Proceedings of the 1995 Winter Simulation Conference
ed. C. Alexopoulos, K. Kang, W. R. Lilegdon, and D. Goldsman

A TUTORIAL ON DISCRETE-EVENT MODELING WITH SIMULATION GRAPHS

Arnold H. Buss

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

ABSTRACT

This tutorial is an introduction to Simulation Graphs for simulation modeling. The Simulation Graph methodology is a paradigm that directly models the future event list underlying the Discrete Event approach to simulation modeling. Simulation Graphs have a minimalist design (a single type of node, two types of edges with up to three options), making them the ideal tool for rapid construction and representation of simulation models.

1 INTRODUCTION

Simulation Graphs (originally called Event Graphs) were introduced by Schruben (1983) as a means of graphically representing discrete event simulation models. Originally called "Event Graphs," they were renamed in Schruben and Yücesan (1993). The Simulation Graph approach is a minimalist one, consisting of only two basic constructs with a handful of options. Nevertheless, Simulation Graph models are extremely powerful and can be used to represent *any* discrete event model. This tutorial is an informal introduction to Simulation Graphs for representing discrete event simulation models; for a rigorous presentation of Simulation Graphs, see Schruben and Yücesan (1993).

2 BASIC ELEMENTS OF DISCRETE EVENT MODELS

We assume the reader is familiar with the basic concepts of discrete event simulation (see any introductory text such as Law and Kelton 1991), so we will only briefly review the components.

A discrete event simulation model consists of two fundamental elements: a set of *state variables*, or states, and a set of *events*. The model simulates the system being studied by producing state trajectories, that is, time plots of the values of the system's state variables. Measures of performance are determined as statistics of these state trajectories. Discrete event models have state trajectories that are piecewise constant. Events are the points in time at which at least one state variable changes value.

A discrete event model should have enough state variables to completely describe the important aspects of the system at any point in time. For example, in a model of a single server queue one possible state variable is the number of customers in the queue, Q. However, Q is not sufficient to completely describe the system at all points in time. If $Q = 0$ then there could be either 0 or 1 customers in the system. Therefore, we would need to add (at least) one additional state variable, such as the number of busy servers B.

As mentioned above, events are defined whenever at least one state variable changes value. For example, the arrival of a customer to a queueing system could be an event in a model of the system, since the number in the queue increases by 1 whenever a customer arrives (i.e. $Q \leftarrow Q+1$, or $Q{++}$ in C terminology). Similarly, when a customer finishes service and leaves the system, a server will become (possibly temporarily) idle. Thus the departure of a customer can also be an event since the value of B is decreased by 1. It is important to note that an event is an *instantaneous* occurrence in the discrete event model. No simulated time passes when an event occurs; simulated time passes only *between* the occurrence of events.

For example, suppose we wanted to generate a Poisson process with a given rate λ, which could be used to model the arrival of customers to a facility. To construct a discrete event model we define the state variable to be the total number of customers generated so far (N). The arrival of a customer causes this state variable to be incremented by 1 ($N{++}$), so we would define "the arrival of a customer" to be an event, which we will denote Arrival. The occurrence of this event corresponds with the state transition $N{++}$. In general, the state transition for an event

can be any mapping from the state space into itself.

The timing of the occurrence of events is controlled by the *Future Event List* (or simply the *Event List*), which is nothing more than a "to-do" list of scheduled events. Whenever an event is scheduled to occur, an *event notice* is created and stored on the future events list. Every event notice contains two pieces of information: (1) What event is being scheduled; and (2) The (simulated) time at which the event is to occur. The future event list keeps the event notices in order by ranking them based on the lowest scheduled time.

The future events list is managed by a "Timemaster" who controls the flow of time in the simulated world of the model. The Timemaster examines the event list to see if there are any scheduled events. An empty list means there is nothing to do, so the Timemaster terminates (i.e. the simulation run ends). If the event list is not empty, the Timemaster moves the simulated clock to the time of the first event notice and executes the event — that is, the state transitions associated with that event are invoked. Figure 1 shows the Timemaster's logic.

