Definitions of Time-Varying Mean and Variance**

The PropertyChangeListener that implements collecting these statistics is called *SimpleStatsTimeVarying*. Figure 5-19 shows the initialization of the counters and accumulators for SimpleStatsTimeVarying.

$$t_{last} = 0.0$$
$$\overline{X}_{0.0} = 0.0$$
$$\overline{S}_{0.0}^2 = 0.0$$
$$\text{Min} = \infty$$
$$\text{Max} = -\infty$$
$$\Delta = X_{0.0}$$

**Figure 5-19. Initialization of SimpleStatsTimeVarying**

Updating the values is a bit more involved, as shown in Figure 5-20. Here $X_{\text{simTime}}$ is the new value of the state variable after it has made its transition. For clarity, '$T$' has been replaced with 'simTime' in the expressions to emphasize the fact that the updates are being performed at the current value of the simulation clock.

$$\overline{X}_{\text{simTime}} = \begin{cases} 0.0 & , \quad \text{simTime} = 0.0 \\ \overline{X}_{t_{last}} + \left(1 - \dfrac{t_{last}}{\text{simTime}}\right)\Delta & , \quad \text{simTime} > 0.0 \end{cases}$$

$$\overline{S}_{\text{simTime}}^2 = \begin{cases} 0.0 & , \quad \text{simTime} = 0.0 \\ \overline{S}_{t_{last}}^2 + \left(1 - \dfrac{t_{last}}{\text{simTime}}\right)\left(\dfrac{t_{last}}{\text{simTime}}\Delta^2 - \overline{S}_{t_{last}}^2\right) & , \quad \text{simTime} > 0.0 \end{cases}$$

$$\text{Min} = \text{Min} \wedge X_{\text{simTime}}$$
$$\text{Max} = \text{Max} \vee X_{\text{simTime}}$$
$$t_{last} = \text{simTime}$$
$$\Delta = X_{\text{simTime}} - \overline{X}_{\text{simTime}}$$

**Figure 5-20. Updating SimpleStatsTimeVarying**

Note in Figure 5-20 that the order is important; $\Delta$ is updated last, so that it is this value which is used the next time a new observation is heard. The reader should verify that the expressions in Figure 5-20 will yield the correct values for the mean and variance, respectively. Interestingly, these equations hold even if the value of the state variable hasn't changed.

**Other Statistics**

These two examples do not span the range of possibilities by a long shot. There are many other statistics and approaches that can be designed to take advantage of the PropertyChangeListener pattern.

## 5.4.2. Other Uses of PropertyChangeListener

The PropertyChangeListener pattern is very useful for other things than collecting statistics. For example, in a computer implementation, a PropertyChangeListener that simply writes the property name, old value, and new value to the console is extremely useful for debugging. Similarly, one that updates a graph of a state variable (or variables) and refreshed the window could provide a useful display of state trajectories as the simulation is running. The possibilities are limited only by the modeler's imagination.

## 6.    Transient Entities

By now it can be appreciated how using simple state variables can be used to create powerful and effectively DES models.  Since all models are abstractions, it is not necessary to explicitly model each and every element of a system.  For example, the SimpleServer model of a multiple server queue component does not explicitly capture each and every customer passing through the system.  Customers are implicitly represented only through the state variables, which simply keep a count of how many are in the queue or in service.  Not only is this representation sufficient for many purposes, Little's formula shows that certain average measures of the system can be indirectly obtained as well.

However, sometimes it is useful to explicitly model individuals in the system.  These "individuals" of course may be people, but could also be non-human things such as a job or an order.  We have seen an element of this when simulation times were stored in a first-in first-out (fifo) container so the delays in queue and times in system could be explicitly observed.  In that model the arrival time was all that was needed for the desired functionality.  However, there are often situations in which a single quantity is not sufficient to represent the data about an individual Entity we wish to model.  For example, customers or orders may be of different "types" and the service times may have different probability distributions depending on their types.  There may be different types of jobs, each of which has a different sequence of workstation types required to complete it.

To model individuals in this way, the idea of a *transient entity*, or simply *Entity*, is valuable.  An Entity can be thought of as an object to which certain *attributes* can be attached.  Indeed, an Entity can be compared to an object in object-oriented programming, and its attributes to fields associated with that object.

Since it cannot be determined beforehand what attributes a modeler requires of an Entity, it is important that new attributes be able to be added to an Entity at the modeler's discretion.

### 6.1.    Built-In Attributes and Features

A particular implementation of transient entities could have few or many built-in attributes and functions.  At a minimum there should be the ones listed in Table 3 below.

| Attributes | Methods |
|---|---|
| name | getName() |
| creationTime | getCreationTime(), getAge() |
| timeStamp | stampTime(), getTimeStamp(), getElapsedTIme() |

**Table 3. Attributes and Methods of the Basic Entity**

When an Entity is created, the simTime when that occurs is stored immutably.  The getAge() method computes the difference between the current simulation time and the Entity's creation time.  The, timeStamp attribute is set by a call to stampTime(), and getElapsedTime() computes the difference between the current simulation time and the value of timeStamp.  Since

the timeStamp attribute can be changed (unlike creationTime), it can be reused and reset to different values throughout a simulation run.

## 6.2. Examples

### 6.2.1. Entity Creator

The simplest way to create entities is through a small component that listens to an Event, instantiates (creates) the Entity and schedules another event with that Entity as the argument. **Error! Reference source not found.** shows the Event Graph for this pattern.
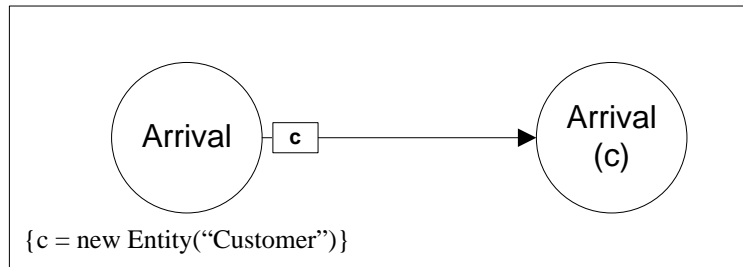
**Figure 6-1. Entity Creator Component**

In **Error! Reference source not found.**, the Entities' name has been set to "Customer."

### 6.2.2. Multiple Server Queue Component

The multiple server queue can be modeled using transient entities to represent the individual customers. A component to do that is shown by the EntityServer component in Figure 6-2. Note the similarity to previous models of servers.
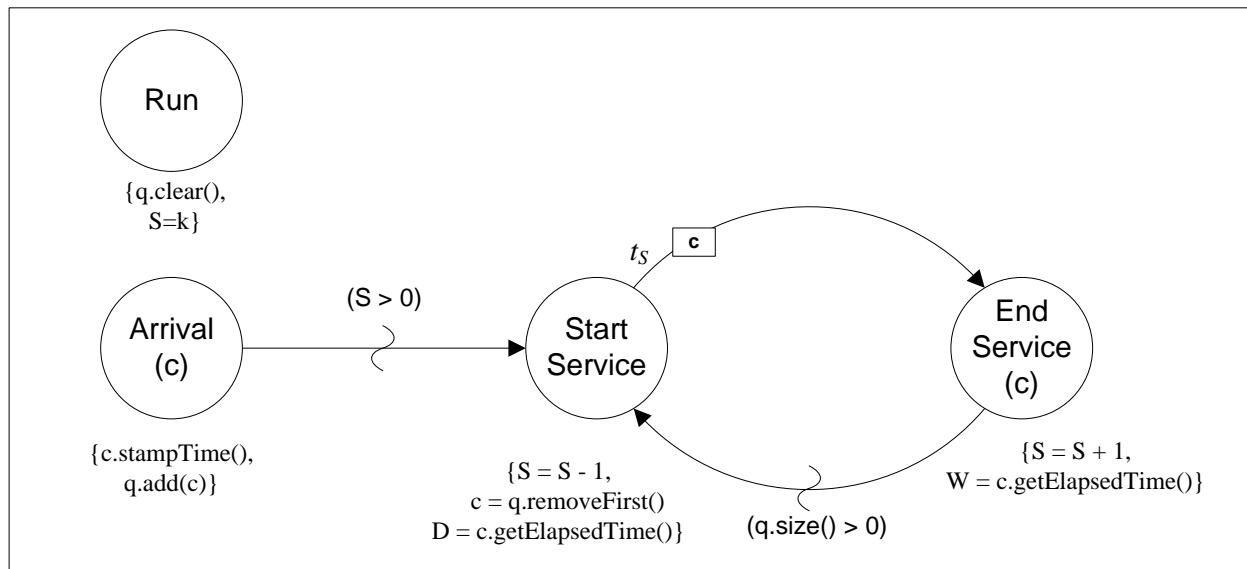
**Figure 6-2. EntityServer Component**

The calculation of delay in queue (D) and time in system (W) is simplified by the Entity's ability to capture the current value of simulation time in a timeStamp and retrieve the amount of elapsed time with a method call (getElapsedTime()).

The listeners for this situation would be connected as shown in Figure 6-3.

```
┌──────────┐        ┌──────────┐        ┌──────────┐
│ Arrival  │▷───────│ Customer │▷───────│ Customer │
│ Process  │        │ Creator  │        │  Server  │
└──────────┘        └──────────┘        └──────────┘
```

**Figure 6-3. Listeners for Customer Server Assembly**

Here, the Arrival event in the Arrival Process would be heard by the Customer Creator component, which in turn would schedule the Arrival(c) event in Customer Creator. This in turn would be heard by the Arrival(c) event in Customer Server component.

## 6.3.  Defining Additional Attributes

Additional attributes may be added to Entities by defining them as such. For example, instead of having the server generate the service time, it could be generated beforehand and added to each arriving Entity as an attribute. To implement this, define an attribute called serviceTime for the Customer Entity, as shown in Table 3.

| Entity | Attribute | Type |
|--------|-----------|------|
| Customer | serviceTime | double |

**Table 4. Customer Entity with `serviceTime` Attribute**

This Customer Entity would also include all the basic attributes described in Table 3, so the additional attributes are added to the pre-existing ones.

Customer Entities could be created using a modified Creator pattern, as shown in Figure 6-4. Note that the service times $\{t_S\}$ would now be a parameter of this component rather than the server component.

```
┌─────────────────────────────────────────────────┐
│                                                   │
│      ╭────────╮  ┌─┐        ╭────────╮            │
│      │ Arrival │──│c│──────▶│ Arrival │           │
│      │         │  └─┘        │  (c)   │           │
│      ╰────────╯             ╰────────╯            │
│                                                   │
│   {c = new Entity("Customer");                    │
│        c.serviceTime = t_S}                       │
│                                                   │
└─────────────────────────────────────────────────┘
```
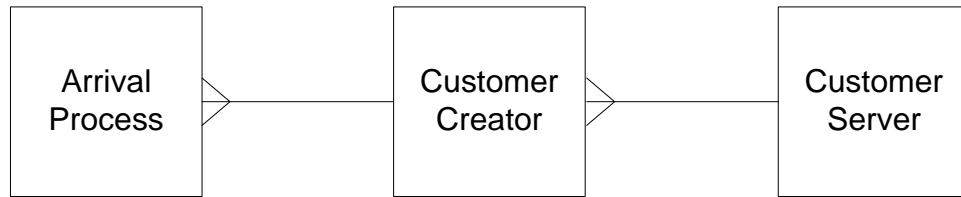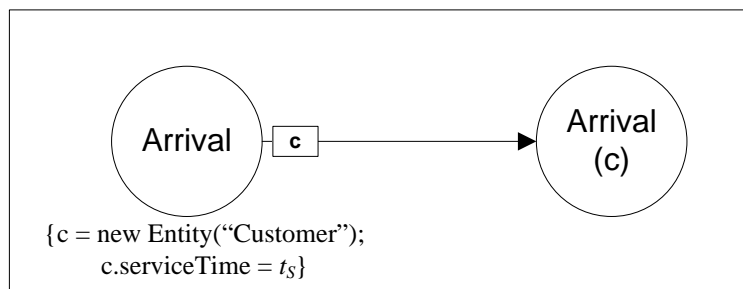
**Figure 6-4. Customer Creator with serviceTime Attribute**

The corresponding server is shown in Figure 6-5. The only difference between Figure 6-5 and Figure 6-2 is the delay on the scheduling edge from StartService to EndService. The listeners in the assembly would be exactly as in Figure 6-3.
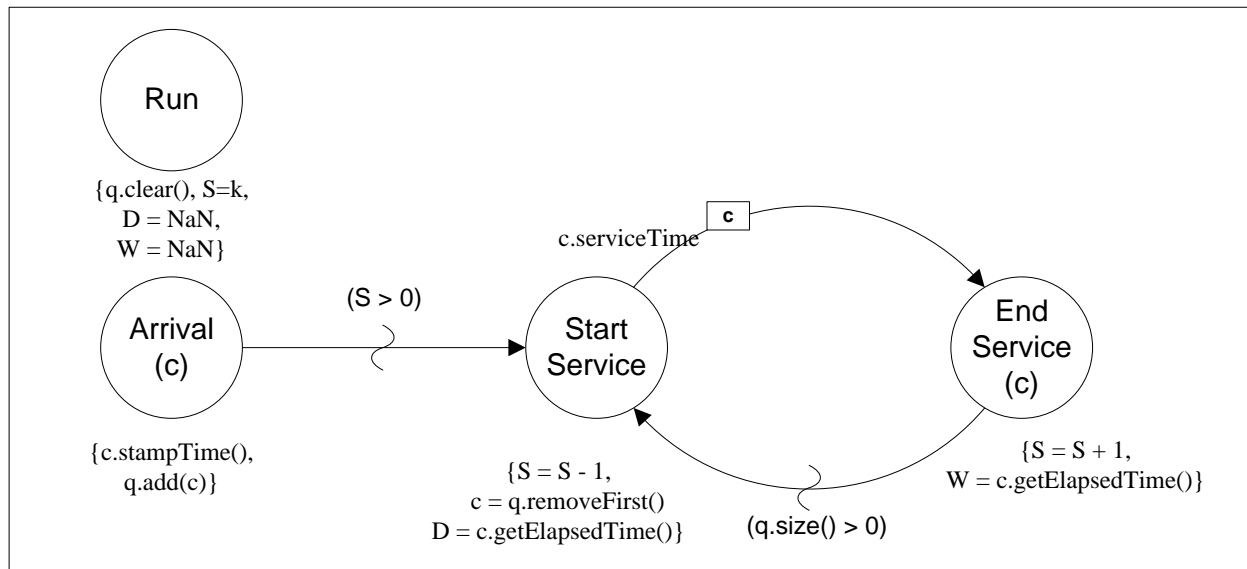
**Figure 6-5. Customer Server for Entities with serviceTime Attribute**

In general, whatever attributes are deemed necessary may be defined to create new Entity types. These attributes may be of any type themselves, including other Entities.

# 7. Discrete Event Simulation Problems for Modeling

## 7.1. Multiple Server Queue

Customers arrive to a facility according to an arrival process one at a time, with the times between arrivals being some sequence of non-negative random variables. There are k servers at the facility in parallel. An arriving customer who finds an available server may begin service immediately. Service times are according to a sequence of non-negative random variables. Arriving customers who find all servers are busy wait in a common queue for service. When a server has completed serving a customer, if there are one or more customers waiting in the queue, then service begins immediately on the next customer. The queue is organized as first-come first-served. When a customer completes service, they leave the system. Initially the system is empty and all servers are idle.

Measures to be estimated are: average number in queue, average utilization, average delay in queue, average time in system.

## 7.2. Multiple Server Queue with Finite Waiting Room

The same situation as in Problem 7.1 but there is a finite amount of waiting room in the queue, c. An arriving customer who finds fewer than c customers already in the queue joins the queue. An arriving customer finding c already in the queue leaves the system without joining the queue and never returns. Initially the system is empty and all servers are idle.

In addition to measures in Problem 7.1, the percentage of lost customers is to be estimated.

## 7.3. Multiple Server Queue with Batch Arrivals

The same situation as Problem 7.1, except that each "arrival" contains a possibly random number of customers. The system starts empty and idle. Additional measures could include the average time to process a complete batch.

## 7.4. Multiple Server Queue with Batch Service

The same situation as Problem 7.1, except that service is done in batches of a fixed size B. Even if a server is available, arriving customers wait until there are B customers in the queue to start service. Upon completion of a batch, another batch is started if there are B or more customers in the queue; otherwise a server will remain idle until the queue reaches size B again to start another batch. Additional measures include the average number in the queue and the average time in the system per customer.

## 7.5. Tandem Queue

The same situation as in Problem 7.1, except that there are two sets of parallel servers in tandem. Customers arriving to the first station are processed exactly as in Problem 7.1. However instead of leaving the system upon completion of service at the first station, a customer proceeds to the second station. If a server at the second station is available, then service starts there immediately. If not, then that customer waits in a second queue. The numbers of servers at each station can be different, as can be the probability distributions of service times. Measures include the ones in Problem 7.1 for each station. Initially the system is empty and all servers are idle.

## 7.6. Tandem Queue with Rework

The same situation as Problem 7.5, except that a customer departing from the first station proceeds to the second station with a certain probability p, and leaves the system with probability 1 – p.

## 7.7. Transfer Line

This is the same situation as in Problem 7.5, except that the number of stations in tandem is a parameter of the model. That is, there are n stations, each consisting of a multiple server queue. Arriving jobs must be processed by an identical sequence of stations in turn, leaving one station to arrive at the next. A job doesn't depart the system until it has completed service at the last station. The model should ideal be such that the number of stations can be specified as a parameter. That is, the same model can represent any number of workstations in series. Initially the system is empty and idle.

## 7.8. Transfer Line with Blocking

The same situation as Problem 7.7, except each station has a finite capacity. Customers arriving to find the first workstation's queue at capacity will balk – exit the system without receiving service. A customer who completes service at one station to find the following station's queue at capacity is blocked – that is, it must remain with the server who is unable to serve another customer. When a space opens in the following station's queue, a blocked customer can then advance, which in turn may cause the server just left to begin service if customers are in their queue. As with Problem 7.7, the model should be able to represent any number of workstations, and it is initially empty and idle. Measures include percentage of customers who balk, average number of blocked customers at each station, average number in queue at each station, and the percentage of time each station's servers are idle, working, or blocked.

## 7.9. Machine Failure Model I

A facility has m machines, each of which fails independently according to given probability distribution over simulated time. There are r repair people available to fix failed machines. A machine that fails when all repair people are busy repairing machines waits in a fifo queue for repair. The times to repair a failed machine are likewise from a

given probability distribution. A repair person works on a machine alone until it is working, then immediately begins repairing another machine if one is waiting for service; otherwise, they become idle. Measures include average utilization of repair people and average number of available machines. Initially all repair people are idle and all machines have just been "turned on."

