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

ELIMINATING EVENT CANCELLATION IN DISCRETE EVENT SIMULATION

Eric L. Savage
Lee W. Schruben

School of Operations Research and Industrial Engineering
Cornell University
Ithaca, New York 14853, U.S.A.

ABSTRACT

The cancellation of previously scheduled events not only results in a model running less efficiently, it precludes the application of some analysis techniques such as Infinitesimal Perturbation Analysis. While some simulation languages (SIMSCRIPT, SIGMA) include an explicit facility for event cancellation, others do not (SLAM, GPSS, SIMAN). From computation theory, it is known that event cancellation is never necessary; but it is sometimes a convenient modeling technique. Unfortunately, there has been no general methodology developed for eliminating event cancellation from a simulation model. In this paper we present a simple general approach. Applications to two classical models where event cancellation is typically used serve as illustrations of the method.

1 INTRODUCTION

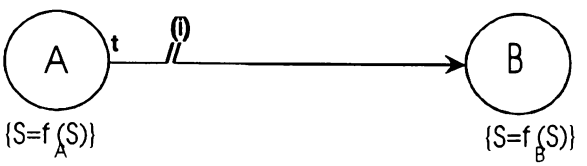
In this paper, we present two common systems that are typically modeled by cancelling events, a preemptive priority queue and a queue with a state dependent service rate. We then show how to create models of these systems without event cancellation. These two examples illustrate a general approach to simulation modeling without event cancellation that may be applied in similar situations.

In real-world models event cancellation is frequently used. Often it arises directly from the perceived logic of the system. In these cases, using event cancellation may be the simplest way to represent that logic. It is known from computation theory that event cancellation is never necessary and it should be thought of as a convenience in creating and understanding a model. Unfortunately, allowing for event cancellation invalidates some proofs of certain analysis techniques such as infinitesimal perturbation analysis to estimate performance gradients (Glasserman 1991) or structural model equivalence (Yucesan and Schruben 1992). Proofs that such methods are valid often use a Generalized Semi-Markov Process representation of a simulation where event cancellation is explicitly excluded (Glasserman) or included by allowing events to be abandoned (Iglehart and Shedler 1983) or by allowing the rates of event clocks to depend on the system state (Glynn 1989). Since event cancellation is not necessary, it can be argued that variable rate event clocks do not improve the modeling power of GSMPs. Even when event cancellation does not invalidate it, simplifying assumptions are often made to facilitate analysis (e.g., Som and Sargent 1989).

This paper will demonstrate ways in which models that allow event cancellation can be transformed into behaviorally equivalent models that do not. For illustration we use event graph models.

1.1 Event Graph Models

We will illustrate elimination of event cancellation using event graph models (EGMs) first described by Schruben (1983) and later enriched by others, including Som and Sargent. Pictorially the vertices of an EGM represent the various events in the simulation. The edges of the graph represent relationships between events. Basically, the edges define the conditions under which and the time delay after which one event will schedule another event to occur. Suppose the following edge is part of a simulation graph,

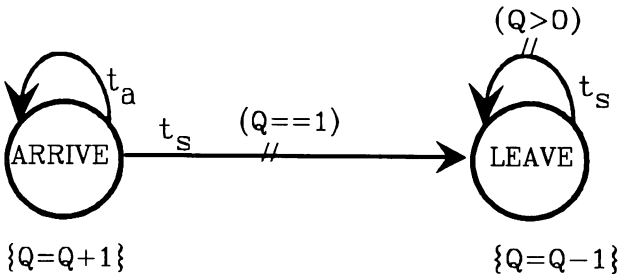


This edge is read as follows:

"Whenever event A occurs, the system state, S, changes to $f_A(S)$. Then, if condition (i) is true, event B will be scheduled to occur after a delay of t."

Appropriate labels are omitted if the inter-event edge delay is zero or if the scheduling is unconditional.