Figure 1 illustrates the convention that all state changes are made before events are scheduled. This is arbitrary, since we could just as easily reverse the order; the models constructed would be slightly different however. Similarly, the event notice could be removed from the event list first rather than last. The events scheduled by the Timemaster are specified by the occurring event itself and may be conditional on certain values of the current state. We will discuss scheduling events further in the following section.

Continuing with the Poisson process example, it is known that the times between arrivals are iid exponential random variables with mean $1/\lambda$. We can invoke the arrival of a customer by having the previous customer generate an interarrival time from an exponential distribution, then placing an Arrival event notice on the event list with scheduled time equal to the current simulated time plus the interarrival time. This example illustrates the fact that an event may schedule itself. A self-scheduling ability event should not be confused with recurrence of procedure calls in conventional programming.

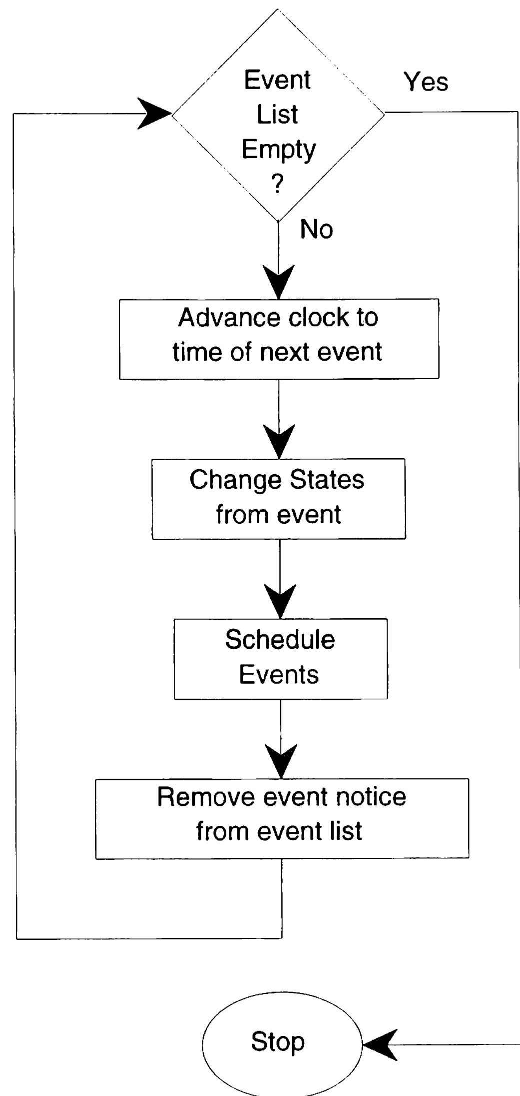

Figure 1: How the Timemaster Manipulates the Future Events List

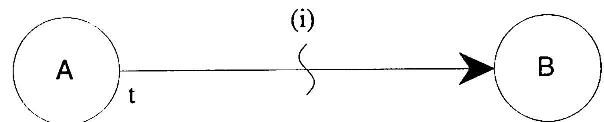

Figure 2: Fundamental Simulation Graph Construct: Whenever Event A occurs, if condition (i) is true after A's state transition, Event B is scheduled to occur t time units later

3 BASIC ELEMENTS OF SIMULATION GRAPHS

Simulation Graphs are a way of representing the Future Event List logic for a discrete-event model. Each Simulation Graph consists of *nodes* and *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 2 shows the fundamental construct for Simulation 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 (after the state transitions for Event A have been made). 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 Simulation Graph paradigm contains only two structures (the event node and scheduling edge) with two options on the edges (time delay and edge condition). The simplicity of the Simulation Graph paradigm is evident from the fact that we can represent *any* discrete event model using *only* these constructs (Schruben 1992, 1995; Schruben and Yücesan 1993). A major advantage of the minimalist approach of Simulation Graphs is that the modeler can spend more time on model formulation and less on learning the constructs of the paradigm.