## 7.10. Machine Failure Model II

The same situation as Problem 7.9 except that machines also process jobs that arrive according to some arrival process. An arriving job that finds no available machine waits in a fifo queue until a machine becomes available. The exception is when a machine fails when processing a part, that part gets put back into the queue, and can be worked on by another machine if available. When processing starts again, a new service time is drawn for the job. Machines still fail according to simulation time. Initially there are no jobs in the system. Additional measures include the proportion of time machines are busy, idle, or failed and the average number of jobs in the queue.

## 7.11. Machine Failure Model III

The same situation as Problem 7.10, except that each job starts with a certain amount of processing time required, generated by the service time distribution. If a machine fails when being processed, that job receives "credit" towards the processing time by the amount it has been worked on. For example, if a job requires 5.3 hours to complete and is processed for 2.4 hours on a machine when it fails, then when the job starts again on a machine (which could be the same one or another), the processing time is the remaining time of 2.9 hours.

## 7.12. Machine Failure Model IV

The same situation as Problem 7.11, except that machine times to failure are a function of their operational time. That is, the "time to failure" means the number of hours of operation a machine has until it fails. When a machine is idle, it is not failing.

## 7.13. Multiple Server Queue with Reneging I

The same situation as Problem 7.1, except that customers are "impatient" and will exit the queue if they are not served within a certain amount of time. A customer who joins a queue and subsequently leaves is said to have "reneged." Each customer's impatience time is from a given probability distribution. Additional measures include the percentage of customers who renege.

## 7.14. Multiple Server Queue with Reneging II

The same situation as Problem 7.13, except that when a customer's renege time comes, they will renege with a given probability based on their position in the queue. For example, if $p_0 = 0.25, p_1 = 0.5, p_i = 1$ for $i \geq 2$, then a customer whose "impatience" time

is up but is at the head of the queue will only renege with probability 0.25, whereas if they are second in line they will renege with probability 0.5, etc.

## 7.15. Multiple Server Queue with Balking and Reneging

The same situation as in Problem 7.13 (or Problem 7.14), except that customers also Balk – that is choose not to enter the queue – based on the number of customers in the queue upon arrival. The probability a customer balks is given by some function $f(Q)$, where Q = number in the queue. One example might be $f(Q) = 1/(Q+1)$. Once the customer has entered the queue, they may renege based on the same criteria described in Problem 7.13 or Problem 7.14.

## 7.16. Multiple Server Queue with Two Types of Servers

The same situation as Problem 7.1, except that there are two types of servers. An arriving customer prefers one type over the other, but will receive service from the non-preferred one if the other isn't available. The system starts empty and idle, and measures include average number in queue and average utilization for each type of server.

## 7.17. Multiple Server Queue with Two Types of Customers

The same situation as Problem 7.1, except that there are two types of customers who arrive according to separate arrival processes and have different service time distributions. The servers "prefer" one type of customer over the other. That is, when a server completes service, they will begin on the preferred type if one is waiting in the queue, even if it entered the system after other non-preferred customers; otherwise they will begin servicing the non-preferred customer. Service times are a function of the customer type. The system starts empty with the servers idle, and measures include average utilization and average number of each type of customer in the system.

## 7.18. Multiple Server Queue with Two Type of Customers and Two Types of Servers

This model combines that of Problem 7.16 and Problem 7.17. There are two types of customers, say A and B, and two types of servers, say 1 and 2. Type A customers prefer type 1 servers, and vice versa, whereas type B customers prefer type 2 servers (and vice versa). However, each will match with the non-preferred type is the preferred one is not available. There are three variants for how the service times are modeled:

6. Service times depend on the customer type only

7. Service times depend on the server type only.

8. Service times depend on both the customer type and the server type.

### 7.19.  Assembly Model

A completed part consists of 1 component of type A and 2 of type B.  Each type of component arrives to the facility according to independent processes.  There are k identical servers who assemble parts when the right mix of components is available.  The assembly time is a sequence of possibly random values.  An arriving component that finds an available server and the right number of other components can begin assembly immediately.  Otherwise it waits in a queue.  Initially the system is empty and all servers are idle.  Measures include average utilization of the servers, production rate of the parts, and average number of components in each queue by type.

### 7.20.  Two Types of Customers with Different Server Requirements

Two types of jobs, A and B, arrive according to independent processes to a service facility with k identical servers.  Jobs of type A require 1 server, while jobs of type B require 2 servers.  Once servers start processing a job they must stay with that job until it is completed.  When there is a choice, servers prefer to work on type B jobs, but will never be idle if there is some work to be done.  Arriving jobs wait in separate queues if they cannot be served.  Initially the system is empty and the servers are idle.  Measures include average number in each type of queue and average utilization of servers.

### 7.21.  Round Robin CPU Model

Computing "jobs" from a finite population arrive to a single CPU for processing.  The jobs are processed in a "round robin" manner instead of as in Problem 7.1.  Unfinished jobs wait in a FIFO queue.  There is a given quantum amount that the CPU will spend on each job, after which if the job is still unfinished, it re-joins the queue at the end to await its turn.  The process of changing jobs is the "swap time" and is a given, fixed amount of time.  So each cycle takes $\sigma + \tau$ where $\sigma$ is the quantum amount and $\tau$ is the swap time.  The initial job times are given by a random variable, and each cycle reduces the remaining processing time by $\sigma$.  Each job starts by waiting a random amount of time and then being "submitted" to the CPU.  When a job is completed, it then waits another random amount of time before being submitted again.  The system starts empty and idle, with each job beginning its wait before submission.  Measures include average response time (the amount of time between job submission and completion), CPU utilization, and average number of jobs in the queue.

### 7.22.  Simple Inventory Model with Backordering

A company is seeking to manage the inventory level for a single product.  Customers purchasing the product arrive according to an arrival process. Each arriving customer attempts to purchase a random number of items, D.  The inventory level at the company is reviewed periodically and a decision is made whether or not to place an order from its supplier. When the company places an order, it takes a certain amount of time ("lead time") for the order to arrive.  Due to a variety of circumstances, the lead times are random.  The company uses a <s,S> ("little s big S") policy for its ordering decisions.  If the inventory position at review time is below s, then an order is placed that is the

difference between it and the number S.  The inventory position includes the amount of the product on-hand and the amount of the product that is on-order (that is, has been ordered from the supplier but not yet received).  Note that it is possible for there to be two or more outstanding orders from the supplier.

When a customer's order cannot be filled, the unfilled portion is put on backorder. When the company receives a shipment from its supplier, backorders are immediately filled and the remainder put in stock. For example, if there are 5 items in stock and a customer wants to buy 8 items, the customer is given the five items in stock and the remaining three are backordered.

Measures include the average amount of inventory on-hand, the average amount on backorder, and the average amount on-order, and the percentage of customers who get their orders filled immediately.  The values of s and S are policy variables that can be chosen by the manager, who presumably wants to pick the "best" ones.

## 8. Event Graph Models and Simkit

Simkit has been designed to make implementing an Event Graph model as straightforward as possible. Every element in an Event Graph model has a corresponding element in Simkit. This note describes some of the basic correspondences between Event Graph models and Simkit classes. To implement a Simkit model, be sure that the latest version of Simkit is installed. If you are using an IDE, make sure it has been configured so that the simkit.jar file is on the class path. This information applies to Simkit version 1.3.7 or greater.

## 8.1. Basics

Table 5 below shows the relationship between the basic elements of an Event Graph model and the corresponding Simkit implementation.

| Event Graph | Simkit |
|---|---|
| Simulation Component | Subclass of SimEntityBase |
| Event Graph Parameter | Private instance variable, setter and getter |
| State Variable | Protected instance variable, getter, no setter |
| Event | 'do' method |
| Scheduling Edge | Call to `waitDelay()` in scheduling event's 'do' method |
| Run Event | `reset()` method to initialize state variables; `doRun()` method to fire PropertyChange events for time-varying state variables |
| Event scheduled from Run event | Call to `waitDelay()` in `doRun()` method |
| Event scheduled from any Event | Call to `waitDelay()` in scheduling event's 'do' method |
| Event cancelled from any Event | Call to interrupt() from canceling event's 'do' method |
| Priority on Scheduling Edge | `Priority` instance as third argument to `waitDelay()` |
| Argument(s) on Events | Arguments in corresponding 'do' method |
| Parameter(s) on Edges | Add parameter values/expressions last (in correct order) in `waitDelay()` |
| Canceling Edge | Call to `interrupt()` |

**Table 5. Event Graph Components and Their Simkit Counterparts**

Simkit components are implemented using Simkit by subclassing the abstract `SimEntityBase` class (in the `simkit` package). Each Event Graph parameter is implemented by defining a private instance variable in the subclass with both a setter and a getter method. Similarly, each state variable is implemented by a protected instance variable with a getter method but no public setter method.

Each Event in the Event Graph is implemented by a corresponding method in the subclass that starts with 'do' followed by the name of the Event. So an event called "Foo" is implemented by a method called "doFoo". The one slight deviation from this is the Run event, which is implemented using two methods: reset() and doRun(). The reset() method is where the state variable initializations are performed, and the doRun() method is where the corresponding PropertyChangeEvents are fired and other Events are scheduled, if there are any specified.

Each 'do' method follows the same order of operations: (1) All state transitions are performed first; (2) Any canceling edges executed next; (3) Finally, any scheduling edges are executed. Remember that a scheduling edge simply places the Event on the Event List, and a canceling edge only removes the next scheduled Event that matches.

A state transition typically entails three steps: (1) The old value of the state is saved in a temporary variable; (2) The new value of the state is stored in its instance variable; (2) A PropertyChangeEvents is fired by executing the firePropertyChange() method, passing the property name, the old value, and the new value.

One exception to this is in doRun(), where there is no logical "old value."

### 8.1.1. Example: Primitive State Variables

Suppose an Event Graph component has an int state variable called foo whose initial value is 0, and suppose the Event Bar has a state transition that increments foo by 2. The parts of the Simkit component class are as follows:

```
. . .
protected int foo;
. . .
public void reset() {
      super.reset();
      foo = 0;
}

public void doRun() {
      firePropertyChange("foo", getFoo());
}

public void doFoo() {
      int oldFoo = getFoo();
      foo = foo + 2;
      firePropertyChange("foo", oldFoo, getFoo());
}

public int getFoo() {
      return foo;
}
```

The variable foo is first defined as a protected instance variable, given a getter method, and initialized to 0 in the reset() method. The PropertyChangeEvent is fired in doRun() using the 2-argument version of the firePropertyChange method. The state transition in doFoo() is done in the three steps described above.

## 8.2.   Scheduling Edge Options

In addition to denoting the event that initiates the scheduling of another event (the event at the tail of the edge) and the event that is scheduled (the event at the head of the edge), every scheduling edge has four additional properties:  (1) A non-negative number, the time delay; (2) An optional Boolean condition; (3) Optional edge parameters; and (4) An optional priority.
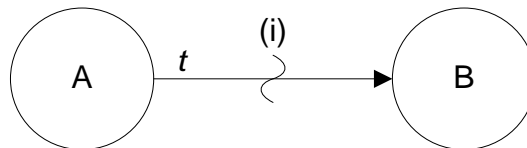
Some Prototypical examples follow.

### 8.2.1.  Simplest Scheduling Edge
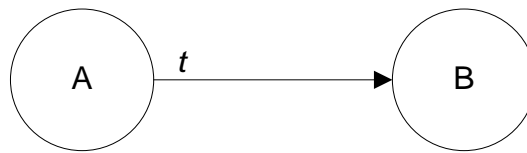


**Simkit Code**

```
public void doA() {
      // State transitions for Event A
      waitDelay("B", t);
}
public void doB() {
      // State transitions for Event B.
}
```
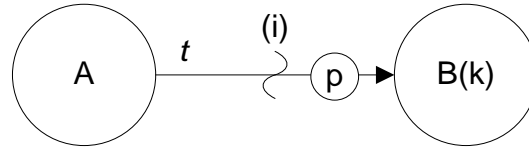
### 8.2.2.  Scheduling Edge with Boolean Condition



**Simkit Code**

```
public void doA() {
      // State transitions for Event A
      if (i) {
            waitDelay("B", t);
      }
}
public void doB() {
      // State transitions for Event B.
}
```
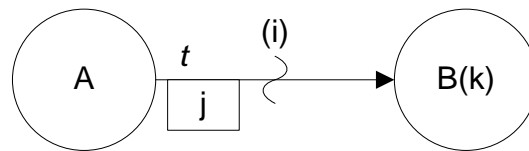
### 8.2.3. Scheduling Edge with Priority



**Simkit Code**

```
public void doA() {
      // State transitions for Event A
      if (i) {
            waitDelay("B", t, Priority.HIGH);
      }
}
public void doB() {
      // State transitions for Event B.
}
```

Note: This assumes that "p" is "High" priority. Other Simkit built-in options include "HIGHER," "HIGHEST," "LOW," "LOWER," "LOWEST," and "DEFAULT." The user can also define custom priority levels

### 8.2.4. Scheduling Edge with Argument



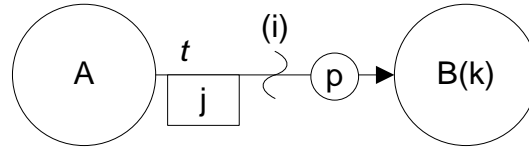**Simkit Code**

```
public void doA() {
      int j = . . .;
      // State transitions for Event A
      if (i) {
            waitDelay("B", t, j);
      }
}
public void doB(int k) {
      // State transitions for Event B.
}
```

Note: this assumes that the argument k is of type `int`. In general, the argument(s) can be any type. The parameter(s) can be any compatible expression.

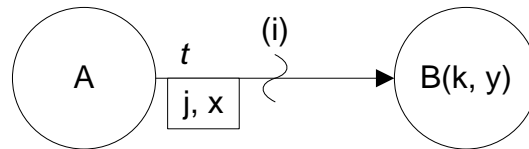### 8.2.5. Scheduling Edge with Argument and Priority



**Simkit Code**

```
public void doA() {
      int j = . . .;
      // State transitions for Event A
      if (i) {
            waitDelay("B", t, Priority.HIGH, j);
      }
}
public void doB(int k) {
      // State transitions for Event B.
}
```

Note: again, priority p is assumed to be "HIGH."

### 8.2.6. Scheduling Edge with Multiple Arguments



**Simkit Code**

```
public void doA() {
      int j = . . .;
      String x = . . .;
      // State transitions for Event A
      if (i) {
            waitDelay("B", t, j, x);
      }
}
public void doB(int k, String y) {
      // State transitions for Event B.
}
```
Note: This assumes that k is an `int` and y is a `String`.

## 8.3.   Canceling Edge Options

Similar to a scheduling edge, every canceling edge has two additional properties: (1) An optional Boolean condition; and (2) Optional edge parameter(s). Canceling Edges do not have time delays or priorities.

### 8.3.1. Simplest Case



**Simkit Code**

```
public void doA() {
      // State transitions for Event A
      interrupt("B");
}
public void doB() {
      // State transitions for Event B.
}
```

### 8.3.2. Canceling Edge with Boolean Condition
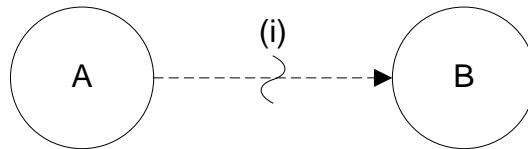


**Simkit Code**

```
public void doA() {
      // State transitions for Event A
      if (i) {
            interrupt("B");
      }
}
public void doB() {
      // State transitions for Event B.
}
```
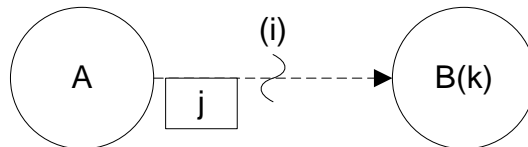
### 8.3.3. Canceling Edge with Argument



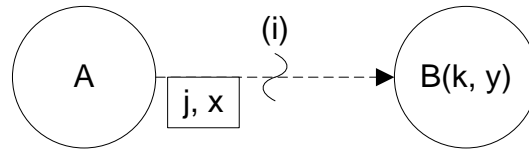**Simkit Code**

```
public void doA() {
      int j = . . .;
      // State transitions for Event A
      if (i) {
```

```
            interrupt("B", j);
      }
}
public void doB(int k) {
      // State transitions for Event B.
}
```

### 8.3.4. Canceling Edge with Multiple Arguments



**Simkit Code**

```
public void doA() {
      int j = . . .;
      String x = . . .;
      // State transitions for Event A
      if (i) {
            interrupt("B", j, x);
      }
}
public void doB(int k, String y) {
      // State transitions for Event B.
}
```

Note: As previously, this assumes that k is an `int` and y is a `String`.

## 8.4. RandomVariates

Streams of random variates are obtained in Simkit using instances of `RandomVariate`. Since `RandomVariate` is an interface, it cannot be instantiated. Instead, instances of `RandomVariate` are obtained from the `RandomVariateFactory` class using the static method `getInstance()`. Many of the common distributions are in Simkit. If a distribution that is not in Simkit is desired, then a class must be written that implements the `RandomVariate` interface. Once this has been done, `getInstance()` can be used to obtain instances of the new class using `RandomVariateFactory`.

The RandomVariate instances in Simkit can be requested by the name of the distribution, such as "Exponential," "Gamma," etc. Consult the Simkit documentation to see which distributions are included.

The convention for naming a RandomVariate class is <Name>`Variate`. If desired, the full name may be used. The full class name or the fully-qualified name can always be used. The `RandomVariate` classes in Simkit are in the `simkit.random` package.

### 8.4.1. Examples

Exponential with mean of 1.7 using shorthand:

```
RandomVariate rv =   RandomVariateFactory.getInstance("Exponential",
1.7);
```

Obtaining this distribution using the full class name:

```
RandomVariate rv =
        RandomVariateFactory.getInstance("ExponentialVariate",
    1.7);
```

Obtaining this distribution using the fully-qualified class name:

```
RandomVariate rv = RandomVariateFactory.getInstance(
        "simkit.random.ExponentialVariate", 1.7);
```

Gamma with parameters $\alpha = 1.2$ and $\beta = 2.5$:

```
RandomVariate rv =
    RandomVariateFactory.getInstance("Gamma", 1.2, 2.5);
```

Constant with value 5.6 (only generates values of 5.6):

```
RandomVariate rv =
    RandomVariateFactory.getInstance("Constant", 5.6);
```

### 8.4.2.  User-Defined Random Variate Classes

More random variate distributions may be written by implementing the
`RandomVariate` interface and its methods.  These may be obtained using
`RandomVariateFactory.getInstance()` in the same manner as for the distributions in
Simkit. The fully-qualified class name will always work.  In addition, the user-defined
package it is in may be added to the search list with the
`RandomVariateFactory.addSearchPackage(String)` method, passing in the name of the
package.  The `xxxVariate` convention, if followed, will allow the abbreviated name as
well.

