Proceedings of the 1990 Winter Simulation Conference
Osman Balci, Randall P. Sadowski, Richard E. Nance (eds.)

SIMULTANEOUS EVENTS AND DISTRIBUTED SIMULATION

Bruce A. Cota
Robert G. Sargent

Simulation Research Group
Syracuse University
Syracuse, New York 13244

ABSTRACT

In this paper we examine the handling of simultaneous events in distributed simulation. We study the ordering relation that a distributed simulation implicitly enforces on simultaneous events. We then give an algorithm that can be used to assign priorities to processes so that the ordering relation enforced by a distributed simulation can also be enforced in a sequential simulation. The motivation for this is that if distributed implementation and a sequential implementation of the same simulation were being developed, it would be necessary for the two implementations to handle simultaneous events in the same manner.

1. INTRODUCTION

It is well known ([Zeigler 1976; Som and Sargent 1989]) that some tie-breaking rule is required in a sequential discrete event simulation to ensure that the results of the simulation (that is, the data collected from the simulation) are well defined if simultaneous events occur. These tie-breaking rules are usually defined by having the modeler assign priorities to specific components of the model ([Zeigler 1976]) or to different kinds of events ([Som and Sargent 1989]).

In a distributed simulation ([Jefferson 1985; Misra 1986]), events may be simulated in parallel or in a different order than in a sequential simulation, but the results of a distributed simulation are the same as the results of a sequential simulation. In [Misra 1986], there is a brief discussion of the fact that simultaneous events must be simulated in correct order. An example is given in which the simulation of one event influences the simulation of a second, simultaneous event, and it is observed that the simulation will not be correct unless the first event is simulated before the second event. It is then stated that a distributed simulation will simulate the events in correct order because "distributed simulation is based on the dependency order". It is also observed that, in a sequential simulation, it is necessary to store "additional facts" with entries in the event list in order to ensure that simultaneous events are simulated according to the dependency order. We are therefore lead to ask what additional facts need to be stored with each entry in order to ensure that a sequential simulation will have the same results as a distributed simulation based on the same model.

There is an important reason for answering this question. If we were to develop a distributed implementation of a simulation language and a sequential implementation of the same language we would need to ensure that simultaneous events were handled exactly the same way in both implementations. We would also need to answer this question if we were using a sequential simulation to validate or to evaluate a distributed simulation algorithm.

In this paper we study the handling of simultaneous events in distributed simulation, and we show that, under the assumptions on model structure that are usually required for distributed simulation, the dependency order can be enforced in a sequential simulation by using a particular assignment of priorities to processes in the simulation model. We begin in section 2 by introducing some basic concepts and terminology pertaining to sequential simulation and simultaneous events. In section 3 we discuss the handling of simultaneous events in distributed simulations and show that it is equivalent to a particular assignment of priorities to processes. In section 4 we give an algorithm that can be used to make this assignment of priorities to processes before a simulation begins. Of course, this cannot be done if new processes are created during the simulation. Therefore, in section 5 we describe how this assignment of priorities might be made dynamically in a simulation in which processes can be created and destroyed. Finally, in section 6, we summarize.

2. BASIC CONCEPTS AND TERMINOLOGY

A discrete event simulation can be viewed as the simulation of a sequence of *events*, which are instantaneous changes in the state of the system being simulated. Each event occurs at some point in time and, in a sequential simulation, events are simulated in order according to the times at which they occur, so that if one event occurs at an earlier time than a second event, the first event is simulated before the second event. In a distributed or parallel simulation, events may not be simulated in this sequence, but the *results* of the simulation — that is, any data collected from the simulation — are as if the events had been carried out in this sequence.

A sequential discrete event simulation typically maintains a list of events called the *event list*, and events in the event list are called *pending events*. The simulation of an event may change the event list by adding, or *scheduling*, pending events, and by removing, or "canceling", pending events. Each pending event is said to be scheduled to occur at some particular point in time. The time at which a pending event is placed in the event list is said to be the time at which the event is scheduled. The time at which a pending event is scheduled to occur must be greater than or equal to the time at which the event was scheduled. The pending event that is scheduled to occur at the earliest point in time is called the *next event*. The simulation is carried out by repeatedly removing the next event from the event list, advancing the simulation clock to the time at which the next event is scheduled to occur, and simulating the next event.

Two or more pending events may be scheduled to occur at the same point in time. Such events are said to be *simultaneous*. Some tie-breaking rule is needed to decide which of any set of simultaneous events is simulated first. The most common way to do this is to use some rule that assigns a priority to each event. When there are two or more simultaneous events, the event with the highest priority is defined to be the next event.