There is a price to the simplicity of Simulation Graphs, however. Since Simulation Graphs represent the *Future Event List*, rather than the physical movement of, say, customers through a queueing system, Simulation Graphs require a higher degree of abstraction on the part of the user than other graphical systems. The author's experience using Simulation Graphs in an introductory simulation course indicates that this higher abstraction is easy to master and provides rich payoffs for understanding and creating discrete event simulations. Indeed, the use of Simulation Graphs tends to accelerate the understanding of the Discrete Event paradigm.

4 EXAMPLES

4.1 The Poisson Process

Our first example is probably the simplest non-trivial Simulation Graph possible, the Poisson process. The Poisson process is an important building block for larger Discrete Event models. A Simulation Graph for the Poisson process is constructed by first defining a state variable N to be the cumulative number of arrivals. Next, the event Arrival is defined to be the event which increments N by one (i.e. an arrival). The event Arrival then schedules another Arrival after a delay of t_A time units, where t_A is an Exponential random variable with mean $1/\lambda$. The Simulation Graph for a Poisson Process is shown in Figure 3. We have indicated the state transition beneath the event node Arrival as "N++" — we will use the C notation of incrementing a variable throughout this tutorial. Note that by changing the interarrival time distribution we can generate any renewal process. Moreover, by suitably augmenting the source of the interarrival times effectively any arrival process may be generated in this manner. Observe that the Simulation Graph does not explicitly model the stream of random interarrival times, but assumes that they are available.

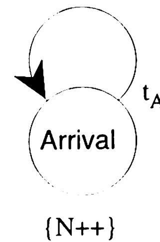

Figure 3: Simulation Graph for Poisson Process

4.2 A Simple Queueing Model

We will now construct an Simulation Graph model for the multiple-server queue. This system is the basis for more complex models and illustrates all the features of Simulation Graphs introduced so far. The system consists of $MaxB$ identical servers and a single waiting line. Arriving customers receive service on a "first come first served" (FCFS) basis; a customer in the queue receives service from the next available server when one becomes available. Note that for $MaxB > 1$ this is not the same as "first-in first-out" due to variations in service times; however, each customer *commences* service before every subsequent customer. Performance measures for this system are the expected number in the queue and expected average utilization of the servers. Other measures, such as the mean delay in the queue, could also be defined. For many queueing systems, measures such as mean delay in queue and mean sojourn time can be indirectly estimated using Little's formula. Direct estimation of these two measures would require the use of transient entities in the model. Although it is possible to construct Simulation Graph models with transient entities, we shall not do so in this tutorial. The interested reader is referred to Schruben (1995)

for a nice discussion of transient versus resident entity models.

To formulate a model for the queueing system we first identify the state variables. Defining the state variables for a model is driven by two concerns: (1) The need to represent the logic of the system being modeled; and (2) The need to compute appropriate performance measures. To estimate the two performance measures above, we must define two state variables: The number in the queue (Q) and the number of busy servers (B). It turns out that these two variables are also sufficient to represent the system at any point in time. Of course, other state descriptions are possible. For example, we could define a single state L as the number of customers in the system, but it would be more difficult to estimate our two performance measures.

Next we define the events in the model. A list of possible events could start with every situation in which a state variable changes value. A useful perspective, reminiscent of the process-orientation, is to consider what each customer encounters as they pass through the system: First a customer arrives to the system; after a possible delay in the queue, the customer starts service; finally, the customer finishes service and leaves the system. Since each of these involves a change of a state variable we identify three events as: Arrival, Start Service, and End Service, respectively. Table 1 completes the verbal description of the model. Recall our convention that events first perform their state transitions, then they schedule events.