One of the simplest examples of an EGM is a single server queue. Here the single state variable, Q, is the number of customers waiting in line (including any customer that might be in service). The random time between customer arrivals is denoted as t_a and the random time of customer service is t_s. Standard C notation is used. The EGM for a generic queue is as follows:



The ARRIVE event simply increments the queue and the LEAVE event decrements the queue.

We formally define an EGM using a directed graph G = {E,V} with edge set E and vertex set V and an associated state space, S. Generic vertices are denoted by v (perhaps with a subscript). Generic edges are denoted as $e = (v_o,v_d)$, which specifies the origin and destination of a directed edge. We label the graph with the following sets:

$F = \{f_v: \forall v \in V\}$ are the state changes associated with each event.

$P = \{p_e: \forall e = (v_o,v_d) \in E\}$ are execution priority expressions used to break time ties.

$T = \{t_e: \forall e = (v_o,v_d) \in E\}$ are the inter-event delay times.

$C = \{c_e: S \rightarrow \Re, \forall e = (v_o,v_d) \in E\}$ when $c_e = 0$, the edge condition is false (as in standard C).

The conditions in C specify whether or not an edge's destination event will be scheduled after the edge's origin event occurs. At any given time in the execution of the simulation, those edges where $c(s) \neq 0$ (i.e. the edge conditions are true) are referred to as *active* edges. Edges where c(s) = 0 can be thought of as being temporarily missing from the graph.

The basic notion of an event graph model, M = (V,E,S,F,P,T,C), is to represent the *indices* for above sets with the edges and vertices of a directed graph. It is this graph of indices that organizes the above sets into a simulation model.

EGMs have a strong similarity to differential equation representations of continuous systems. The fundamental element in both is an expression of the *changes* in system state. Both require only the specification and initial and termination boundary conditions for execution. This is in contrast to Petri nets, finite state machines, or queuing rate transition diagrams where vertices express the *values* of states rather than the changes in these values. The basic thesis of research on EGMs is that directed relationship graphs are an effective means for modeling these systems and that graph theory is part of an appropriate mathematical base for the analysis of these models.

Enrichments of the event graph model include edges that cancel rather than schedule events, attributes attached to edges that can store values on the events list and parameters for each vertex that determine the assignment of these stored values. These enrichments are included in event graphs not out of necessity but rather as a convenience for modeling. That they are not necessary has been proven by the modeling of a Turing Machine (Yucesan 1989).

Cancelling edges are the particular concern in this paper. The other enrichments can all be thought of as modeling conveniences because they can be expanded in a relatively straightforward way. Attributes and parameters for example, usually represent multiple copies of the appropriate vertices and edges.

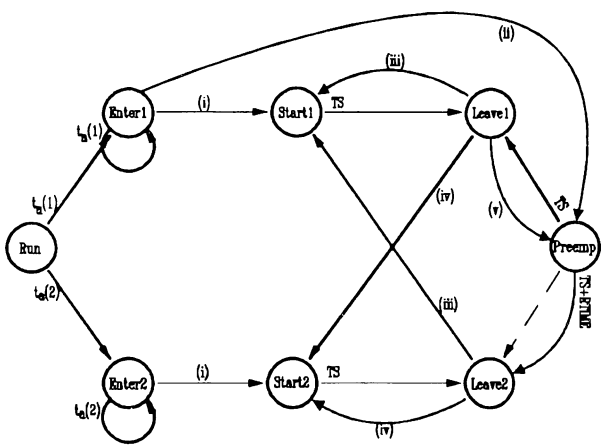
Cancelling edges on the other hand have a distinctly different function than any of the basic elements. As different as their function is, it has been proven that they along with the other enrichments are not necessary. What this proof does not provide, is a way to eliminate cancelling edges from existing models without changing the behavior of the model. If it is possible to replace cancelling edges with behaviorally equivalent structures, then a model that includes them could be transformed into one more readily analyzed. Although different in form, the transformed model would mimic the behavior of the original system.