For example, suppose the class `CauchyVariate` is implemented in the `myrng`
package.  If `RandomVariateFactory.addSearchPackage(myrng)` is invoked somewhere,
then an instance may be obtained by any of the following invocations:

```
RandomVariateFactory.getInstance("Cauchy");
RandomVariateFactory.getInstance("CauchyVariate");
RandomVariateFactory.getInstance("myrng.CauchyVariate");
```

## 8.5.   Event Graph Component Examples in Simkit

This section will illustrate the mapping between Event Graph models with some
Event Graph components that are more concrete and useful.
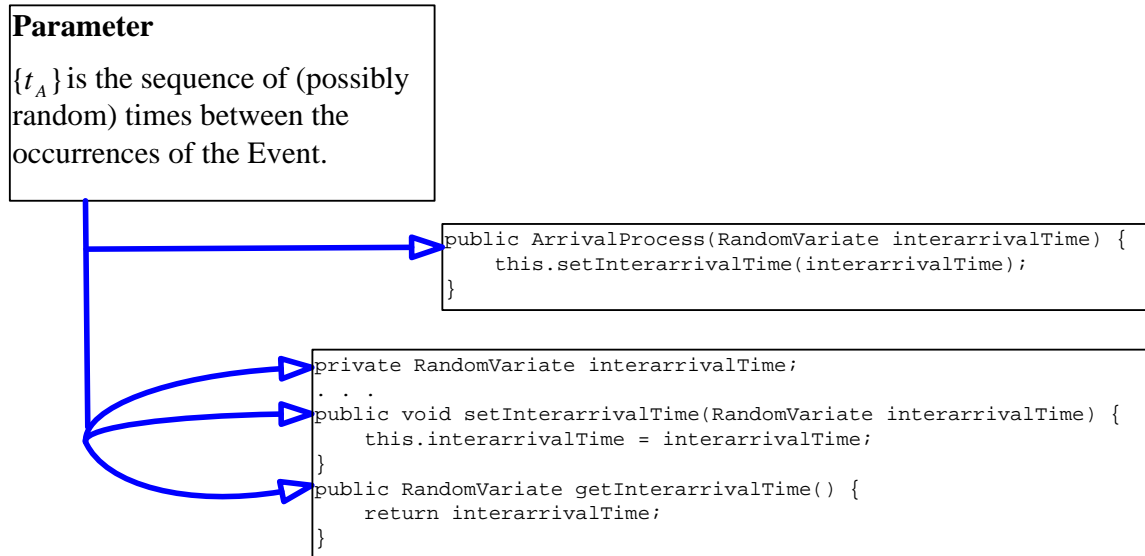
### 8.5.1.  The Arrival Process

The mapping between the parts of an Event Graph component and a Simkit class
will now be illustrated with an example.  Recall the definition of the ArrivalProcess
Event Graph component.

The single parameter is the sequence of interarrival times $\{t_A\}$.  The
corresponding Simkit variable will be called "`interarrivalTime`" to make the code more
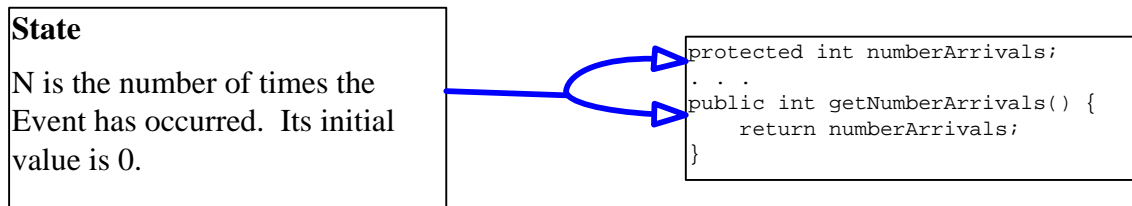
readable.  It is defined to be a private instance variable with a setter and a getter, as described previously.  The connection is shown in Figure 8-1.  Note also that the parameter is also an argument to the constructor for the class.
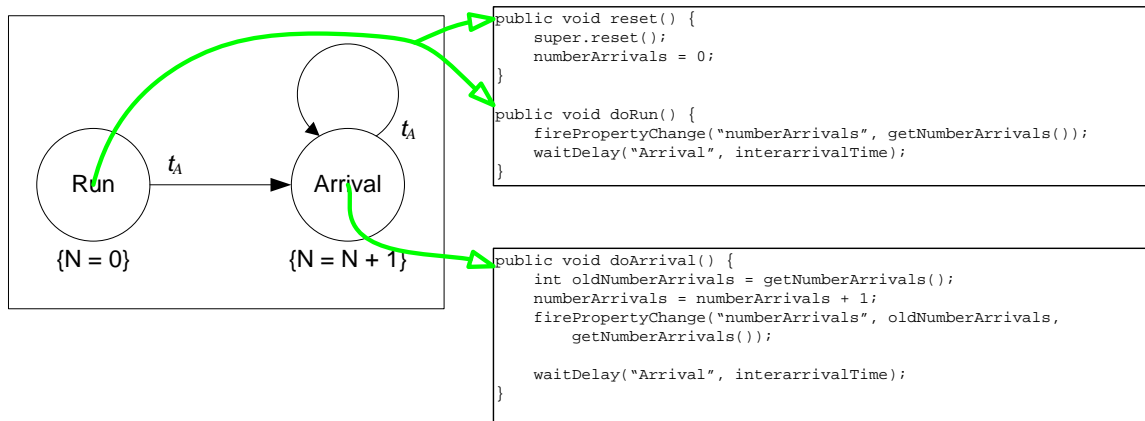
**Parameter**

$\{t_A\}$ is the sequence of (possibly random) times between the occurrences of the Event.

```
public ArrivalProcess(RandomVariate interarrivalTime) {
    this.setInterarrivalTime(interarrivalTime);
}
```

```
private RandomVariate interarrivalTime;
. . .
public void setInterarrivalTime(RandomVariate interarrivalTime) {
    this.interarrivalTime = interarrivalTime;
}
public RandomVariate getInterarrivalTime() {
    return interarrivalTime;
}
```

**Figure 8-1. ArrivalProcess Parameter Defined in Simkit Component**

Similarly, the state variable N will be called "numberArrivals" and is defined to be a protected instance variable with a getter but no setter, as shown in Figure 8-2.

**State**

N is the number of times the Event has occurred.  Its initial value is 0.

```
protected int numberArrivals;
. . .
public int getNumberArrivals() {
    return numberArrivals;
}
```
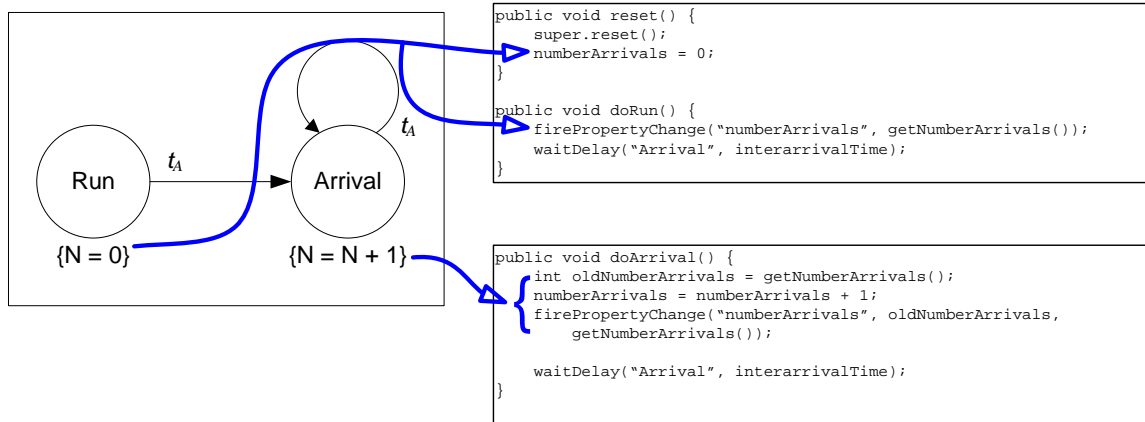
**Figure 8-2. Arrival Process State Defined in Simkit Component**

The Run Event is implemented as both `reset()` and `doRun()` methods, while the Arrival Event is defined to be the method `doArrival()`, as shown in Figure 8-3.

Run  $t_A$  Arrival  $t_A$

{N = 0}   {N = N + 1}

```
public void reset() {
    super.reset();
    numberArrivals = 0;
}

public void doRun() {
    firePropertyChange("numberArrivals", getNumberArrivals());
    waitDelay("Arrival", interarrivalTime);
}
```

```
public void doArrival() {
    int oldNumberArrivals = getNumberArrivals();
    numberArrivals = numberArrivals + 1;
    firePropertyChange("numberArrivals", oldNumberArrivals,
        getNumberArrivals());

    waitDelay("Arrival", interarrivalTime);
}
```

**Figure 8-3. ArrivalProcess: Defining Events in Simkit Component**

The state transition for Run has one part in `reset()` and another in `doRun()`. The initial value is assigned in `reset()` and the corresponding property change is fired in `doRun()`, as shown in Figure 8-4. The `firePropertyChange()` call in `doRun()` is the 2-argument type. In contrast, the state transition for Arrival is completely in the `doArrival()` method and consists of three parts: saving the old value, making the state transition, and finally firing the property change with the old value and the new value, as seen in Figure 8-4.



**Figure 8-4. ArrivalProcess: State transitions in Simkit Component**

Scheduling edges are implemented by calls to `waitDelay()`. The arguments to the respective waitDelay() calls for Run and Arrival events are shown in Figure 8-5.
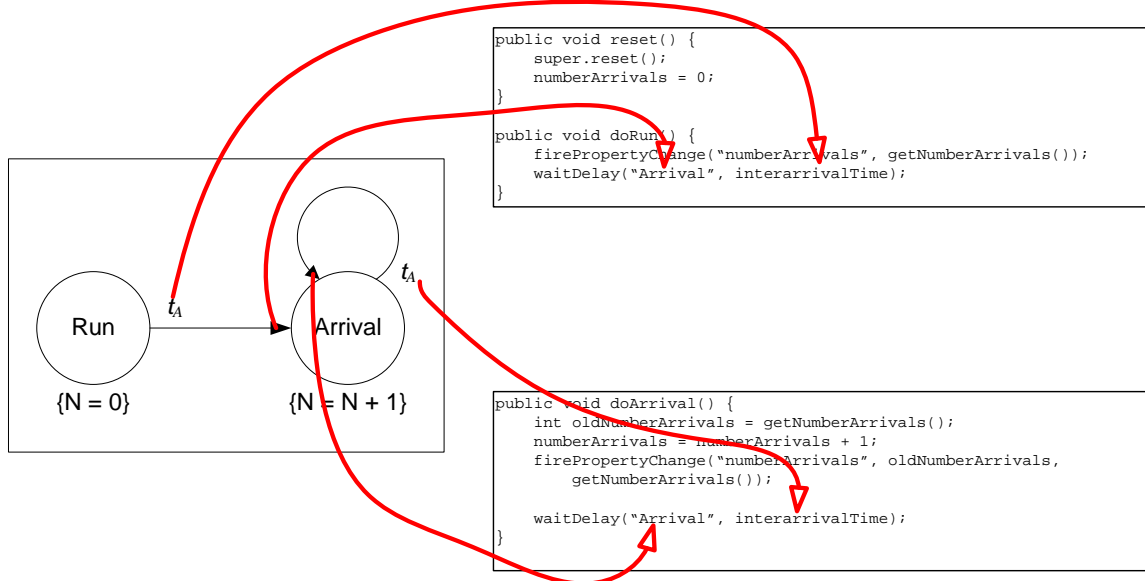


**Figure 8-5. ArrivalProcess: Scheduling Edges and Delays in Simkit Component**

Note that the Event nodes each scheduling edge points to corresponds to a String of the same name that is the first argument of the corresponding `waitDelay()` call. Also, the delays on each scheduling edge correspond to a reference to `interarrivalTime`, the `RandomVariate` that implements the parameter $\{t_A\}$.

Putting it all together, the complete code for the Simkit class is shown in Figure 8-6. Note that the class has been defined to be in a package called "`simkit.examples`" and that the relevant import calls have been added. It is also important to understand that the comments have been omitted from the code here.

```
package simkit.examples;

import simkit.SimEntityBase;
import simkit.random.RandomVariate;

public class ArrivalProcess extends SimEntityBase {

    private RandomVariate interarrivalTime;

    protected int numberArrivals;

    public ArrivalProcess(RandomVariate rv) {
        this.setInterarrivalTime(rv);
    }

    public void reset() {
        super.reset();
        numberArrivals = 0;
    }

    public void doRun() {
        firePropertyChange("numberArrivals", getNumberArrivals());

        waitDelay("Arrival", interarrivalTime);
    }

    public void doArrival() {
        int oldNumberArrivals = getNumberArrivals();
        numberArrivals = numberArrivals + 1;
        firePropertyChange("numberArrivals", oldNumberArrivals,
            getNumberArrivals());

        waitDelay("Arrival", interarrivalTime);
    }

    public void setInterarrivalTime(RandomVariate interarrivalTime) {
        this.interarrivalTime = interarrivalTime;
    }

    public RandomVariate getInterarrivalTime() {
        return interarrivalTime;
    }

    public int getNumberArrivals() {
        return numberArrivals;
    }
}
```

**Figure 8-6. Complete Code for ArrivalProcess Class**

Although for a well-named getter or setter method a comment could be considered optional, for the key methods (i.e. the 'do' methods) comments, especially

Javadoc comments, are essential.  Of course too much commenting can be nearly as bad as too little.  Figure 8-7 shows a reasonable comment for the `doArrival()` method.

```
/**
 * An arrival of a customer.<br>
 * State transition: increment number of arrivals <br>
 * Schedule another arrival after interarrivalTime delay
 */
public void doArrival() {
    . . .
```

**Figure 8-7. A Reasonable Comment for doArrival()**

## 8.5.2.  Multiple Server Queue Component

Recall the simple server component that is a minimalistic model of a multiple server queue.  The Event Graph is reproduced in Figure 8-8; to emphasize the fact that the StartService Event is scheduled with high priority, that is indicated on the two edges that schedule it in Figure 8-8.
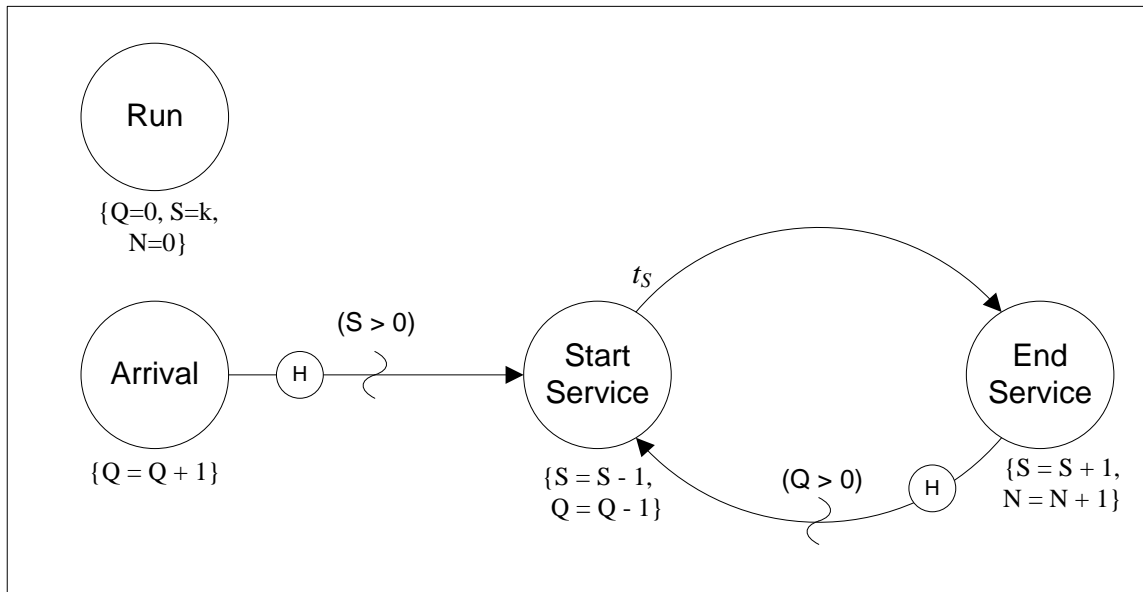


**Figure 8-8. SimpleServer Component**

The Simkit definition of the parameters ($k$ and $\{t_S\}$) and the state variables (Q, S, and N) are straightforward and not shown here.  To demonstrate the conditions and setting of priorities on scheduling edges, the code for the two scheduling edges are shown in Figure 8-9.

```
public void doArrival() {
       . . .
    if (getNumberAvailableServers() > 0) {
        waitDelay("StartService", 0.0, Priority.HIGH);
    }
}

public void doEndService() {
```

```
        . . .
    if (getNumberInQueue() > 0) {
        waitDelay("StartService", 0.0, Priority.HIGH);
    }
}
```

**Figure 8-9.  Scheduling Edges with Conditions and Priorities**

Note how the Boolean conditions are implemented by wrapping the `waitDelay()` calls in `if` statements and how the priorities are set using `Priority.HIGH`.  Also, note that even though the time delays are not explicitly shown in Figure 8-8, they are explicitly implemented as 0.0 in the code.