Table 1: Verbal Description of Discrete Event Model for the Multiple Server Queue

Event	State Changes	Schedule
Arrival	Increment Queue	Arrival, after Interarrival time Start Service, if server available
Start Service	Decrement Queue Decrement # Available Servers	End Service, after Service Time
End Service	Increment # Available Servers	Start Service , if queue not empty

The final step is to translate the verbal description in Table 1 into an Simulation Graph. This is done by assigning a node to every event and an edge connecting the occurring event with the scheduled event, each with the appropriate condition and time delay.

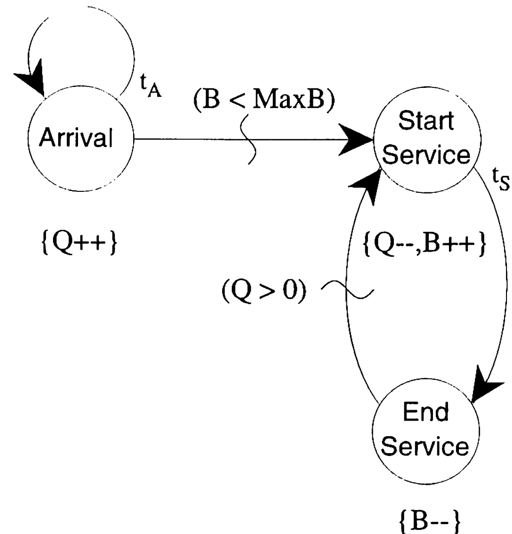

Figure 4: Discrete Event Model for Multiple Server Queue

The edge conditions are expressed as a Boolean expression in the state variables (i.e. a mapping from the state space to {TRUE, FALSE}. The Simulation Graph model in Table 1 is shown in Figure 4. The Boolean edge condition for scheduling Start Service is $B < MaxB$, since a customer arriving to find available servers can being service immediately. In Figure 4, the state transitions for each event is shown in curly braces beside the corresponding node (recall that we are using the C notation for incrementing and decrementing variables).

The Simulation Graph model in Figure 4 cannot be operationalized in its current form any more than the descriptive model of Table 1 can. To implement *any* discrete event model there must be a means for inputting system parameters, initializing the run (by placing event notices on the future events list at the start of the run), and stopping the simulation run. While these are very important (indeed, they are essential to actually running the simulation), they are primarily aspects of *implementation* and can distract the modeler from the core logic of the model. The Simulation Graph in Figure 4, while not completely specifying the starting and ending conditions, *does* convey in a powerful visual manner the interrelationships of the events. Taking the perspective of the Timemaster (i.e. the Event List logic) gives a unique perspective for constructing a Discrete Event model. In contrast to other graphical approaches, the Simulation Graph *directly* models the workings of the Event List. In the life cycle of a simulation model it is this core logic that must be correctly modeled before any

meaningful runs can be made. Other aspects, such as initial conditions and terminating conditions, are likely to vary substantially during the whole simulation process for a given model.

In the following subsection we will discuss initialization of Simulation Graph models, and we will cover one way the run can be terminated in Section 5.2 below.

4.3 Initialization

Simulation Graphs as described so far model the dynamics of the Future Event List provided that there are event notices already on the list to be executed. Initialization of a simulation run consists of three tasks: (1) All parameters (such as number of servers, mean interarrival times, etc.) must be set; (2) The initial values for each state variable must be set; and (3) The initial event notices must be placed on the Future Event List. The first two are obvious, but the third may not be so clear. Recall that the Timemaster follows the Event List logic *as long as the Event List is not empty.* If the Timemaster is invoked with an empty event list, then nothing happens.

It may not be obvious which events must be scheduled initially, especially for a complex model. The Simulation Graph paradigm allows a the modeler to easily determine *which* events must be scheduled initially: *Every event that has only incoming or self-scheduling edges must be scheduled at the beginning of the run.* Any event meeting the above criteria that is *not* initially scheduled will never occur during that simulation run. Depending on the model, other events may also need to be scheduled initially as well.

A useful convention for initialization is to specify one event that is *always* placed on the Event List at time 0. This initial event (which we will denote Run, following Schruben's (1995) terminology) performs the initializations as its state transition function and has outgoing edges to all initial events.