2 EXAMPLE 1: A PREEMPTIVE SERVICE QUEUE WITH TWO TYPES OF CUSTOMERS

The system being modeled is a single server queue with two types of customers. Of the two, type 1 customers are considered to be high priority. If a type 2 customer is being served when a type 1 customer arrives, service of the type 2 customer will be preempted by the start of service of the type 1 customer. That is, the type 2 customer will have to wait for the rest of its service until

the server services any type 1 customers in the system. This is a common system usually modeled with cancelling edges and, especially for distributed simulation, different modeling techniques have been developed that avoid event cancellation (Cota and Sargent, 1989). These new approaches are important for their contributions to distributed simulation, what we show is just that it is possible to remove cancelling edges without appealing to a new representation.

There are two options for the modeling of the arrival process. In this paper we model two separate arrival processes, one for each type of customer (Figure 1). The other possibility is to model one arrival process and randomly assign a priority level to the incoming customers. The first approach allows us to model separate distributions for the inter-arrival time of each type of customer whereas the second would be more appropriate if we do not have separate distribution information for each type but only the ratio of the two types of customers. Although we use the first approach, the transformation would be the same for either approach to the model since the use of cancelling edges does not affect the arrival process in any way.



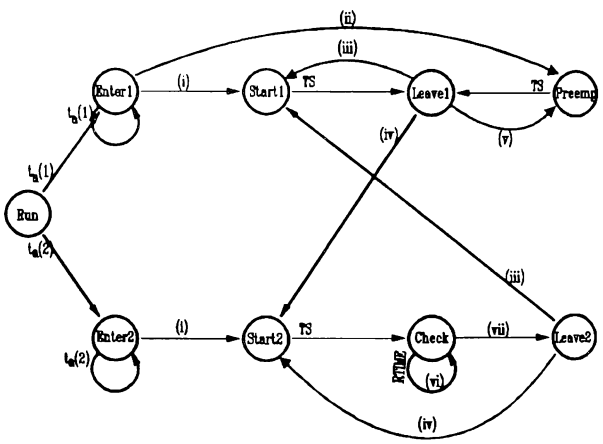
	State Changes	Conditions
Enter(I):	{Q(I)=Q(I)+1}	(i) S=1
Start(I):	{Q(I)=Q(I)-1, S=S-1, PR=2-I, TS=t_s(I), FTIME=CLK+TS}	(ii) S=0 and PR=0
		(iii) Q(1)>0 and S=1
Leave(I):	{S=S+1, PR=0}	(iv) Q(1)=0 and Q(2)>0 and S=1
Preemp:	{Q(1)=Q(1)-1, S=S-1, PR=1, TS=t_s(1), RTIME=FTIME-CLK, FTIME=FTIME+TS}	(v) Q(1)>0 and S=0

Figure 1: Preemp.MOD With a Cancelling Edge

The focus here is on the way in which the preemption is modeled. If a priority customer arrives (I = 1 at the Enter1 event), and the server is busy (S = 0) with a non-priority customer (PR = 0) then service is preempted. In addition to removing the priority customer from the queue (Q(1) = Q(1) - 1), the server status is set to -1 which indicates that preemption has occurred and PR = 1 indicates that we are now serving a priority customer. In order to properly finish the non-priority customer when the priority queue is once again empty, we need to keep track of the amount of time remaining for the type 2 service (RTIME). When the type 2 customer resumes service, the leave event would be rescheduled in that amount of time.

In this model we can reschedule the leave event immediately. If the type 1 service time is pre-computed (not calculated as the edge delay) then we can calculate the new finish time for the type 2 customer (FTIME = FTIME + TS). This new time is contingent on there being no more preemptive service events. If another type 1 customer does arrive however, the Preemp event will be executed and the finish time for the type 2 customer will again be recalculated.