### 8.5.3.  EntityCreator

This is a simple creator pattern that illustrates passing a parameter on a scheduling edge as well as for using transient entities.  The Event Graph component for EnityCreator is shown in Figure 8-10.
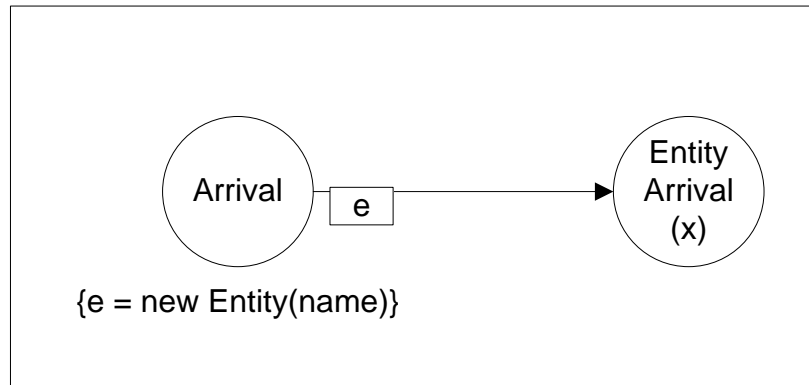


**Figure 8-10. EntityCreator Event Graph Component**

There are no state variables, and there is a single parameter `name`.  Figure 8-11 shows the corresponding code for the Arrival event.

```
public void doArrival() {
    Entity entity = new Entity(getName());
    waitDelay("EntityArrival", 0.0, entity);
}
```
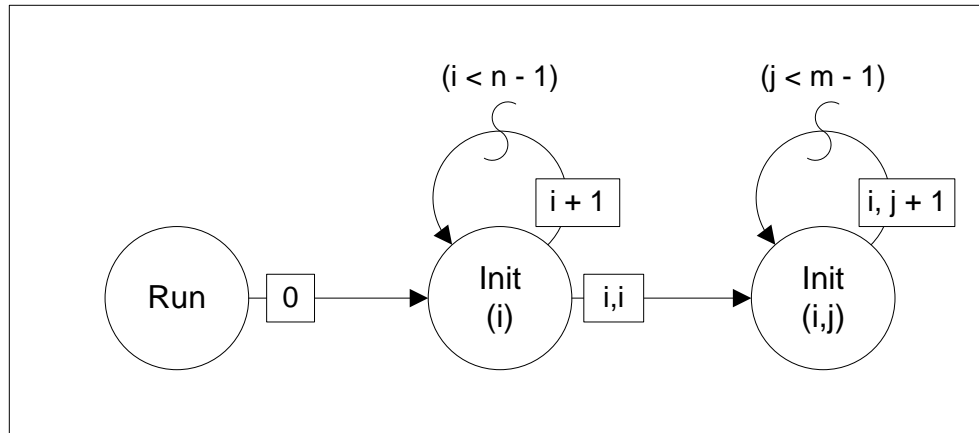
**Figure 8-11. Code for Arrival Event for EntityCreator**

Note that the argument, the local variable, `entity`, is simply added as the last argument to the `waitDelay()` call to pass it to the EntityArrival event.

### 8.5.4.  Nested For Loop

A slightly more complex illustration is shown in Figure 8-12, the Event Graph version of a nested 'for' loop.

**Figure 8-12. Nested 'for' Loop Event Graph Component**

The corresponding code for this (with the setters and getters omitted) is shown in Figure 8-13.

```
public class NestedForLoop extends SimEntityBase {

    private int innerLimit;
    private int outerLimit;

    public NestedForLoop(int outerLimit, int innerLimit) {
        this.setOuterLimit(outerLimit);
        this.setInnerLimit(innerLimit);
    }

    public void doRun() {
        waitDelay("Init", 0.0, Priority.HIGH, 0);
    }

    public void doInit(int i) {
        if (i < getOuterLimit() - 1) {
            waitDelay("Init", 0.0, Priority.HIGH, i + 1);
        }
        waitDelay("Init", 0.0, Priority.HIGHER, i, 0);
    }

    public void doInit(int i, int j) {
        if (j < getInnerLimit() - 1) {
            waitDelay("Init", 0.0, Priority.HIGHER, i, j + 1);
        }
    }
. . .
*** Setters and getters omitted ***
```

**Figure 8-13. Code for Nested 'for' Loop**

Note that n and m are implemented as `outerLimit` and `innerLimit`, respectively. Also, this example shows how to pass more than one argument on a scheduling edge via `waitDelay()`. Specifically, the arguments are simply listed in order last in the call. This also illustrates that if a priority is desired, it comes before the arguments and after the time delay. In this case, the inner Init(i,j) event has HIGHER priority than Init(i), which only

has HIGH priority.  Although it is likely not germane to the model, this prioritization ensures that the innermost Events are completed before the outermost Event advances, the way a nested 'for' loop works in programming languages.

## 8.6.  Discussion

At this point there is sufficient information to enable the creation of Simkit Java classes from Event Graph components.  The process of doing this is a direct translation of each element of the Event Graph component, as summarized in Table 5.  There remains the issue of actually executing the model, which will be presented next.

## 8.7.  Creating and Executing a Model in Simkit

At a minimum creating and executing a Simkit model involves the following steps:

1.  Instantiate the Event Graph components

2.  Connect the components as SimEventListeners or by using Adapters, if necessary.

3.  If collecting statistics, instantiate statistical PropertyChangeListener objects and connect to the relevant SimEntity objects as listeners; if dumping properties, instantiate SimplePropertyDumper and connect it

4.  Configure the Schedule class (e.g. verbose mode)

5.  Initialize and run

6.  Output statistics from PropertyChangeListener objects, if this is done, or any post-simulation information desired

### 8.7.1.  ArrivalProcess Execution

Figure 8-14 shows the contents of a `main()` method that executes the ArrivalProcess component discussed earlier.  The numbers in parentheses are not part of the code but correspond to the step in the previous section.

```
(1)
RandomVariate interArrivalTime = RandomVariateFactory.getInstance(
        "Exponential", 3.2);
ArrivalProcess arrivalProcess = new ArrivalProcess(interArrivalTime);

(3)
SimplePropertyDumper simplePropertyDumper = new SimplePropertyDumper();
arrivalProcess.addPropertyChangeListener(simplePropertyDumper);

System.out.println(arrivalProcess);

(4)
Schedule.setVerbose(true);
Schedule.stopAtTime(15.0);

(5)
Schedule.reset();
```

```
Schedule.startSimulation();

(6)
System.out.println("At time " + Schedule.getSimTime() +
    " there have been " + arrivalProcess.getNumberArrivals() + "
arrivals");
```

**Figure 8-14. Code to Run ArrivalProcess in Verbose Mode**

(1) The RandomVariate instance is obtained from RandomVariateFactory (in this case specifying an Exponential(3.2) distribution) and this instance is passed into the constructor for ArrivalProcess.

(2) Since this example consists of a single component, no listeners or adapters are needed.  (3) A SimplePropertyDumper is instantiated to display the state transitions, but there are no statistical listeners.

4)Schedule is set to verbose mode and the run will end at time 15.0.

(5) The run is initialized and executed.

(6) In this case the final number of arrivals is output after the simulation run finishes.

The output for the run is shown in Figure 8-15.  Note that as each Event occurs, the state transition is displayed, followed by the Event being executed and the Event List status at that point in time.

```
simkit.examples.ArrivalProcess.1
        interArrivalTime = Exponential (3.2)

** Event List 0 -- Starting Simulation **
0.000        Run
15.000        Stop
 ** End of Event List -- Starting Simulation **

numberArrivals: null => 0
Time: 0.0000        CurrentEvent: Run [1]
** Event List 0 --  **
0.645        Arrival
15.000        Stop
 ** End of Event List --  **

numberArrivals: 0 => 1
Time: 0.6455        CurrentEvent: Arrival [1]
** Event List 0 --  **
0.648        Arrival
15.000        Stop
 ** End of Event List --  **

numberArrivals: 1 => 2
Time: 0.6485        CurrentEvent: Arrival [2]
** Event List 0 --  **
2.801        Arrival
15.000        Stop
 ** End of Event List --  **
```

```
numberArrivals: 2 => 3
Time: 2.8010        CurrentEvent: Arrival [3]
** Event List 0 --  **
9.292        Arrival
15.000        Stop
 ** End of Event List --  **

numberArrivals: 3 => 4
Time: 9.2922        CurrentEvent: Arrival [4]
** Event List 0 --  **
15.000        Stop
19.982        Arrival
 ** End of Event List --  **

Time: 15.0000        CurrentEvent: Stop [1]
** Event List 0 --  **
19.982        Arrival
 ** End of Event List --  **

At time 15.0000 there have been 4 arrivals
```

**Figure 8-15. Output of Verbose ArrivalProcess Execution**

## 8.8.   Array State Variables

## 8.9.   Subclassing Simkit Components

One of the advantages of Object-Oriented programming is the ability to subclass existing classes and utilize the power of inheritance. While many Simkit components are best implemented via a direct subclass of SimEntityBase or BasicSimEntity, there are some circumstances when it is advantageous to subclass an existing Simkit component. These circumstances are, of course, when there is an existing component that does most of what is desired, and a slight modification is needed. The desired modifications are typically of two types: (1) Additional events and variables are needed; and/or (2) Existing events require different or additional functionality.

Consider the modeling functionality of the arrival of customers modeled as Entities, to be used by a queueing component that processes Entities, such as the EntityServer component in Figure 6-2 instead of simply counting, as with the SimpleServer component of Figure 5-3. One way is by way of the EntityCreator component, shown in Figure 6-1, together with an ArrivalProcess, connecting the listeners as shown in Figure 6-3. Alternatively, the ArrivalProcess can be subclassed with the resulting component adding the functionality of instantiating the Entity object and scheduling the Arrival(Entity) event. A portion of the code is shown in Figure 8-16. Since the superclass Arrival functionality is still desired (incrementing the numberArrivals state variable and scheduling the next Arrival event), there is first a call to the doArrival() method in the superclass. It must be emphasized that this is the *only* situation when writing simulation components that a 'do' method should be directly invoked; all other times waitDelay(. . .) must be used. Next, the Arrival(Entity) event is scheduled with the new Entity object as its argument.

```
public class EntityArrivalProcess extends ArrivalProcess {

. . . // Constructors omitted

    @Override
    public void doArrival() {
        super.doArrival();
        waitDelay("Arrival", 0.0, new Entity());
    }
}
```

**Figure 8-16. Code for Subclassing ArrivalProcess**

Note that the only omitted code in Figure 8-16 consists of the two constructors (and comments). Also, the `@Override` annotation reinforces that the `doArrival()` method is overridden.

In other situations, the functionality of the superclass 'do' method must be completely replaced. In those cases, there would be no call to the superclass' 'do' method.

<TODO: give example>

Great care must be taken when overloading 'do' methods with arguments. Specifically, do *not* overload 'do' methods with arguments that could be ambiguous. For example, having `doAnEvent(Number)` in a superclass and `doAnEvent(int)` in the subclass should be avoided.

# 9.     Random Variate Generation

## 9.1.     Introduction

This section is a basic introduction to random variate generation, or more accurately, pseudo-random variate generation. It is intended to be an elementary introduction, so there will be many areas not covered. The reader is encouraged to consult the references at the end for further information and more in-depth coverage of this topic. The primary aim is to give a modeler the basic tools with which to implement the modeling of randomness in a simulation model. The implementation of the algorithms may be done in a conventional programming language, as scripting language, or even a spreadsheet.

## 9.2.     The Basic Problem

For the purpose of developing algorithms, it is assumed that there is some source of independent, identically distributed (iid) Un(0,1) random numbers available in infinite quantity. In implementations, this sequence of numbers will be pseudo-random generated by some appropriate algorithm. The methodology to select, develop, or test such algorithms is a fundamentally different problem than the one addressed here, and there is a wealth of literature on such methods (cite references).

The problem addressed in this note is as follows: given a sequence $\{U_n\}$ of Un(0,1) random numbers, derive a sequence $\{X_n\}$ of random variates having a specified distribution. We will primarily consider the case where the sequence $\{X_n\}$ is iid for some random variable $X$.

The remainder of this note is organized as follows. The following three sections present the primary generic methods for generating random variates (Inverse Transform, Composition, and Acceptance/Rejection Methods). Each section will first give the general algorithm followed by some examples.

## 9.3.     Inverse Transform Method

The Inverse Transform Method is the most basic of all methods and can be considered the primary building block on which the other methods considered here will build.

### 9.3.1.  The Method

The Inverse Transform Method starts with the cumulative distribution function (cdf) of the desired random variate $X$:

(1) $$F_X(y) = \Pr\{X \le y\}$$

The method itself is based on the fact that if U is a Un(0,1) random variable, then $F^{-1}(U)$ is a random variable whose cdf is $F$. That is,

(2) $$\Pr\{F_X^{-1}(U) \le y\} = \Pr\{X \le y\}.$$

Some care must be made in defining the inverse cdf since the cdf may have jumps (as the case with discrete random variables) or regions where the cdf is "flat" (also the case with all discrete random variables, but also true for many other probability distributions).

Equation (2) can be applied to the entire sequence $\{U_n\}$ giving an immediate solution to the problem, namely $\left\{F_X^{-1}(U_n)\right\}$ being a sequence of iid random variables each having the desired cdf $F_X$. This may be implemented using the following method.

**(3)**                                    **Generate U ~ Un(0,1)**

                             Return $F_X^{-1}(U)$

This method returns a random variable having cdf $F_X$.

Instead of the cdf, the complementary cdf (ccdf) may be used instead. The ccdf is defined to be

(4)                                    $\overline{F}_X(y) = \Pr\{X > y\}.$

For some distributions using $\overline{F}_X^{-1}$ instead of $F_X^{-1}$ may be more convenient or efficient.

For distributions with "flat" regions (i.e. segments with probability of zero) the inverse must be defined carefully:

(5)                                    $F_X^{-1}(w) = \min\{y : F_X(y) \geq w\}$ [5]

The expression in (5) is used for finding the inverse transform method of a discrete distribution, as will be seen in the examples.

## 9.3.2. Examples

We will now work through some examples of the Inverse Transform Method, starting with some simple distributions and moving to some more complicated ones.

### 9.3.2.1. Un(a,b) Distribution

One of the simplest non-trivial distributions to consider is the general Un(a,b). Note that for Un(0,1), the assumption that there is a supply of such random variates makes that problem a trivial one. As with most continuous distributions, the Un(a,b) random variable is typically specified by its pdf, rather than its cdf. Thus, if $X\sim$Un(a,b), its pdf is:

(6)                          $f_X(y) = \begin{cases} \dfrac{1}{b-a} & , a < y < b \\ 0 & , \text{otherwise} \end{cases}.$

In order to apply the Inverse Transform Method, the inverse cdf is required, which in turn requires the cdf.

---

[5] Technically the "min" should be "inf" (or *infimum*).

(7)
$$F_X(y) = \begin{cases} 0 & , y \le a \\ \dfrac{y-a}{b-a}, & a < y < b. \\ 1 & , y \ge b \end{cases}$$

The inverse cdf is the functional inverse of the cdf. Recall that to find the inverse of a function, set the functional expression equal to some variable and solve for the function's argument. Setting $z = F_X(y)$ and solving for $z$, therefore, the inverse cdf for the Un(a,b) distribution is:
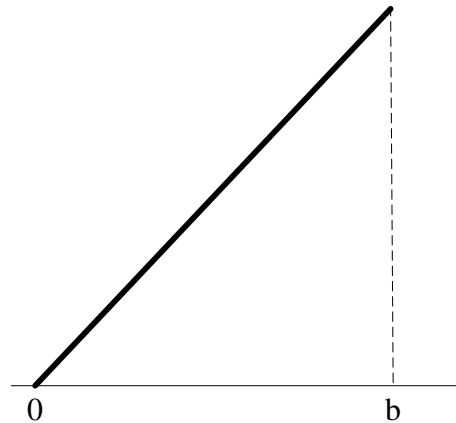
(8)
$$F_X^{-1}(z) = a + z(b-a), 0 < z < 1.$$

Thus the Inverse Transform Method for generating Un(a,b) random variates is given in ..

(9)
Generate U ~ Un(0,1)

Return $a + U(b-a)$

### 9.3.2.2. Triangle(0,b,b) Distribution

The simplest form of the triangle distribution has a pdf as shown in Figure 9-1



**Figure 9-1. Triang(0,b,b) Pdf**

Using the fact that the area of a pdf is 1, the equation of the pdf in Figure 9-1 is

(10)
$$f_X(y) = \begin{cases} \dfrac{2}{b^2} y & , \quad 0 < y \le b \\ 0 & , \quad \text{otherwise} \end{cases}$$

The cdf is therefore

(11)
$$F_X(y) = \begin{cases} 0 & , \quad y < 0 \\ \dfrac{y^2}{b^2} & , \quad 0 < y \le b \\ 1 & , \quad y \ge b \end{cases}$$

The inverse cdf therefore is

(12) $$F_X^{-1}(w) = b\sqrt{w}, \ 0 < w < 1$$

Note that the positive root must be chosen when inverting the cdf because the values generated must be in the interval $(0, b)$. Therefore the method for generating triang(0,b,b) is:

(13)
Generate $U \sim Un(0,1)$

Return $b\sqrt{U}$

### 9.3.2.3. Triang(-b,0,-b) Distribution

The triang(-b,0,-b) distribution has pdf

(14) $$f_X(y) = \begin{cases} -\dfrac{2}{b^2}y & , \quad -b < y \le 0 \\ 0 & , \quad \text{otherwise} \end{cases}$$

With cdf

(15) $$F_X(y) = \begin{cases} 0 & , \quad y < -b \\ 1 - \dfrac{y^2}{b^2} & , \quad -b < y \le 0. \\ 1 & , \quad y \ge 0 \end{cases}$$

The inverse cdf is found to be

(16) $$F_X^{-1}(w) = -b\sqrt{1-w}, \ 0 < w < 1$$

Note that in this case the negative root had to be chosen because the generated values have to be negative, The method is therefore

(17)
Generate $U \sim Un(0,1)$

Return $-b\sqrt{1-U}$

### 9.3.2.4. Exponential (λ) Distribution

The exponential distribution can be parameterized by either the mean or the rate (1/mean). Typically for simulation purposes the mean parameterization is more convenient, and that is the one we shall use. The Exponential(λ) distribution has pdf

(18) $$f_X(y) = \begin{cases} \dfrac{1}{\lambda}e^{-y/\lambda}, & y \ge 0 \\ 0, & y < 0 \end{cases}$$

The cdf is

(19) $$F_X(y) = \begin{cases} 1 - e^{-y/\lambda} & y \ge 0 \\ 0 & y < 0 \end{cases}$$

For $0 < w < 1$, setting $w = F_X(y)$ and solving for w gives $F_X^{-1}(w) = -\lambda \ln(1-w)$. The method is thus

(20)
$$\text{Generate } U \sim Un(0,1)$$
$$\text{Return } - \lambda \ln(1 - U)$$

Often the complementary cdf is used instead: $F_X^{-1}(w) = -\lambda \ln(w)$.