Figure 5 shows how this could be implemented in the queueing model of Figure 4. The event Run is added to the model and by convention is put on the Event List at time 0. The parameter ($MaxB$) for Run, the number of servers, by convention means that it is determined upon initialization of the simulation run.

5 ENHANCEMENTS

The Simulation Graph paradigm described above is a simple and elegant way to represent discrete event logic. Without any further enhancements it has sufficient flexibility and power to represent *any* discrete event model. We will discuss two enhancements of the basic Simulation Graph paradigm: passing attributes to scheduled events on scheduling edges and event canceling edges. Strictly speaking these enhancements do not increase the power of Simulation Graphs, only their readability and ease of construction.

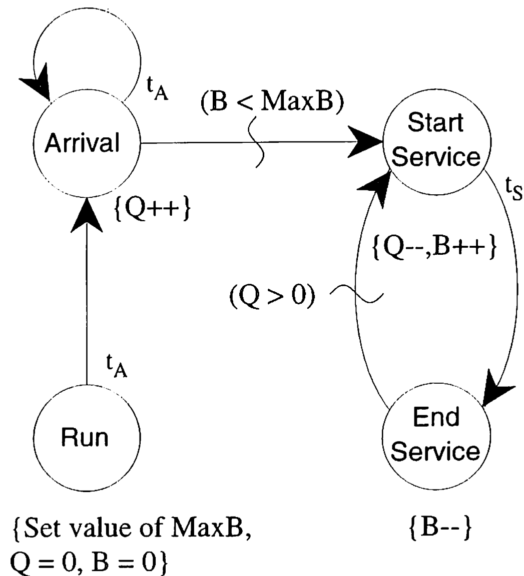

Figure 5: Discrete Event Model for Multiple Server Queue with Initial Event Run

5.1 Passing Attributes on Edges

The first enhancement provides the event node with the capability to pass attributes on an event scheduling edge to the scheduled event. Figure 6 illustrates the basic construction and is interpreted as follows: When event A occurs then, after A's state transitions have been made, if condition (i) is true event B is scheduled to occur after a delay of t time units with parameter j set equal to k. The passed parameter k could be a parameter list, as with a procedure call with arguments.

This simple enhancement allows complex models to be built up from simpler components in a relatively straightforward manner. To illustrate we will extend the queueing model of the previous section to a series of queues. This could be used to model a production facility or transfer line consisting of N machine groups, each group having a single waiting line. Jobs enter at machine group 1 and upon leaving go to machine group 2, etc. (see Figure 7). For simplicity, we will assume the queues (buffers) all have infinite

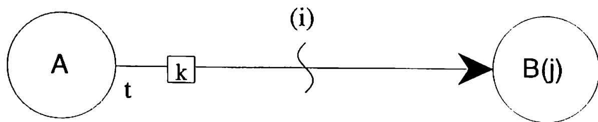

Figure 6: Passing Attributes on Edges: Whenever Event A occurs, if condition (i) is true after A's state transition, Event B is scheduled to occur t time units later with parameter j set equal to k

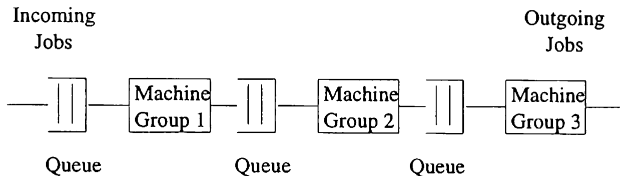

Figure 7: Transfer Line with $N = 3$ Machine Groups

capacity.

Modeling this system is made much simpler by the observation that each machine group operates like the multiple-server queueing system with two exceptions: the departure from a machine group schedules the arrival of a job to the next machine group, and the only arrival of jobs from outside the facility are to machine group 1. The state space must also be expanded to identify the number of jobs in queue as well as the number of busy machines at *each* workcenter. It is convenient to simply make Q and B arrays, with $Q(j)$ the number in queue and $B(j)$ the number of busy machines at machine group j. Similarly, the parameters of the system are now an array, with $MaxB(j)$ the number of machines in machine group j.