When a preemptive service occurs, according to the conditions given above, the old Leave2 event is canceled, A Leave1 event is scheduled to occur after the service time TS and a new Leave2 event is scheduled to occur after TS + RTIME.



	State Changes	Conditions
Enter(I):	{Q(I)=Q(I)+1}	(i) S=1
Start(I):	{Q(I)=Q(I)-1, S=S-1, PR=2-I, TS=t_s(I), FTIME=CLK+TS}	(ii) S=0 and PR=0
		(iii) Q(1)>0 and S=1
Leave(I):	{S=S+1, PR=0}	(iv) Q(1)=0 and Q(2)>0 and S=1
Preemp:	{Q(1)=Q(1)-1, S=S-1, PR=1, TS=t_s(1), FTIME=FTIME+TS}	(v) Q(1)>0 and S=0
Check:	{RTIME=FTIME-CLK}	

Figure 2: Preemp.MOD Without Cancelling Edges

To model this system without cancelling edges (Figure 2), it is necessary to avoid putting a Leave2 event on the list when it may have to be canceled later. Instead of scheduling the Leave2 event directly, the arrival of a type 2 customer schedules a Check event. This event will then schedule the Leave2 event only if no preemption has occurred. If preemption has

occurred, the Check event must schedule another Check when the type 2 service is now rescheduled to be finished.

The Preemp event state changes no longer need to include the calculation of remaining time because this event no longer schedules the new Leave2 event. As before, it does recalculate the scheduled finish time so that this will be done for each preemptive service.

It is the Check event that calculates the remaining service time. If the remaining service time is 0, then either the type 2 customer was not preempted or the preemptive services are done and the leftover service time has elapsed; Leave2 can be scheduled. If the remaining time is positive then another check is required when this time runs out.

3 THE BASIC METHOD

The model transformation in Section 2 is a special example of a more general approach. If an event V might be canceled in the course of the simulation, it should not be scheduled until it is clear that it will not be canceled, e.g. immediately before its execution. To accomplish this we create a dummy event called CheckV. Any event that, in the original model, would schedule V schedules CheckV instead. Any event that would cancel V instead increments a variable VCancel. When CheckV executes, it simply decrements VCancel unless VCancel = 0, in which case it schedules V to occur immediately.

More formally, let v be the event that may be canceled, v_{ch} be the added check event and v_x represent an event that might cancel event v.
The following steps replace the cancelling edge:

1) Remove the cancelling edge (v_x, v).
2) Create new state variables VC (to count the cancellations of v) and SV (= 1 if v has not been canceled, 0 otherwise).
3) Add {VC = VC + 1} to the state changes at v_x.
4) Add the vertex v_{ch} with state change: {SV = 1 iff (VC = 0), VC = max(VC - 1, 0)}.
5) Add the edge (v_{ch},v) with edge condition c_e = SV
6) Replace all scheduling edges (v',v) with scheduling edges (v',v_{ch}) with the same time delay and edge conditions.

This approach assumes a cancelling edge with zero time delay and works best in cases where an event is canceled and rescheduled due to some kind of interruption. It would also work in any case where the successively scheduled copies of the event have a non-decreasing scheduled time. If a canceled event is rescheduled at an *earlier* time, the method described here cancels the earliest scheduled event when it next attempts to execute. Suppose at time t, we want to cancel an event V scheduled to happen at time t + x but event V is later scheduled (by some other event) to occur at time t + y, 0 < y < x. In the original model the event V scheduled at t + x will be canceled but in our translation the event V at t + y will be canceled instead.