### 9.3.2.5. More than One Functional Form

When inverting the cdf, care must be taken when obtaining the break points for the inverse cdf when there are multiple functional forms for different ranges of values for the random variable. If the cdf is of the form

(21)
$$F_X(y) = F_{X_i}(y), b_{i-1} < y \le b_i, i = 1,\ldots n$$

where $-\infty = b_0 < b_1 < \cdots < b_n = \infty$, then the inverse cdf is given by:

(22)
$$F_X^{-1}(w) = F_{X_i}^{-1}(w), F_{X_i}(b_{i-1}) < w \le F_{X_i}(b_i), i = 1,\ldots n.$$

The generic method in this case is therefore:

(23)
$$\text{Generate } U \sim Un(0,1)$$
$$\text{Return } F_{X_i}^{-1}(U), \text{ where } F_{X_i}(b_{i-1}) < U \le F_{X_i}(b_i)$$

As an example, consider the following pdf:

(24)
$$f_X(y) = \begin{cases} \dfrac{3}{8} & , \quad 1 < y \le 3 \\ \dfrac{1}{4} & , \quad 3 < y \le 4 \\ 0 & , \quad \text{otherwise} \end{cases}.$$

The corresponding cdf is:

(25)
$$F_X(y) = \begin{cases} 0 & , \quad y \le 1 \\ \dfrac{3}{8}(y-1) & , \quad 1 < y \le 3 \\ \dfrac{y}{4} & , \quad 3 < y \le 4 \\ 1 & , \quad y > 4 \end{cases}.$$

The inverse cdf is therefore

(26)
$$F_X^{-1}(w) = \begin{cases} \dfrac{8}{3}w + 1 & , \quad 0 < w \le \dfrac{3}{4} \\ 4w & , \quad \dfrac{3}{4} < w < 1 \end{cases}.$$

The method is therefore:

Generate $U \sim Un(0,1)$

$\text{If} \left( U < \dfrac{3}{4} \right),$

(27)

$\text{Return } \dfrac{8}{3}U + 1$

Else

$\text{Return } 4U$

### 9.3.2.6. Functions of Random Variables

It is evident that if there is a method of generating a random variate $X$, and the desired distribution is $g(X)$, then first $X$ can be generated and $g(X)$ returned. Specifically, if the function $g$ is invertible, then the inverse cdf of $g(X)$ is $g\left(F_X^{-1}(w)\right)$.

For example, a linear transformation $(c - a)X + a$ can be applied to the triang(0,1,1) distribution to obtain the triang(a,c,c) distribution, and the transformation $(b - c)X + c$ can be applied to the triang(-1,0,-1) distribution to obtain the triang(c,b,c) distribution. The respective inverse cdf's are

(28)

$$a + (c - a)\sqrt{w}, 0 < w < 1, \text{ for triang}(a, c, c)$$

$$b - (b - c)\sqrt{1 - w}, 0 < w < 1 \text{ for triang}(c, b, c)$$

### 9.3.2.7. Triangle(a,b,c) Distribution

The general triangular distribution has pdf
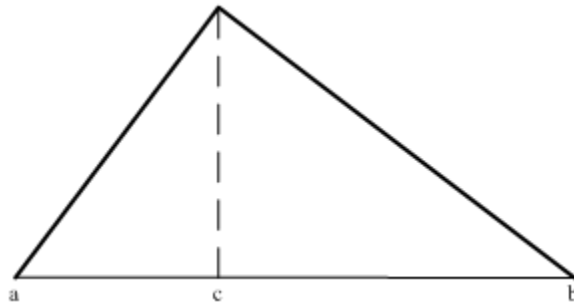


**Figure 9-2. Triangle(a,b,c) PDF**

with functional form:

(29)

$$f_X(y) = \begin{cases} \dfrac{2(y - a)}{(b - a)(c - a)} & , \quad a < y \le c \\ \dfrac{2(b - y)}{(b - a)(b - c)} & , \quad c < y \le b \\ 0 & , \quad \text{otherwise} \end{cases}$$

9-6

The cdf is easily found to be:

(30)
$$F_X(y) = \begin{cases} 0 & , & y < a \\ \dfrac{(y-a)^2}{(b-a)(c-a)} & , & a \le y < c \\ 1 - \dfrac{(b-y)^2}{(b-a)(b-c)} & , & c \le y < b \\ 1 & , & y \ge b \end{cases}.$$

The inverse cdf is therefore

(31)
$$F_X^{-1}(w) = \begin{cases} a + \sqrt{w(b-a)(c-a)} & , & 0 < w \le \dfrac{c-a}{b-a} \\ b - \sqrt{(1-w)(b-a)(b-c)} & , & \dfrac{c-a}{b-a} < w < 1 \end{cases}.$$

The Inverse Transform Method is therefore:

(32)
Generate $U \sim Un(0,1)$
If $(U < (c-a)/(b-a)$
    Return $a + \sqrt{U(b-a)(c-a)}$
Else
    Return $b - \sqrt{(1-U)(b-a)(b-c)}$

### 9.3.2.8.Bernoulli(p) Distribution

The Inverse Transform Method can be applied to discrete distributions as long as the functional form of the inverse cdf is defined carefully. The cdf for the Bernoulli distribution is

(33)
$$F_X(y) = \begin{cases} 0 & , & y < 0 \\ 1-p & . & 0 \le y < 1 \\ 1 & , & y \ge 1 \end{cases}.$$

The inverse cdf is found by applying Equation (5); for $0 < w < 1$:

(34)
$$F_X^{-1}(w) = \begin{cases} 0 & , & 0 < w \le 1-p \\ 1 & , & 1-p < w < 1 \end{cases}.$$

The method is therefore:

(35)
Generate $U \sim Un(0,1)$
If $U < 1-p$
    Return 0
Else
    Return 1

### 9.3.2.9. Geometric(p) Distribution

The Geometric distribution arises from a sequence of Bernoulli trials. The variable itself is the number trials until a '1' is generated. The cdf is given by:

$$(36) \qquad F_X(y) = \begin{cases} 1 - (1-p)^{\lfloor y \rfloor} & y \geq 0 \\ 0 & y < 0 \end{cases},$$

where $\lfloor y \rfloor$ is the larger integer $\leq y$. The inverse cdf is therefore given by:

$$(37) \qquad F_X^{-1}(w) = \left\lceil \frac{\ln(1-w)}{\ln(1-p)} \right\rceil,$$

where $\lceil z \rceil$ is the smallest integer $\geq z$. The inverse transform method for generating a Geometric(p) random variate is therefore:

$$\text{Generate } U \sim Un(0,1)$$

$$(38) \qquad \text{Return } \left\lceil \frac{\ln(1-U)}{\ln(1-p)} \right\rceil \quad .$$

### 9.3.3. General Discrete Distributions

In general, for a discrete probability distribution with mass function given by

$$(39) \qquad p_X(y) = \Pr\{X = y\}, y \in S$$

where S is the set of possible values of $X$ (which need not necessarily be integers).

The cdf is given by

$$(40) \qquad F_X(y) = \sum_{\{x \in S: z \leq y\}} p_X(z)$$

And the inverse cdf is therefore

$$(41) \qquad F_X^{-1}(w) = \min\left\{ y \in S : \sum_{\{z \in S: z \leq y\}} p_X(z) \geq w \right\}.$$

If the elements or S are specified by $S = \{y_0, y_1, \ldots\}$ where $y_0 < y_1 < \ldots$ and S can be either finite or infinite, then the inverse cdf can be given as

$$(42) \qquad F_X^{-1}(w) = \left\{ y_J \in S : \sum_{i=0}^{J-1} p_X(y_i) \leq w < \sum_{i=0}^{J} p_X(y_i) \right\}.$$

The form of the inverse cdf in (42)(42) can be used for a generic inverse transform method for discrete random variates.

### 9.3.4. Functions of Random Variates

Functions of random variates can be easily generated by first generating the underlying random variates and then applying the function. If $g(X_0, \ldots, X_n)$ is a function of random variates $X_0, \ldots, X_n$ and it is known how to generate $X_0, \ldots, X_n$, then a method for generating $g(X_0, \ldots, X_n)$ random variates is as follows:

(43)
$$\begin{array}{l} \text{Generate } X_1, \ldots, X_n \\ \text{Return } g(X_1, \ldots, X_n) \end{array}.$$

While this may seem trivial, it is actually quite useful. One common usage is a linear transformation of a single random variable, having the effect of "stretching" and "shifting" it. Thus, if a method for generating the random variable $X$ is known, then to generate $aX + b$:

(44)
$$\begin{array}{l} \text{Generate } X \sim F_X \\ \text{Return } aX + b \end{array}.$$

## 9.4.  Composition Method

Sometimes a cdf can be expressed as a mixture of other cdf's, which are typically simpler in form. That is, there are cdf's $F_{X_0}, F_{X_1}, F_{X_2}, \ldots$ for which there are known algorithms for generation, and a set of non-negative values $p_0, p_1, p_2, \ldots$ with $\sum_i p_i = 1$ for which

(45)
$$F_X(y) = \sum_i p_i F_{X_i}(y).$$

The strategy is then to choose one of the distributions by selecting an index at random and then generating a *single* variate from the chosen distribution.

The typical use when devising a method to generate from a given distribution might be more accurately called "decomposition" because the approach is to decompose a complicated distribution into a number of simpler parts. In order to do that, both the distributions $F_X$ and the weights $p_i$ must be found. When the desired distribution is continuous (i.e. has a pdf) then this can be accomplished by "slicing" the pdf into various pieces. Each piece must be then proportional to a pdf with a known method for generation. By differentiating (45), in the case of a continuous distribution the decomposition can be written:

(46)
$$f_X(y) = \sum_i p_i f_{X_i}(y)$$

where $f_{X_0}, f_{X_1}, \ldots$ are the respective pdf's.

### 9.4.1. The Method

The general composition method is therefore shown in (47):

$$\text{Generate index } I \text{ with } \Pr\{I = i\} = p_i$$

$$\text{Generate } Y \text{ according to cdf } F_{X_I}$$

$$\text{Return } Y$$

Note that only *one* of the various cdf's is generated from each time the algorithm is executed. Also, most of the examples will involve continuous distributions, so the pdf will typically be used instead of the cdf. Also, note that the composition method will always require at least two uniform random variates – one to be used to choose the index and another to generate from the selected distribution itself. Depending on the exact method used, of course, more uniform variates may be needed.

Also, it turns out that often the exact functional form of the desired distribution is not necessary, as long as the component distributions are readily identified from their shape.

The most common use of the Composition method is in conjunction with the Acceptance/Rejection method, described in the following section. However, to illustrate the technique, it is useful to give some examples.

### 9.4.2. Examples

### 9.4.2.1. Mixture of Two Uniforms

Consider the following pdf:

(48)
$$f_X(y) = \begin{cases} \dfrac{3}{8} & 1 < y \le 3 \\ \dfrac{1}{4} & 3 < y \le 4 \\ 0 & \text{Otherwise} \end{cases}.$$

This can be seen to be a mixture of a $Un(1,3)$ random variable (with probability ¾) and a $Un(3,4)$ random variable with probability ¼. Therefore, a composition method is as follows.

$$\text{Generate } U., V \sim Un(0,1) \text{ (independently)}$$

$$\text{if } V < 3/4$$

(49)
$$\qquad \text{Return } 1 + 2U$$

$$\text{Else}$$

$$\qquad \text{Return } 3 + U$$

Note that another decomposition is given by a $Un(1,4)$ with probability ¾ and $Un(1.3)$ with probability ¼. For that decomposition, the method is:
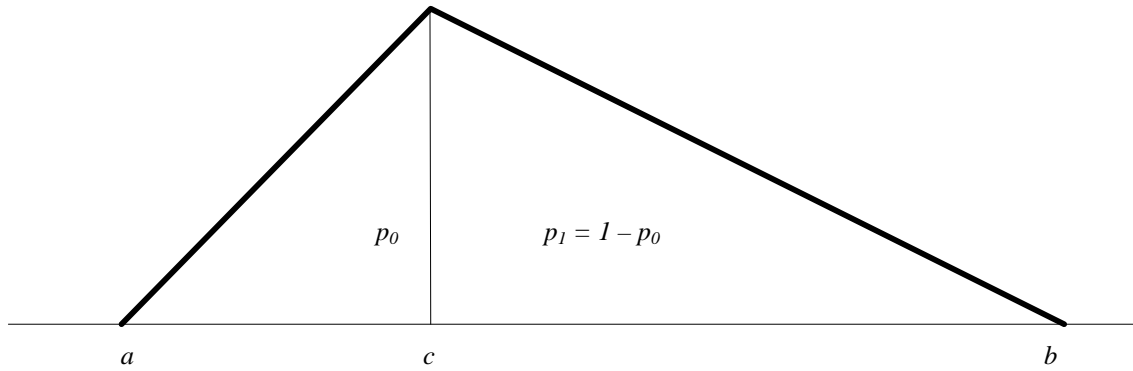
$$\text{Generate } U., V \sim Un(0,1) \text{ (independently)}$$

$$\text{if } V < 3/4$$

(50)
$$\text{Return } 1 + 3U$$

$$\text{Else}$$

$$\text{Return } 1 + 2U$$

### 9.4.2.2. Triangle(a,b,c) by Composition

The general triangle(a,b,c) distribution can be generated by decomposing the pdf into a right triangle (triangle(a,c,c)) and a left triangle (triangle(c,b,c)), as shown in Figure 9-3.



**Figure 9-3. Triangle(a,b,c) PDF**

To find the $p_0$, note that the height of the pdf is $2/(b-a)$ so that $p_0 = (c-a)/((b-a)$. Utilizing the inverse transform method for each piece, the method is:[6]

$$\text{Generate } U, V \sim Un(0,1)$$

$$\text{If } V < \frac{c-a}{b-a}$$

(51)
$$\text{Return } a + (c-a)\sqrt{U} \quad .$$

$$\text{Else}$$

$$\text{Return } b - (b-c)\sqrt{1-U}$$

In this case the method is superfluous given the existence of an inverse transform method for the same random variable. However, it is a useful example of how a "complicated" pdf can be sliced into simpler ones.

### 9.4.2.3. Right Wedge(a,b,h)

A right wedge (a,b,h) distribution has a pdf shown in Figure 9-4.[7] The pfd has been decomposed into Un(a,b) and a triangle(a,b,b) variates.

---

[6] $U$ and $V$ are independently generated.
[7] The parameter $h$ is the height of the short side.
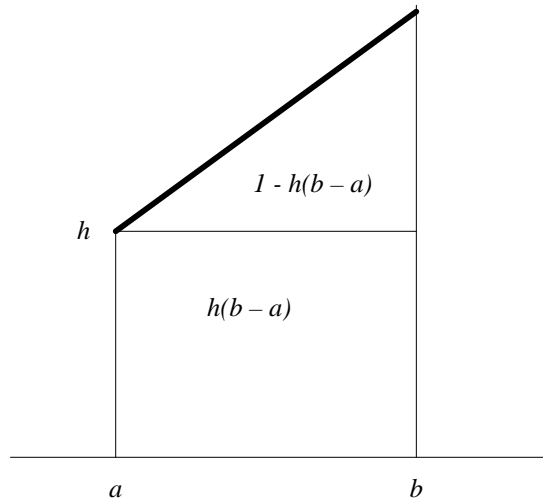
**Figure 9-4. Right Wedge PDF[8]**

The weights for the two pieces are easily determined by simple geometry. The method immediately is seen to be

$$\text{Generate } U, V \sim Un(0,1)$$
$$\text{If } V < h(b-a)$$
(52) $$\quad \text{Return } a + (b-a)U$$
$$\text{Else}$$
$$\quad \text{Return } a + (b-a)\sqrt{U}$$

Note that the triangle density is not given by the part that is "hanging" at height $h$, but rather is obtained by first dropping the function down to the horizontal axis after removing the rectangle (like the game of Tetris). In general, the shape of the pdf obtained from a piece of the original pdf requires a similar "dropping down." The "slice" need not be horizontal or vertical. For example, the right wedge variate can be decomposed as two triangle distributions by an angled "cut," as shown in Figure 9-5.