Figure 8 shows the Simulation Graph model for the transfer line. We have omitted the initial Run event for clarity. The similarity of this model to the queueing model in Figure 4 is self-evident. The self-scheduling edge for the Arrival(j) event adds the condition that $j = 1$ to generate the arrival of jobs from outside the shop. The other Arrival(j) events are scheduled from the previous machine group. However, an End Service(j) event with $j = N$ results in a job leaving the system. Therefore, there is the condition $j < N$. All other edges in the model are the same as the corresponding ones in Figure 4, with parameter j representing the current machine group being passed. The state transitions for the events are similarly indexed by the corresponding machine group number.

The transfer line model could have been modeled

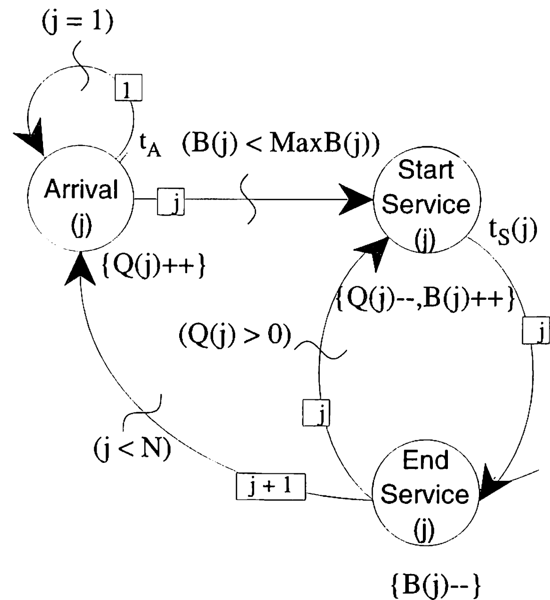

Figure 8: Simulation Graph for Transfer Line Model

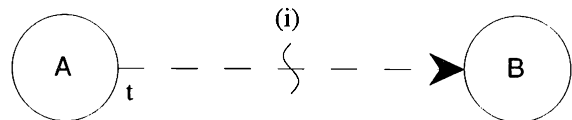

Figure 9: A Canceling Edge

using just the basic constructs in Section 3 (without passing attributes) by simply stringing together copies of the model in Figure 4 and making the appropriate adjustments in edges. However, that approach would "hard-wire" the number of machine groups N into the model. To simulate facilities having different numbers of machine groups a new model would have to be constructed. In contrast, the Simulation Graph in Figure 8 can be used to model transfer lines of *any* size by simply setting the appropriate value of N and of $MaxB(j)$ for $j = 1, \ldots, N$.

5.2 Canceling Edges

The second enhancement covers situations in which the modeler wishes to have an event notice removed from the event list. That is, a scheduled event needs to be canceled. This is accomplished in a Simulation Graph by the addition of *canceling edges* denoted by dashed arrows; Figure 9 shows the basic construction of a canceling edge. The interpretation of Figure 9 is: When event A occurs, then (after the appropriate state transitions are made), if Condition (i) is true,

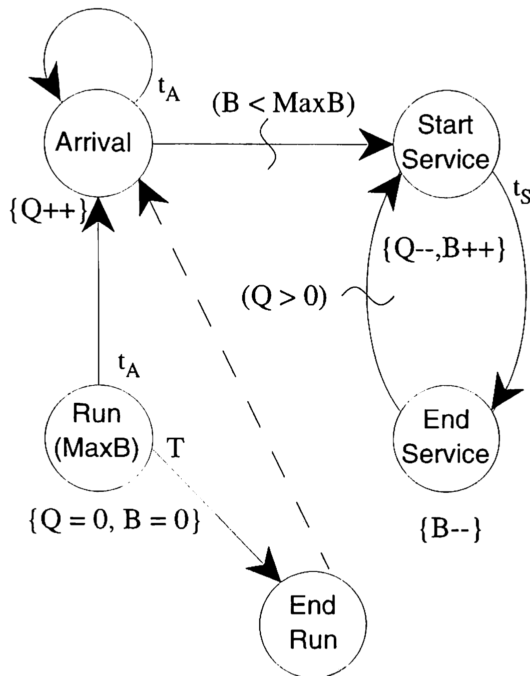

Figure 10: The Run and End Run Events Initialize and Terminate the Simulation Run