This simple check event approach is similar to that proposed by Narain and has similar problems. In his DMOD formulation the check event examines the history of the simulation to see if a cancelling event has occurred since the event that scheduled it (Narain 1991). (Instead of examining the history, the event graph simulation records the occurrence of the cancelling event by means of the new state variable VCancel.) He does not explicitly provide for the case where two or more copies of the event have been scheduled and not executed. In the absence of this possibility, VCancel would be a simple binary variable. It would have to be assumed that an event will not be scheduled again until after the previous copy has tried to execute. If this is not the case and the algorithm merely checks to see if the cancelling event has been executed (VCancel = 1 in the analogous event graph). A second scheduling of event V, which will have to set VCancel equal to zero, will re-enable the canceled event. Alternately, VCancel could be reset by the attempt to execute but then the above problem is encountered, i.e. if the new scheduled time is earlier than the old one, the wrong event is canceled.

One solution to the problem is to assign a scheduling number to each occurrence of the event that represents the order in which it was scheduled. Another state variable would be added to keep track of the scheduling number of the earliest event on the events list. The cancel variable would now be an array of binary variables (one for each instance of the event being scheduled). When an event is to be canceled, the entry corresponding to the scheduling number of the earliest event is set to 1 and the earliest event scheduling number updated.

It is possible that these scheduling numbers could be stored in a separate array that would be searched or updated when the event is scheduled or canceled. To do it properly, it would be necessary to keep a separate list of the scheduling numbers ranked by time of scheduled execution. The scheduling number would be put on the list when the event is scheduled and taken off when the simulation attempts to execute the scheduled event. In all, three arrays would be created. Let SN be the scheduling number of an occurrence of event V. VSTime(SN) would be the scheduled time of execution of that occurrence of event V, VCancel(SN) would be a binary variable indicating whether that occurrence had

been canceled and VOrder() would be the list of scheduling numbers ranked by the values stored in VSTime().

The following modifications would be made to the simple check algorithm:

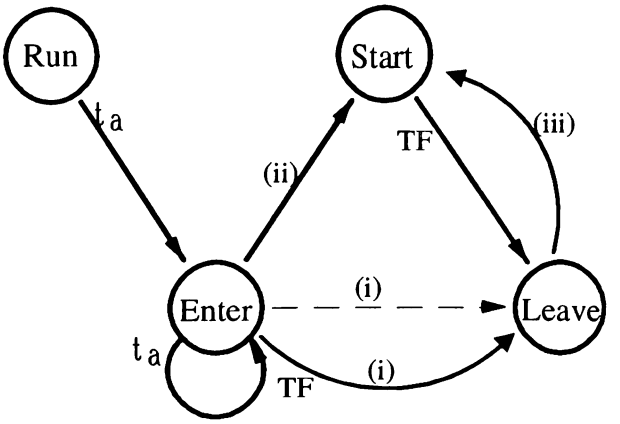
1) For each vertex v' that may schedule v, create a new event v'_v that increments the scheduling number SN, assigns the value of VSTime(SN) and updates VOrder() accordingly.
2) Add edges (v',v'_v) with the same edge condition as (v',v_{ch}) but with no time delay.
3) Any cancelling event v_x finds the first SN in VOrder() such that VCancel(SN) = 0 and sets VCancel(SN) = 1.
4) The event v_{ch} checks VCancel(VOrder(0)) instead of VC and updates VOrder() by removing the first element and updating the array.

4 EXAMPLE 2: A SINGLE SERVER QUEUE WITH STATE-DEPENDENT SERVICE RATE

Another system that is often modeled with cancelling edges is a single server queue with state-dependent service time. Specifically, the system being modeled is a single server queue where the service rate depends on the number of customers waiting in the queue. The base rate of service is the rate at which the server works if the only customer in the system is the one being served, i.e. there is no one waiting in line. The rate function is r(Q) where Q is the length of the queue. As stated above, r(0) = 1.

The usual strategy in modeling this situation is similar to that in the previous example. When a customer starts service, the end of service is scheduled according to the present service rate. When a new customer arrives, increasing the number in the queue, the now incorrect end of service is canceled and a new one is scheduled (Figure 3).