---

[8] Note that only the heavy line is the actual pdf; the thin lines are just for clarity. Note also that $h < 1/(b-a)$ in order for Figure 9-4 to be an accurate depiction of the pdf.
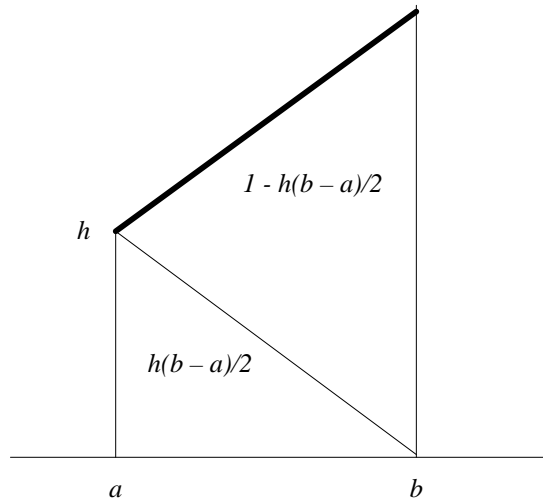
**Figure 9-5. Another Decomposition of the Right Wedge**

The fact that one piece is a triangle(a,b,a) with probability $h(b-a)/2$ is evident from the geometry of Figure 9-5. When that piece is removed, the graph may be thought to look something like Figure 9-6.
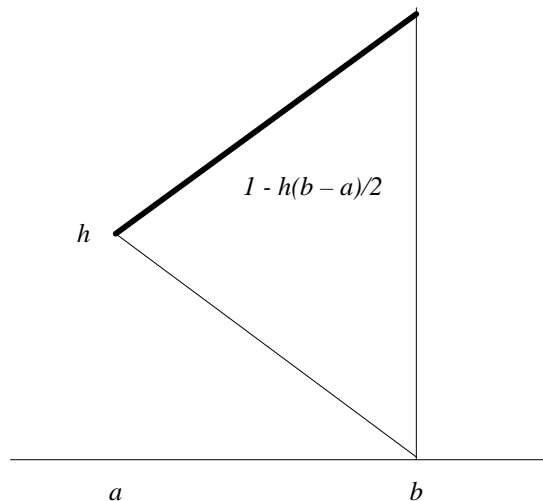


**Figure 9-6. Right Wedge with Left Triangle "Removed"**

However, the graph in Figure 9-6 the remainder of the original pdf is not the graph of a function, since there is nothing corresponding to having pieces "missing" below it. To adjust the remainder requires "dropping" the triangle in Figure 9-6 until it "sits" on the horizontal axis. The result is proportional to a triangle(a,b,b) pdf, as shown in Figure 9-7.
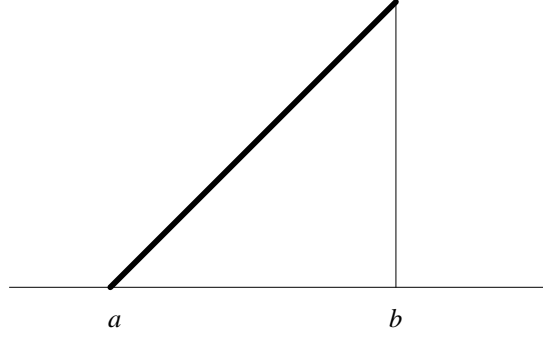
**Figure 9-7. Proportional to Triangle (a,b,b) –
After Removal of Left Triangle and "Dropped" to Horizontal Axis**

Note that the unscaled "remainder" in Figure 9-7 is not a pdf because its area is $1 - h(b - a) < 1$. However, when divided by this quantity, the result is the triangle(a,b,b) pdf. Since the shape of the two pdf are recognized and there are methods to generate from each, the respective areas of the pieces can be used as the corresponding weights. The second method for generating from the wedge is therefore:

(53)

$$\text{Generate } U, V \sim Un(0,1)$$
$$\text{If } V < h(b - a)/2$$
$$\text{Return } a + (b - a)\sqrt{U} \ .$$
$$\text{Else}$$
$$\text{Return } b - (b - a)\sqrt{1 - U}$$

This example is for illustrative purposes, since the method in (53) requires more expensive computations than the one in (52). Specifically, for the method in (52), with probability $h(b - a)$ the computation of a square root is not required, whereas in (53) a square root must be computed regardless.
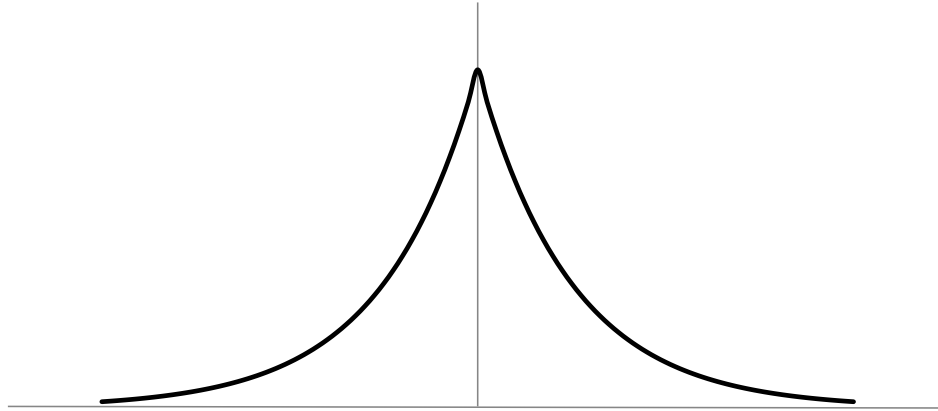
### 9.4.2.4. Laplace (β) Distribution

The Laplace(β) distribution is a double-sided exponential random variable. Its pdf is given by:

(54)

$$f_X(y) = \frac{2}{\beta} e^{-|y|/\beta}, -\infty < y < \infty.$$

A graph of the pdf is shown in Figure 9-8.

**Figure 9-8. Graph of Laplace pdf**

Note that the left piece of the pdf is proportional to the pdf for $-X$, where $X \sim \text{Exp}(\beta)$, so the Laplace($\beta$) distribution is seen as a mixture of an Exp($\beta$) and $-$Exp($\beta$) with equal probabilities (that is, $p_0 = p_1 = 1/2$). The method is therefore,[9]

(55)

$$\text{Generate } U, V \sim Un(0,1)$$
$$\text{If } V < 1/2$$
$$\quad \text{Return} - \beta \ln(U) \qquad .$$
$$\text{Else}$$
$$\quad \text{Return } \beta \ln(U)$$

---

[9] Or 1-*U* could be used instead.

## 9.5.    Acceptance/Rejection Method

Often the desired distribution cannot be inverted or decomposed into parts that can be generated. In those situations, the Acceptance/Rejection method can often be applied.

### 9.5.1.  The Method

Suppose there is a function $t(y)$ that

- Majorizes $f_X(y)$ - that is, $f_X(y) \le t(y)$ for all $y$.

- Has finite area – that is, $\int_{-\infty}^{\infty} t(y)dy = c < \infty$

- Is proportional to a pdf from which we can generate. That is, there is a known method for generating random variates having pdf $\dfrac{t(y)}{c}$.

Then the following method produces a random variate with pdf $f_X(y)$:

7.  Generate $Y$ having pdf $\dfrac{t(y)}{c}$

8.  Generate $U \sim Un(0,1)$

9.  If $U < \dfrac{f_X(Y)}{t(Y)}$ Return (accept) $Y$; Else discard (reject) $Y$ and repeat from step 1.

An alternative way of describing the method that is more amenable to an indefinite control flow is:

do {

   Generate $Y$ having pdf $\dfrac{t(y)}{c}$

   Generate $U \sim Un(0,1)$

} while $\left( U > \dfrac{f_X(Y)}{t(Y)} \right)$

Return $Y$

Note that the probability of accepting in any give iteration is $1/c$ where $c$ is the area under the majoring function $t(y)$. This value can be considered a measure of how a given algorithm exploits the uniform random numbers used, and is sometimes called the *efficiency* of the method.
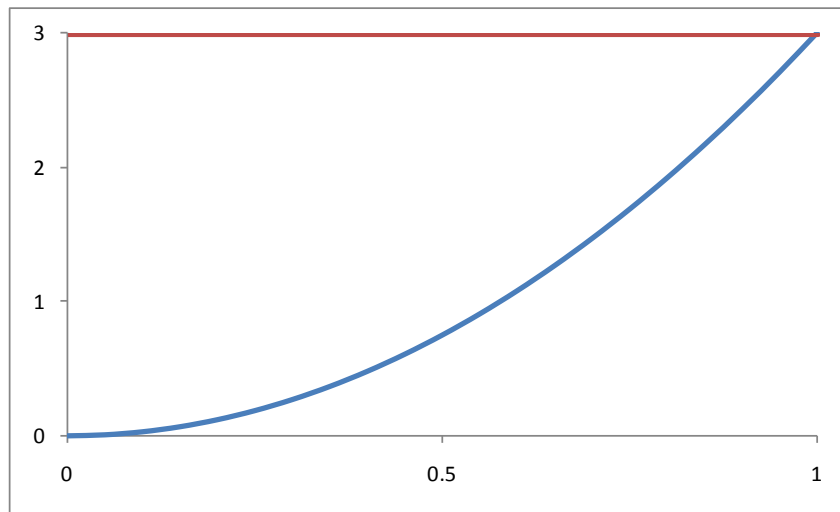
### 9.5.2. Examples

#### 9.5.2.1. Beta(3,1)

The Beta(3.1) distribution has pdf

(56)
$$f_X(y) = \begin{cases} 3y^2 & 0 < y < 1 \\ 0 & \text{Otherwise} \end{cases}$$

A majorizing function is

(57)
$$t(y) = \begin{cases} 3 & 0 < y < 1 \\ 0 & \text{Otherwise} \end{cases}$$



**Figure 9-9. Beta(3,1) with Uniform Majorizing Function**

The majorizing function is proportional to a Un(0,1) pdf, and the area $c = 3$. The ratio of the two functions is

(58)
$$\frac{f_X(y)}{t(y)} = \frac{3y^2}{3} = y^2, 0 < y < 1,$$

so an acceptance/rejection method for generating from the Beta(3,1) distribution is therefore:

(59)
$$\begin{aligned} &\text{do } \{ \\ &\quad \text{Generate } U, Y \sim Un(0,1) \\ &\} \text{ while } (U > Y^2) \\ &\text{Return } Y \end{aligned}$$

The efficiency of this method is 1/3; that is, 2/3 of the generated random variates will be rejected. This is an extremely low efficiency, and Figure 9-9 suggests that the reason is the large amount of area between the pdf and the majorizing function. This also suggests an alternate

majorizing function that more closely matches the overall shape of the pdf. One such function is a triangle, shown in Figure 9-10.
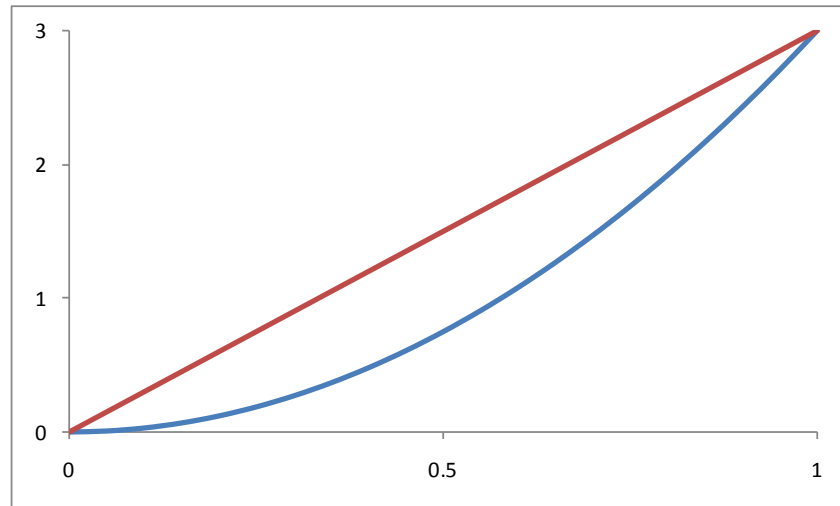


**Figure 9-10. Beta(3,1) with Triangle Majorizing Function**

The majorizing function Now the generated variate will be from a triang(0,1,1) distribution. The ratio of the functions in Figure 9-10 is

(60)
$$\frac{f_X(y)}{t(y)} = \frac{3y^2}{3y} = y, 0 < y < 1$$

So an acceptance/rejection method based on this majorizing function is

(61)
$$
\begin{aligned}
&\text{do \{} \\
&\quad \text{Generate } U, V \sim Un(0,1) \\
&\quad \text{Set } Y = \sqrt{V} \\
&\text{\} while } (U > Y) \\
&\text{Return } Y
\end{aligned}
$$

Alternatively, the following method is equivalent and slightly faster, since it replaces the relatively expensive square root with a square and only computes the square root when there is acceptance:

(62)
$$
\begin{aligned}
&\text{do \{} \\
&\quad \text{Generate } U, Y \sim Un(0,1) \\
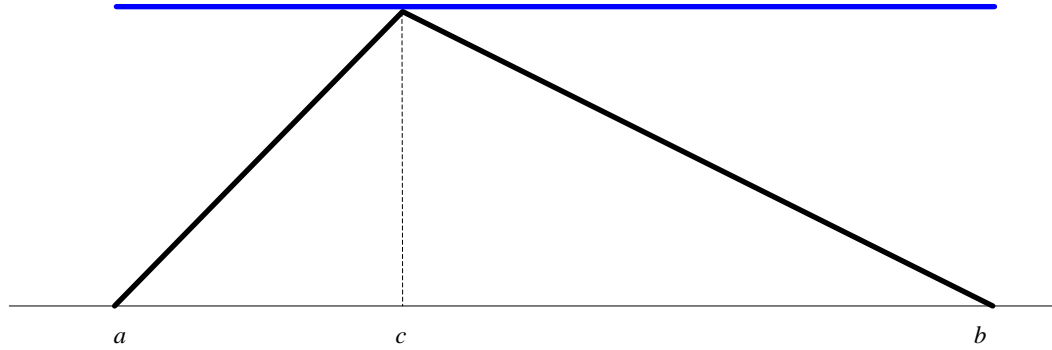&\text{\} while } (U^2 > Y) \\
&\text{Return } \sqrt{Y}
\end{aligned}
$$

The area under the majoring function is 3/2, so efficiency of the method is 2/3, a considerable improvement over the first one.

### 9.5.2.2. Triangle(a,b,c)

Applying the Acceptance/Rejection method to the triangle(a,b,c) pdf, a natural majorizing function is:

$$(63) \qquad t(y) = \begin{cases} \dfrac{2}{b-a} & a < y < b \\ 0 & \text{Otherwise} \end{cases}$$

(see Figure 9-11).

**Figure 9-11. Triangle(a,b,c) with Uniform Majorizing Function**

The ratio of the pdf to the majorizing function is:

$$(64) \qquad \frac{f_X(y)}{t(y)} = \begin{cases} \dfrac{y-a}{c-a} & a < y < c \\ \dfrac{b-y}{b-c} & c \leq y < b \\ 0 & \text{Otherwise} \end{cases}.$$

The method is therefore:

$$(65)
\begin{aligned}
&\text{do } \{ \\
&\quad \text{Generate } U, V \sim Un(0,1) \\
&\quad \text{Set } Y = a + (b-a)V \\
&\text{\} while } (Y < c \,\&\, U > (Y-a)/(c-a) \,\| \\
&\qquad\qquad Y \geq c \,\&\, U > (b-Y)/(b-c)) \\
&\text{Return } Y
\end{aligned}$$

The area under the majorizing function is 2, so the efficiency is ½.

### 9.5.2.3. Beta(2,2)

The Beta(2,2) distribution has pdf

$$(66) \qquad f_X(y) = \begin{cases} 6y(1-y) & 0 < y < 1 \\ 0 & \text{Otherwise} \end{cases}$$

Although the cdf can be determined in closed form, it is in the form of a cubic, the inversion of which is possible but extremely messy. Thus, Acceptance/Rejection is a reasonable approach because Inverse Transform is not practical. For Beta with higher valued parameters it is not even possible to determine the cdf in closed form.

Since the pdf achieves its maximum of 1.5 at $y=0.5$, a majorizing function of

(67)
$$t(y) = \begin{cases} 1.5 & 0 < y < 1 \\ 0 & \text{Otherwise} \end{cases}$$

is a natural one (see Figure 9-12).
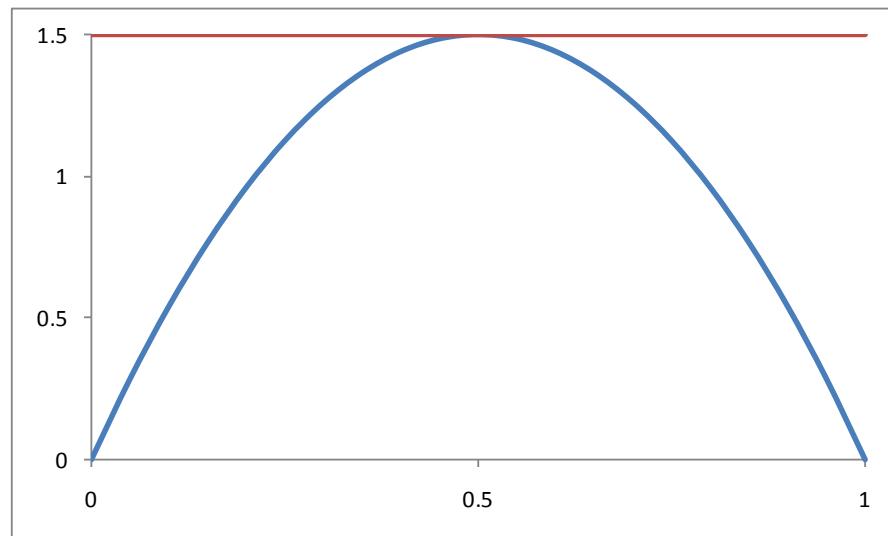


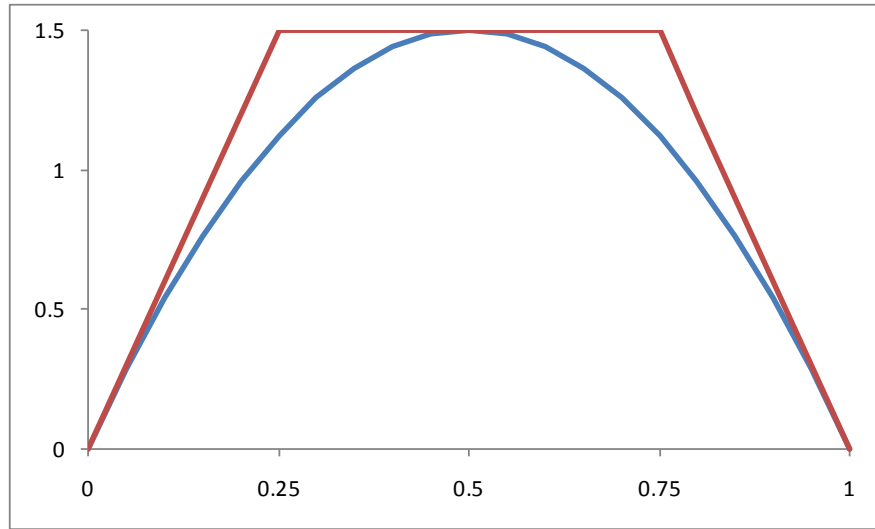**Figure 9-12. Beta(2,2) with Uniform Majorizing Function**

The ratio is

(68)
$$\frac{f_X(y)}{t(y)} = \begin{cases} 4y(1-y) & 0 < y < 1 \\ 0 & \text{Otherwise} \end{cases}.$$

The method is therefore:

(69)
do {
    Generate $U, Y \sim Un(0,1)$
} while $(U > 4Y(1-Y))$
Return $Y$

The area under $t(y)$ is 3/2, leading to an efficiency of 2/3. This leads to the question of whether it can be improved. Consider a majorizing function that is a trapezoid, as shown in Figure 9-13.