the next occurrence of event B is removed from the Event List. If no event notice for B is on the event list, then nothing happens.

We can use a canceling edge to terminate a simulation after a given amount of simulated time has elapsed. Suppose we wish to run the simulation for T minutes, then cut off the arrival of customers, but serve all the customers who are in the system at time T. We define an End event that is scheduled to occur T minutes after Run (see Figure 10). The event End causes no state changes, but simply cancels the next Arrival. This has the effect of cutting off all incoming customers at time T. Thus, at time T the only event notices possibly on the Event List are End Service events. Eventually the Event List will empty and the run will consequently end. If the modeler wants the simulation to terminate at time T without serving all customers in the system, then all Start Service and End Service event notices must also be canceled. One way to accomplish this is to add canceling edges from End Run to Start Service and End Service and add a self-scheduling edge to End Run. We leave it as an exercise for the Reader to do this.

6 IMPLEMENTATIONS

The only commercially available (to the author's knowledge) software that allows a user to create, run, and analyze Simulation Graphs is Sigma, available for DOS and Windows (Schruben 1992, 1995). Sigma allows the user to create event nodes and scheduling edges with easy clicks of the mouse. State transitions and parameters for nodes, as well as edge conditions and delays, are entered using pop up menus. There is provision for plotting state variables. A significant capability of Sigma is its ability to translate Simulation Graph models into portable C code, as well as Pascal. Compiled C models in Sigma are often faster than comparable models using conventional simulation packages. A unique feature of Sigma is that the user can also translate an Simulation Graph model into an English description of the discrete-event model. Sigma's features make it a desirable piece of software for any user of simulation.

7 CONCLUSIONS

We have provided a brief informal introduction to Simulation Graphs for discrete-event simulation models. Simulation Graphs are a simple, yet powerful. visual representation of the discrete-event logic of the Event List manipulation. They are currently the *only* graphical paradigm that directly models this process. Straightforward enhancements to the basic paradigm allow the modeler to easily leverage simple models into more complex ones. More important, the visual power of the Simulation Graph gives the modeler a unique perspective on the model and allows the key underlying relationships to be vividly represented. The availability of Simulation Graph software in the form of Sigma allow simulation modelers to take advantage of the benefits offered by the Simulation Graph paradigm.

ACKNOWLEDGEMENTS

Support for this work from the Naval Postgraduate School is gratefully acknowledged.

REFERENCES

Law, Averill and David Kelton. 1991. *Simulation Modeling and Analysis, Second Edition* , McGraw-Hill.

Schruben, Lee. 1983. Simulation Modeling with Event Graphs, *Communications of the ACM*, **26**, 957–963.

Schruben, Lee and Enver Yücesan. 1993. Modeling Paradigms for Discrete Event Simulation, *Operations Research Letters*, **13**, 265–275.

Schruben, Lee. 1995. *Sigma: A Graphical Simulation Modeling Program*, Boyd and Fraser Publishing Company, Danvers, MA.

Schruben, Lee. 1995. *Graphical Simulation Modeling and Analysis Using Sigma for Windows*, Boyd and Fraser Publishing Company, Danvers, MA.

AUTHOR BIOGRAPHY

ARNOLD BUSS is currently a Visiting Assistant Professor in the Operations Research Department at the Naval Graduate School. He is interested in many aspects of simulation, most of which are too bizarre to discuss in polite company.