Specifically, when a customer starts service, a number of required service units SU is generated according to a distribution that represents the service time required at the base rate. The present rate r(Q) is calculated, the Leave event is scheduled to occur SU / r(Q) time units later and the scheduled finish time is recorded. If an arrival occurs while the server is busy, the queue length is increased, the remaining service units are calculated (SU = (FTIME - CLK) * r(Q - 1)), the Leave event is canceled and a new Leave event is scheduled to occur after SU / r(Q) time units. In the model, these two calculations are combined (TF = (FTIME - CLK) * r(Q - 1) / r(Q)).

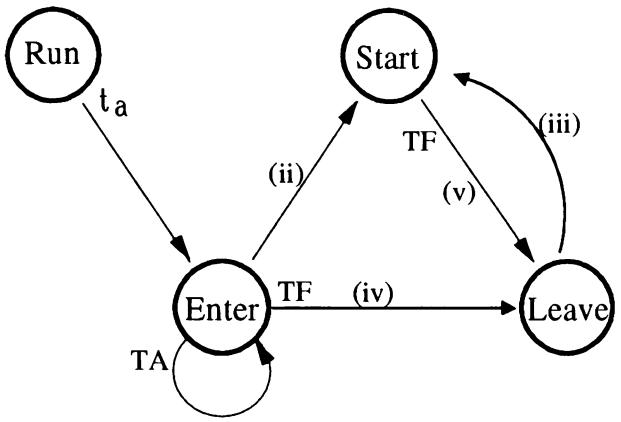


	State Changes	Conditions
Run:	{S=1}	(i) S=0
Enter:	{Q=Q+1, TF=(FTIME-CLK)*r(Q-1)/r(Q), FTIME=CLK+TF}	(ii) S=1
Start:	{S=S-1, Q=Q-1, SU=t_s, TF=SU/r(Q), FTIME=CLK+TF}	(iii) Q>0
Leave:	{S=S+1}	

Figure 3: Statedep.MOD With a Cancelling Edge

The simple check event approach works here if the rescheduled event occurrence is guaranteed to happen after the old one was scheduled. If the rate of service increases with Q, the new occurrence will always be earlier than the old one and we cannot simply add a Check event as we did in the previous example. In this case, the new Leave event is scheduled at an earlier time. The Check event would fire at the time of the original Leave; but by then the new Leave should have happened.

It would be possible to use the array approach described in Section 3, but this seems too cumbersome for this model. Fortunately, we can exploit certain features of this system to eliminate the need for cancelling edges without resorting to this solution (Figure 4). Again the fundamental approach is the same: prevent the Leave event from being scheduled until we know it will not be subsequently canceled. It would be canceled by the arrival of another customer, so we need to figure out whether service can be completed before the next arrival. This means comparing the time left to finish at the present rate TF with the time to the next arrival TNA - CLK (or TA at the Enter event). If TF is smaller, then the Leave event will not be canceled by an arrival so it can be scheduled to occur after TF time units. If TF is larger, the next arrival will occur before service is finished so the Leave is not scheduled. The scheduled finish time is updated, however, so that the remaining service requirement can be properly calculated at the next occurrence of the Enter event.



State Changes		Conditions
Run:	{S=1}	(ii) S=1
Enter:	{Q=Q+1, TA=t_a, TNA=CLK+TA, TF=(FTIME-CLK)*r(Q-1)/r(Q), FTIME=CLK+TF}	(iii) Q>0 (iv) S=0 and TF<=TA (v) TF<=TNA-CLK
Start:	{S=S-1, Q=Q-1, SU=t_s, TF=SU/r(Q), FTIME=CLK+TF}	
Leave:	{S=S+1}	

Figure 4: Statedep.MOD Without Cancelling Edges

As in the previous example, we see here a specific example of a more general approach. The strategy here is to determine which events may cancel an event V and to find a way to determine whether one of these events will happen before the time at which V would be scheduled. If this can be done, in this case by comparing the remaining service and inter-arrival times, then V is not scheduled until it will not be preceded by any of the possibly cancelling events. These cancelling events are given the ability to schedule V if the criterion is met. If another such event will happen, then any information needed to keep track of the scheduled occurrence of V, the remaining service time in this example, is updated but the event V is not scheduled.