**Figure 9-13. Beta(2,2) with Trapezoid Majorizing Function**

The majorizing function is given by

(70)
$$t(y) = \begin{cases} 6y & 0 < y < 0.25 \\ 1.5 & 0.25 \le y < 0.75 \\ 6(1-y) & 0.75 \le y < 1 \\ 0 & \text{Otherwise} \end{cases}$$

And the ratio is

(71)
$$\frac{f_X(y)}{t(y)} = \begin{cases} 1-y & 0 < y < 0.25 \\ 4y(1-y) & 0.25 \le y < 0.75 \\ y & 0.75 \le y < 1 \\ 0 & \text{Otherwise} \end{cases}$$

The composition method will be used to generate the "candidate" random variate $Y$. First the majorizing function needs to be rescaled to be a pdf (i.e. have area of 1). The areas under the three parts of the majorizing function are shown in Figure 9-14.
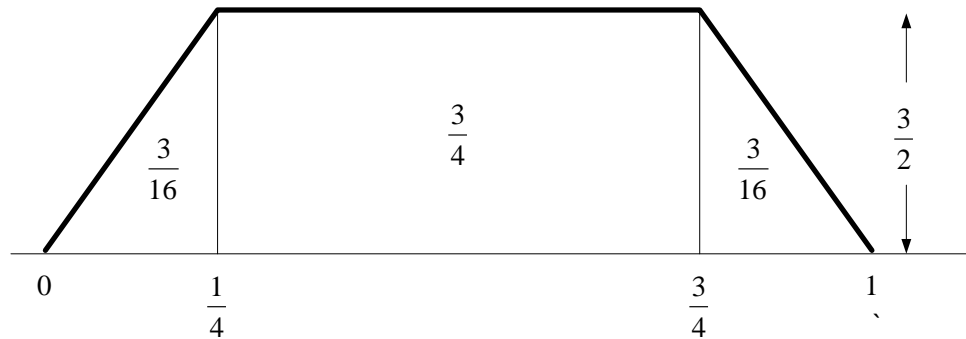


**Figure 9-14. Majorizing Function Areas**

We see from Figure 9-14 that the total area is $\dfrac{3}{16} + \dfrac{3}{4} + \dfrac{3}{16} = \dfrac{18}{16} = \dfrac{9}{8}$, so the efficiency is

seen to be $\dfrac{8}{9} \approx 0.889$. Furthermore, the corresponding pdf and the weights can be determined by dividing each area of the majorizing function by the total area. The result is shown in Figure 9-15.
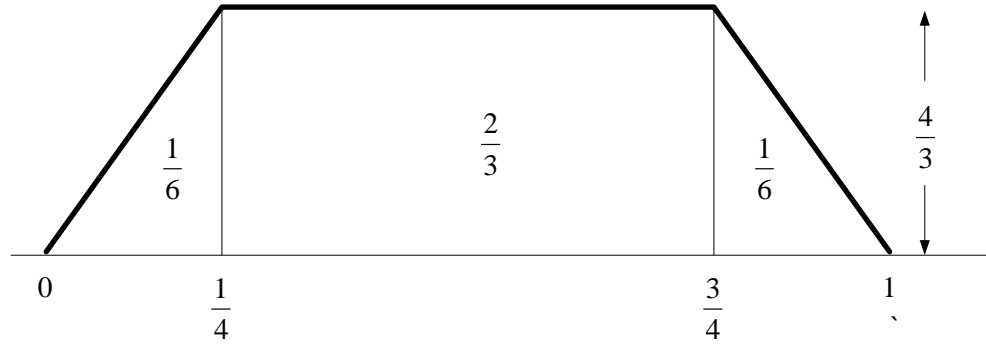


**Figure 9-15. Trapezoid pdf – Scaled to Area of 1**

It is thus seen that the trapezoid pdf is a mixture of triang(0,1/4,1/4) with probability 1/6, Uniform(1/4,3/4) with probability 2/3, and triangle(3/4,1,3/4) with probability 1/6.

Finally, it is noted that the order of generating the pieces can be arbitrarily set, and it is sometimes recommended that it be done in order of decreasing probabilities.

The method is therefore:

(72)

$$
\begin{aligned}
&\text{do } \{ \\
&\quad \text{Generate } U, V, W \sim Un(0,1) \\
&\quad \text{If } W < 2/3 \\
&\qquad Y = \frac{1}{4} + \frac{1}{2}V \\
&\quad \text{Else if } W < 5/6 \\
&\qquad Y = \frac{1}{4}\sqrt{V} \\
&\quad \text{Else} \\
&\qquad Y = 1 - \frac{1}{4}\sqrt{1-V} \\
&\quad \} \text{ while } (\,(Y < 0.25 \;\& \; U > 1 - Y)\| \\
&\qquad\qquad (0.25 \le Y < 0.75 \;\& \; U > 4Y(1-Y))\| \\
&\qquad\qquad (Y \ge 0.75 \;\& \; U > Y)) \\
&\quad \text{Return } Y
\end{aligned}
$$

**9.5.2.4.Normal(0,1)**

      The Normal distribution is perhaps the most commonly used probability model. If $Z$ is a Normal(0,1), then $\mu + \sigma Z$ Normal($\mu, \sigma^2$) random variable. Therefore, if a way to generate a Normal(0,1) is found, then a Normal with any mean and variance can be obtained by this linear transform. The pdf for a standard Normal (with mean 0 and variance 1) is:

(73)
$$f_Z(y) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}, -\infty < y < \infty$$

      Majorizing functions with a finite range, as have been dealt with so far, cannot be used for distributions with tails, such as the Normal distribution. However, it turns out that the Laplace distribution with parameter 1, described previously for general β, can have its pdf be scaled to majorize the Normal(0,1) pdf. That majorizing function is given by:

(74)
$$t(y) = \frac{1}{\sqrt{2\pi}} e^{-|y|+1/2}, -\infty < y < \infty .$$

      A graph of this function majoring the Normal pdf is shown in Figure 9-16.
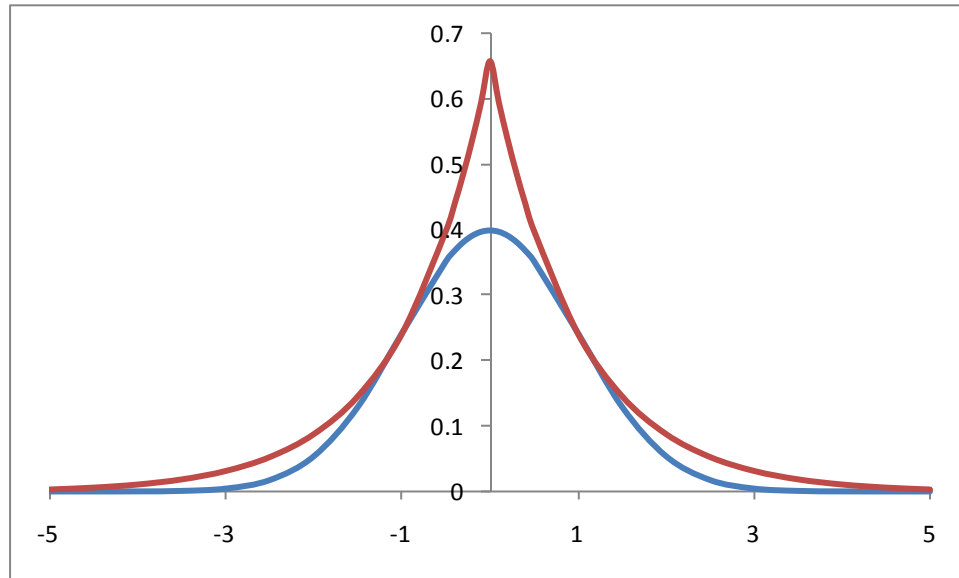


**Figure 9-16, N(0,1) with Laplace Majorizing Function**

      The ratio $f_Z(y)/t(y)$ is seen to be:

(75)
$$\frac{f_Z(y)}{t(y)} = \frac{e^{-y^2/2}}{e^{-|y|+\frac{1}{2}}} = e^{-\frac{1}{2}(y^2-2|y|+1)} = e^{-\frac{1}{2}(|y|-1)^2} .$$

      The Composition method developed previously can be used to generate Laplace random variates for the Acceptance/Rejection method. The overall method is as follows.

```
do {
    Generate U, V, W ~ Un(0,1)
    if W < 1/2
        Y = -ln V
    Else
        Y = ln V
} while (U > e^{-\frac{1}{2}(|Y|-1)^2})
Return Y
```

The area under the majorizing function is $\sqrt{\dfrac{e}{2\pi}} \approx 1.315$ so the efficiency is given by

$\sqrt{\dfrac{2\pi}{e}} \approx 0.760$.

This raises the question of whether the efficiency can be improved. Examination of Figure 9-16 indicates that the portion of the majorizing function in the region of 0 is high relative to the pdf, meaning that a larger proportion of generated values with small absolute value are being rejected. One remedy is to truncate the majorizing function for those values. This results in a majorizing function that is constant for values $y$ with $|y| < 1$, resulting in Figure 9-17
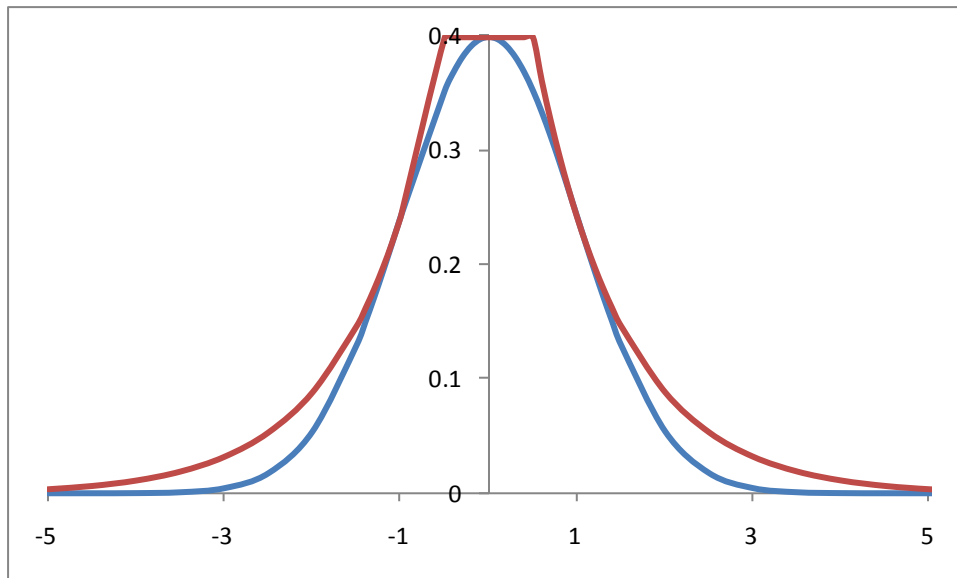


**Figure 9-17. N(0,1) with Truncated Laplace Majorizing Function**

The functional form of this majorizing function is

$$t(y) = \begin{cases} \dfrac{1}{\sqrt{2\pi}} & |y| < \dfrac{1}{2} \\ \dfrac{1}{\sqrt{2\pi}} e^{-|y|+\frac{1}{2}} & |y| \geq \dfrac{1}{2} \end{cases}, \text{ so the ratio is}$$

$$(78) \qquad \frac{f_Z(y)}{t(y)} = \begin{cases} e^{-\frac{1}{2}y^2} & |y| < \frac{1}{2} \\ e^{-\frac{1}{2}(|y|-1)^2} & |Y| \geq \frac{1}{2} \end{cases}.$$

The area under the majorizing function in Figure 9-17 is

$$\frac{1}{\sqrt{2\pi}} + \frac{1}{\sqrt{2\pi}} + \frac{1}{\sqrt{2\pi}} = \frac{3}{\sqrt{2\pi}} \approx 1.197 \quad \text{so the efficiency is} \quad \frac{\sqrt{2\pi}}{3} \approx 0.836. \text{ The method is:}$$

do {

    Generate $U, V, W \sim Un(0,1)$

    If $W < \dfrac{1}{3}$

        Set $Y = -\dfrac{1}{2} + \ln V$

    Else if $W < \dfrac{2}{3}$

        Set $Y = V - \dfrac{1}{2}$

    Else

        Set $Y = \dfrac{1}{2} - \ln V$

    } while $\left( (|Y| < \dfrac{1}{2} \&\& U > e^{-\frac{1}{2}Y^2}) \| \right.$

$(79) \qquad \left. (|Y| \geq \dfrac{1}{2} \&\& U > e^{-\frac{1}{2}(|Y|-1)^2}) \right)$

Return $Y$

## 9.6.  References

1.  Devorye, Luc. *Non-Uniform Random Variate Generation*. Springer-Verlag, New York, NY (1986).
1.  Gentle, James E. *Random Number Generation and Monte Carlo Methods*. Springer, New York, NY (1998).
2.  Law, Averill, and Kelton, David. *Simulation Modeling and Analysis, Third Edition*. McGraw Hill, Boston, MA (2000).

# 10. Implementing and Using Random Variates in Simkit

This section will illustrate how random variates are implemented in Simkit as well as to demonstrate how to implement new random variates and use them as "first-class citizens" in simulation components.

## 10.1. Implementing Random Numbers

Simkit captures basic source of pseudo-randomness in classes that implement the `RandomNumber` interface. The "default" `RandomNumber` algorithm is the Mersenne Twister, and there are a number of other standard algorithms implemented in different classes, such as the Tausworthe, linear congruential, and others.

## 10.2. Implementing Random Variates

Simkit captures the generation of different probability distributions through an interface (`RandomVariate`) and abstract base class (`RandomVariateBase`), as shown in Figure 10-1, which shows two concrete classed (`ExponentialVariate` and `GammaVariate`) in the hierarchy.
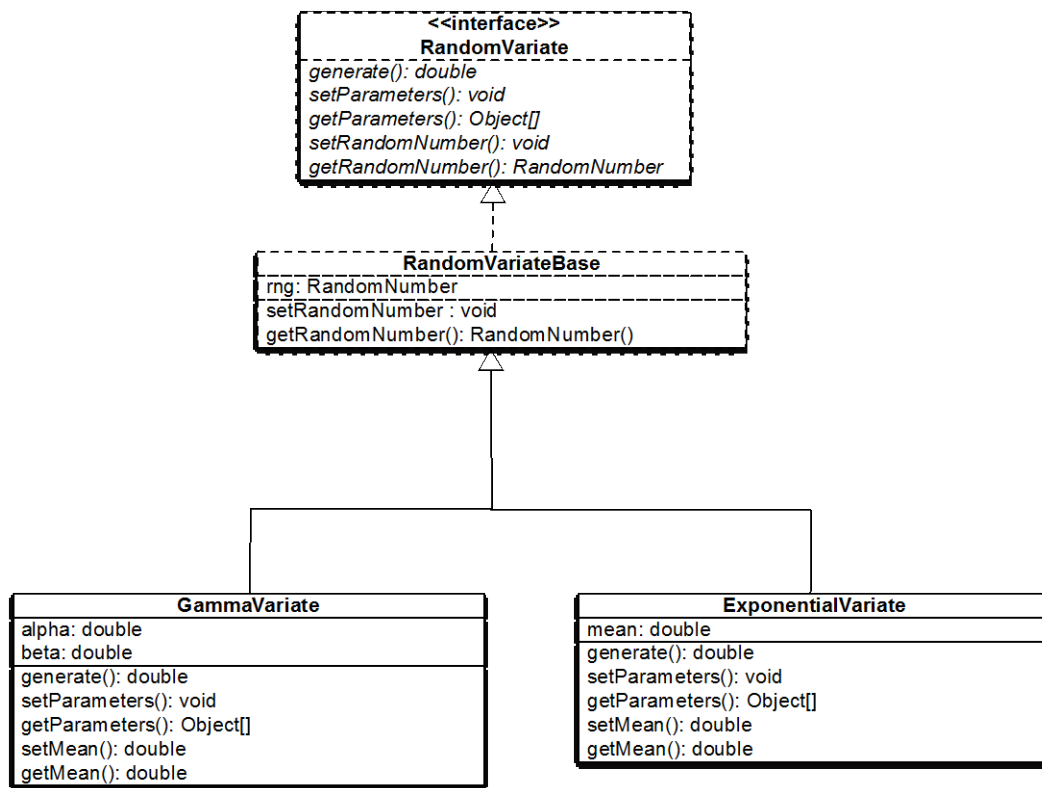


**Figure 10-1. Class and Interface Diagram for RandomVariates**

Concrete classes can either implement the `RandomVariate` interface or subclass `RandomVariateBase`. In addition to implementing the methods of the `RandomVariate` contract, concrete classes must implement necessary parameters as read/write, using the Java conventions for setters and getters. Finally, they are expected to adhere to the JavaBeans standard of providing a zero-parameter constructor.

The "normal" way of obtaining a RandomVariate instance is through a static method call to RandomVariateFactory.getInstance(String, Object...). The first argument is the name of the RandomVariate class, which may be the fully qualified name, the unqualified name (if the class is either in the simkit.random package or in one of the designated search packages), or the name without "Variate". The convention is that a RandomVariate class's name end in "Variate" but that is not required.

When obtained from a call to RandomVariateFactory.getInstance(), the object is instantiated using the zero-argument constructor and the RandomNumber instance set using the RandomVariateFactory's default RandomNumber instance. Finally, the parameters are set using the corresponding setter methods. If the RandomVariateFactory cannot find the name of a class from the first argument, it throws an IllegalArgumentException. By default, The RandomVariateFactory only searches the simkit.random package for the class. However, additional packages may be added by invoking addSearchPackage(String). For example, to allow unqualified class names from the mv3302.random package, invoke RandomVariateFactory.addSearchPackage("mv3302.random").

The RandomVariateFactory uses the same RandomNumber instance for each RandomVariate class obtained using the getInstance() call above This ensures that all generated values will be independent. This behavior may be overridden by specifying a different RandomNumber instance via an overloaded call getInstance(String, RandomNumber, Object...).

# 11. Simple Output Analysis

Output analysis for a simulation model consists of attempting to estimate certain measures associated with the system being modeled. This is done by estimating measures obtained from the simulation model. If the model is a faithful representation of the system in question, then valid estimates of the simulation measures may be inferred to be representative of the corresponding measures in the system itself.

In this section we will explore the simplest version of output analysis, namely estimating a single parameter of the simulation model. This parameter is some fixed, but unknown quantity of the model. For simplicity, we will only examine estimating parameters that are means.

In statistics the two primary estimators are point estimators and interval estimators. Simulation models allow for a third possibility, interval estimators with a desired level of precision.

Statistical analysis typically starts with a random sample from the population in question. This random sample is assumed to be independent and identically distributed (iid). Unfortunately, most raw simulation output data are not iid, so the typical estimators cannot be blindly applied.

As an example of the non-iid nature of raw simulation data, consider the sequence of delays in queue for customers in a multiple server queueing system with $k$ servers. At the start of the simulation replication, the system is empty and idle. The delay in queue for the first k customers will always be exactly 0. Thus, the distribution of the first k delays in queue is constant with a value of 0. The delay in the queue for customer k+1 is also 0 if that customer arrives after the completion of at least one service, and is greater than 0 if none of the first k customers have completed service when the k+1$^{st}$ customer arrives. Therefore, the expected delay in queue for the k+1$^{st}$ customer is greater than 0:

$$(80) \qquad E\big[D_{k+1}\big] = E\left[\min\big(t_{A_1}+t_{S_1}..t_{A_k}+t_{S_k}\big) - t_{A_{k+1}} \Big| t_{A_{k+1}} < \min\big(t_{A_1}+t_{S_1}..t_{A_k}+t_{S_k}\big)\right] > 0,$$

where $\{t_{A_i}\}$ are the interarrival times, $\{t_{S_i}\}$ are the service times, and $\{D_i\}$ are the successive delays in queue. Similarly, the expected delay in queue for the k+2$^{nd}$ customer is different, and so forth. So the sequence of delays in queue $\{D_i\}$ is not identically distributed. If the system has a steady state and is run long enough, however, we expect that the distribution of successive delays in queue would be approximately the same.