Unlike the simple check event approach in Section 3, it is not clear that a perfectly general transformation of the type used in this section exists. It may not always be possible to determine whether an event will occur before event V, especially if there are two or more random time intervals on the scheduling path between events that schedule or cancel event V. In this example there is only one random time interval, indeed only one edge, so we can make the determination by pre-computing the random variates.

5 CONCLUDING REMARKS

Where previous work has shown, in theory, that is possible to eliminate cancelling edges from event graph models, in this paper we have demonstrated this fact in a practical way by presenting a method by which cancelling edges can be eliminated from an existing model. The intent is to provide a way to simplify models for analysis.

Unfortunately, the most general algorithm involves the use of arrays and a ranked list which may complicate analysis in different ways. In many models, of which we have shown two examples, it is possible to apply simpler translations. These translations are adaptations of the general algorithm that make use of properties and assumptions that may be shared by a wide variety of models.

One concern in the simpler translations is what effect the pre-generation of edge delay times will have on analysis. The time delays on event graph edges are usually intended to be sampled from a probability distribution. In order to remove the cancelling edges, the distribution is sampled as part of the state change so that the value can be stored as a state variable. It is unclear what effect, if any, this will have on the analysis. It is noteworthy that the general algorithm for translation does not incur this particular complication.

ACKNOWLEDGMENT

The authors are grateful to the Division of Design, Manufacture and Industrial Innovation of the National Science Foundation for their generous support through a research grant to Cornell University.

REFERENCES

Cota, B. A. and R, G. Sargent. 1989. Automatic Lookahead Computation for Conservative Distributed Simulation, CASE Center Technical Report No. 8916, Syracuse University, Syracuse, NY.

Glasserman, P. 1991. Structural Conditions for Perturbation Analysis Derivative Estimation: Finite Time Performance Indices. *Operations Research* 39:724-738

Glynn, P. 1989. A GSMP Formalism for Discrete Event Systems. *Proceedings of the IEEE* 77(1):14-23.

Iglehart, D. L. and G. S. Shedler. 1983. Simulation of Non-Markovian Systems. *IBM Journal of Research and Development* 27(5):472-479.

Narain, S. 1991. An Axiomatic Basis For General Discrete Event Modeling. *Proceedings of the 1991 Winter Simulation Conference* (B. L. Nelson, W. D. Kelton and G. M. Clark, eds.), Phoenix, AZ, 1073-1082.

Schruben, L. W. 1983. Simulation Modeling with Event Graphs. *Communications of the ACM* 26(11):957-963.

Som, T. K. and R. G. Sargent. 1989. A Formal Development of Event Graphs as an Aid to Structured and Efficient Simulation Programs. *ORSA Journal on Computing* 1(2):107-125.

Yucesan, E. 1989. Simulation Graphs for Design and Analysis of Discrete Event Simulation Models, Ph.D. Dissertation, School of OR&IE, Cornell University, Ithaca, NY.

Yucesan, E. and L. W. Schruben. 1992. Structural and Behavioral Equivalence of Simulation Models. *ACM Transactions on Modeling and Computer Simulation* 2(1):82-103.

AUTHOR BIOGRAPHIES

ERIC L. SAVAGE is a Ph.D. candidate in the School of Operations Research at Cornell University. He received his undergraduate degree in mathematics from Rensselaer Polytechnic Institute and an M.S. degree in Operations Research from Cornell University. His research interests include modeling logic and methodology in graphical representations.

LEE W. SCHRUBEN is a Professor in the School of Operations Research at Cornell University. He received his undergraduate degree in engineering from Cornell University and his Ph.D. from Yale University. His research interests are in statistical design and analysis of simulation experiments and in graphical simulation modeling methods.