However, even in that case, successive delays in queue are not independent. To illustrate this, consider a delay in queue $D_n$ that is longer than "average." The chances are that the next delay in queue $D_{n+1}$ will also be larger than average, since customer n+1 will have spent some time waiting just behind customer n. Similarly, if customer n has a smaller than average delay in queue, the chances are that customer n+1 will also have a smaller than average delay in queue. Thus, intuitively at least, the sequence of delays in queue $\{D_i\}$ are correlated.

## 11.1. Terminating and Non-Terminating Simulations

For our purposes, there are two types of simulations: terminating and non-terminating.

A terminating simulation is one in which there is a natural event or ending criterion for a replication. One example is a combat model in which the first side to reach a certain casualty level is declared to have "lost" the battle. Another might be that an objective has been reached by a certain point in time. A bank might open at 9:00 am and close at 5:00 pm each day. For each of these situations the terminating event is built into the situation. The terminating event might be an artificial one imposed by the analyst. For example, the measure of interest might be for the first 1000 customers arriving to the system.

A key property of measures of terminating simulations is that the initial conditions influence the desired measures, sometimes dramatically. Even if the system in question could potentially reach steady state, the analysis considers all the data from the beginning.

A non-terminating simulation is one for which there is no natural terminating event. For non-terminating simulations, steady state measures are of interest. For example, the steady state expected delay in queue for a multiple server queueing system might be of interest.

The distinguishing characteristic of a steady state measure is that the initial conditions do *not* affect its value. Therefore, any influence of the initial conditions will bias the results, which is why this phenomenon is known as *initialization bias*.

Analyzing the output of simulation models is a rich and broad area, and in this note we are only able to dip our toes in it. For more about output analysis, consult a standard simulation text, such as Law and Kelton (2000) or Banks, Carson, and Nelson (1996).

## 11.2. Estimating a Mean Measure for a Terminating Simulation

For terminating simulations, the execution of the runs and corresponding analysis is simple and straightforward. A number of independent replications are performed using identical starting conditions, and the desired measure from each run either stored or accumulated. Since the standard statistical assumptions (iid) for these data apply, the resulting estimator can be found in the traditional manner. For example, to estimate a mean, a $100(1 - \alpha)$% confidence interval can be determined by the standard approach. Specifically, if $\{X_1, X_2, \ldots, X_n\}$ are the results of $n$ replications of the simulation, an approximate $100(1-\alpha)$% confidence interval is

$$\overline{X}_n \pm \frac{S_n}{\sqrt{n}} t_{n-1,1-\alpha/2}, \tag{1}$$

where $\overline{X}_n$ is the sample average of the n replications, $S_n$ is the sample standard deviation of the n replications, and $t_{n-1,1-\alpha/2}$ is the $(1 - \alpha/2)$th percentile of the Student-t distribution. This of course assumed that n is sufficiently large for the Central Limit Theorem to be appropriate.

Often the quantities $X_i$ from the runs are themselves averages, such as the average delay in the queue, so that $\overline{X}_n$ is an average of averages.

The challenge for terminating simulations lies not so much in the runs and estimation, but in defining exactly what it is being estimated. Typically, the individual data point from a replication is an average, as noted above, but the quantities being averaged are not identically distributed. This leads to the question, what is being estimated by this?

11-2

To continue the example of delay in queue for a terminating simulation, the raw data for each replication are the successive delays for each customer. The average of these over a replication is an observation therefore of the expected average delay in queue over the replication. Alternatively, the output of a combat model might be simply 1 if the Blue side "wins" and 0 if they lose. In that case, the computed confidence interval is an estimate of the probability that Blue wins the battle.

## 11.3. Estimating a Mean Measure for a Simulation in Steady State

As mentioned above, a steady-state mean parameter is more straightforward to define, being a single quantity. For example, the steady state mean delay in queue is a single number. However, obtaining good estimates of that number are more problematic due to two factors: initialization bias and auto-correlation.

In general, statistical bias occurs when an observation's expected value does not equal the quantity it is being used to estimate. Initialization bias exists in most simulations because it is typically not started in steady-state, and therefore the expected value of the initial observations do not equal the steady-state parameter.

For example, Figure 82 shows the difference between the steady-state mean $\mu$ and the time-varying value $\mu(t)$. The initial observations of $\mu(t)$ would be biased for $\mu$.
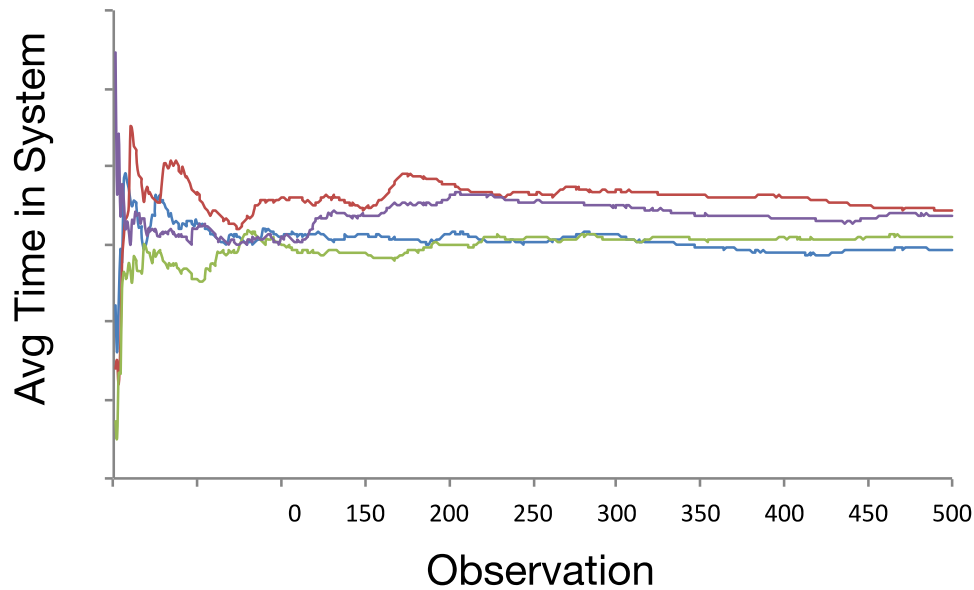


$$\mu(t)$$
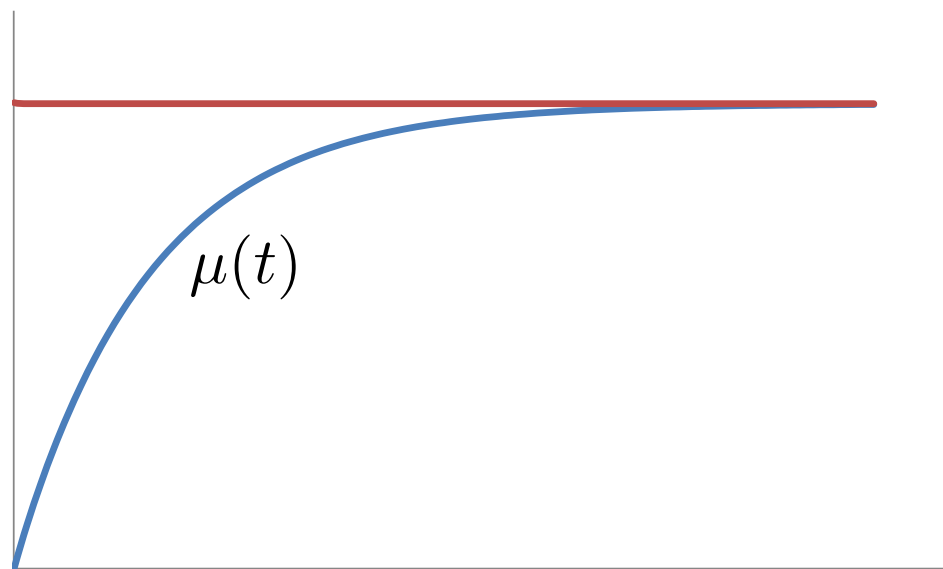
**Figure 2. Initialization Bias[10]**

An example of the running mean output for a simulation over several replications is shown in Figure 83 below.

---

[10] Note that this graph is notional, and that there and many different possible ways initial bias can manifest.

**Figure 3. Running Average of Time in System**

For each observation in the initial transient region, more and more observations will result in more and more accurate estimates of the wrong quantity. In general, there is little relationship between the initial observations and the steady-state values. Therefore, the common approach is to "warm up" the simulation for a period of time and then "truncate" – meaning that the statistics for the initial warm-up period are discarded, and the remaining observations used to construct the estimate of μ.

Obtaining the truncation point rigorously is quite a challenge, and most approaches are rather *ad hoc*; following either a simple or complicated approach, the means are plotted and the truncation point selected based on whether the result has appeared to converge. In this example, if more observations are taken, the resulting graph (Figure 84) suggests that 10,000 observations is a reasonable warm up period.
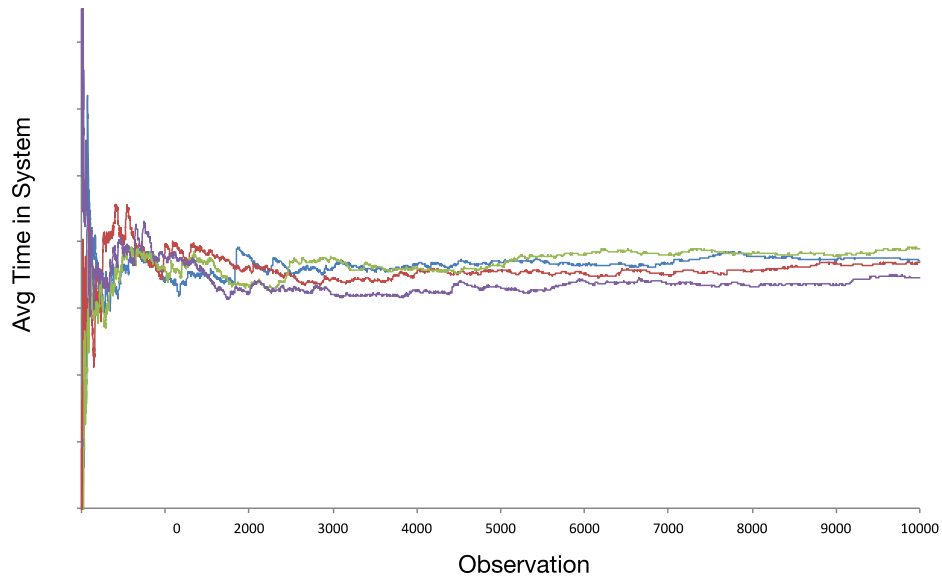
**Figure 4. Determining Truncation Point**

After the truncation point, data are collected and used to estimate the steady-state parameter. Choosing the number of observations after the truncation point is also an *ad hoc* choice. In this example, 10,000 additional observations were taken after the truncation point for four replications, resulting in the following Figure 85. The resulting replications all seem to settle very close to a single value.
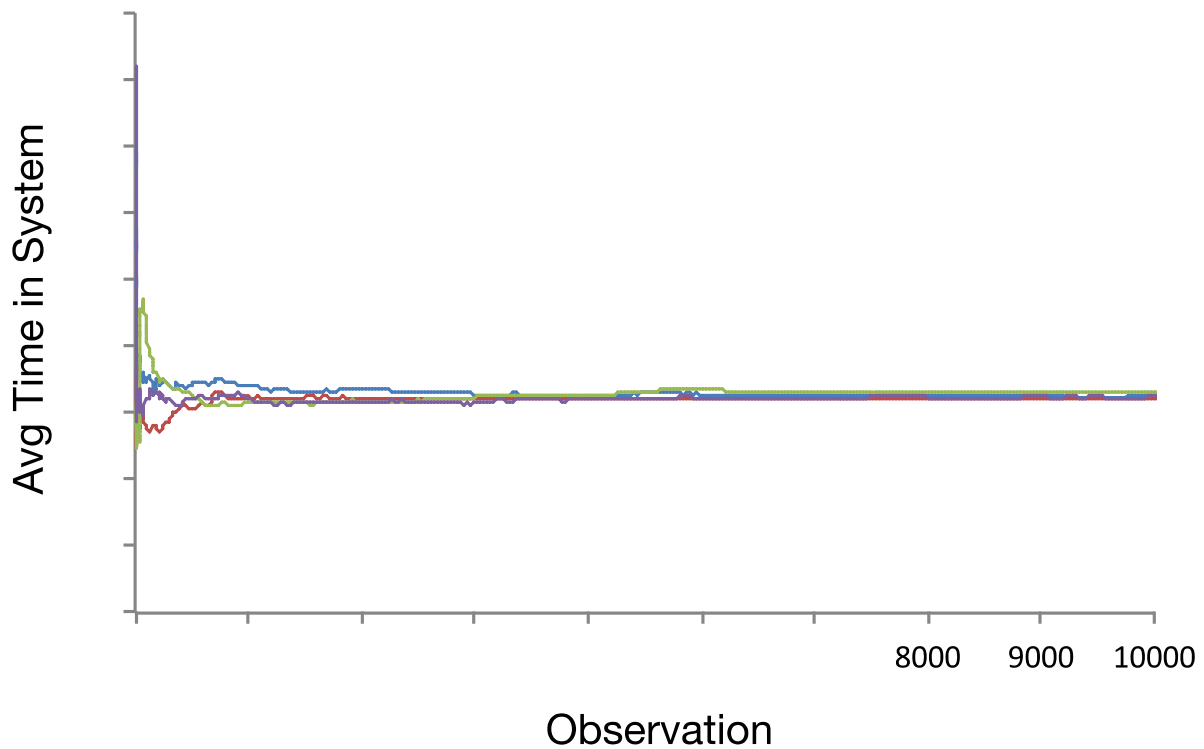


**Figure 5. Successive Observations After Truncation Point**

11-5

Effectively, the warm-up period with truncation amounts to selecting an initial state that is from the steady-state distribution, assuming that the truncation point is sufficiently large so that the simulation is indeed in steady state. In order to produce a confidence interval, the most straightforward approach is to consider each replication as one observation, and repeat the entire process (warm-up, truncation, follow-on observations) as many times as desired. This indeed results in iid observations and, assuming that indeed steady state has been reached, is unbiased for the steady-state mean of interest. This approach is known as "replication/deletion" (Law and Kelton, 2000).

An example of how the confidence intervals behave using the replication/deletion approach is shown in Figure 86.
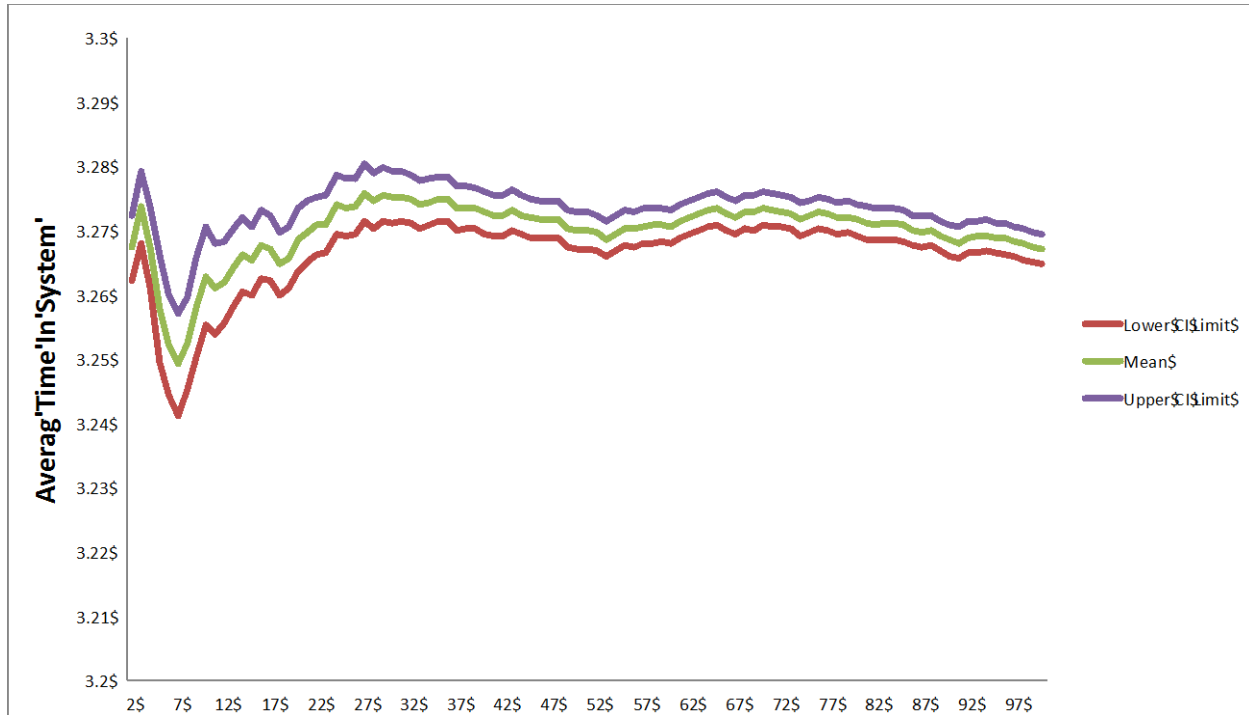


**Figure 6. Replication/Deletion Confidence Intervals**

The scale in Figure 86 has been shrunk to emphasize the small size of the confidence interval. For this particular model, it turns out that the parameters chosen for the experiments have resulted in an accurate estimate of the steady-state mean. In this case, the final confidence interval using 100 replications is 3.269±0.002. For other models, these values might not be sufficient to get an accurate estimate.

## 11.4. Specifying Precision

In general, a problem with fixed-sample methods for constructing confidence intervals is that the analyst has no control over the size of the half width. That is, the accuracy of the estimate is whatever it is, and it may not be sufficient.

## 11.5. References

[1] Banks, J., J.S. Carson, and B.L. Nelson. 1996. *Discrete-Event System Simulation, Second Edition*. Prentice Hall, Upper Saddle River, NJ.

[2] Law, A, and Kelton, David. *Simulation Modeling and Analysis, Third Edition*. McGraw Hill, Boston, MA (2000).

[3] Sanchez, S. (2001). The ABCs of Output Analysis. *Proceedings of the 2001 Winter Simulation Conference*, B. A. Peters, J. S. Smith, D. J. Medeiros, and M. W. Rohrer, eds.