In general, a model needs a tie-breaking rule in order to be well defined. However, as discussed by [Misra 1986] and by [Som and Sargent 1989], the order in which a set of simultaneous events occur does not always effect the results of the simulation. When the order in which two simultaneous events is simulated does not effect the results of the simulation, the pair of events is said to be *independent*. If a pair of simultaneous events is not independent, then the pair is said to be *dependent*.

Note that a tie-breaking rule gives an ordering on each possible set of simultaneous events in the simulation. We refer to this ordering as the *tie-breaking ordering*. For example, if priorities are assigned to events, then one event precedes a second in the tie-breaking ordering if and only if the first event has a higher priority than the second event. However, if a pair of events in a set of simultaneous events are independent, then those two events need not be *related* under the tie-breaking ordering. That is, neither event need precede the other under the ordering. Thus, for example, independent events may have identical priorities. In general, for a model to be well defined, we need a tie-breaking rule that defines an ordering on every possible set of simultaneous events that relates every pair of dependent events in that set.

3. DISTRIBUTED SIMULATION

In a distributed simulation, different components of a system are simulated in parallel. Each component is typically referred to as a *physical process* and the simulation of each component is referred to as a *logical process*. We subsequently use the term *process* to refer to a component of the model that corresponds to a physical process. Each event in the simulation is assumed to be a state change in exactly one physical process and is simulated by exactly one logical process, but an event may cause messages to be sent to other processes as described below.

In a distributed simulation, events within each logical process are simulated in order according to the times at which they occur. Simultaneous events from the same process can therefore be handled in any of the ways that simultaneous events are handled in a sequential simulation (by assigning priorities to events). However, in a distributed simulation, events from different processes may be simulated out of order or in parallel. As mentioned by [Misra 1986], a distributed simulation implicitly uses the "dependency order" as a tie-breaking rule to determine which of a set of simultaneous events is simulated first. In this section we show that this dependency order can be enforced in a sequential simulation by using a particular assignment of priorities to processes in the model.

In this paper, we discuss tie-breaking rules that are based on the assignment of priorities to processes. An assignment of priorities to processes can be used for any set of simultaneous events from different processes. We assume that some "local" tie-breaking rule has been given for each process that can be used to determine which of a set of simultaneous events from that process. This might take the form of an assignment of "relative" priorities to events within that process. Note that it is straightforward to combine an assignment of priorities to processes with "local" tie-breaking rules. We can do this by using the hierarchical approach to tie-breaking rules of [Zeigler 1984]. Given a set of simultaneous events from different processes, we first select the process with highest priority from among those processes whose events are in the set. We then use the process's "local" tie-breaking rule to select an event from that process as the next event.

Throughout the rest of this paper, whenever we refer to simultaneous events we are referring to simultaneous events from different processes. Similarly, when we refer to tie-breaking rules, we are referring to tie-breaking rules to be used on sets of simultaneous events that are all from different processes.

Distributed simulation requires algorithms that synchronize logical processes in a way that ensures correctness. Most work in distributed simulation is based on the algorithms discussed by [Jefferson 1985] and/or those discussed by [Misra 1986]. These algorithms are based on a particular paradigm of simulation (discussed in detail in [Misra 1986]), in which logical processes communicate only by sending timestamped messages. In this paper, we discuss the handling of simultaneous events in distributed simulations based on this paradigm. The handling of simultaneous events in different kinds of distributed and parallel simulations are discussed by [Cota and Sargent 1989] and [Cota and Sargent 1990].

In the paradigm of simulation discussed by [Misra 1986], processes communicate through message passing. Every message is sent by an event and the time at which that event occurs is said to be the time at which the message is sent. When a message is sent, the message is assigned a *timestamp* that is greater than or equal to the time at which the message is sent. We assume that the sending of a message with timestamp 't' to process 'P' causes an event from P to be simulated at time t (which may do nothing more than process that message).

As described by [Misra 1986], a distributed simulation will give the same results as a particular sequential simulation. The sending of a message in the distributed simulation corresponds to the scheduling of an event in a sequential simulation. The time at which the message is sent is the time at which the event is scheduled, and the timestamp of the message corresponds to the time at which the event is scheduled to occur. The correctness of the distributed simulation is ensured by simulating events according to the *dependency order*, which means that whenever the simulation of one event can influence the simulation of a second event, the first event is simulated before the second event. (In the Time Warp algorithm of [Jefferson 1985], events may be simulated out of dependency order, but whenever this happens at least one of the events is "rolled back" and simulated again according to the dependency order, so that the results of the simulation are as if events were simulated in dependency order.)

In this section, we say that an event 'e' *causes* a second event, 'e'', if e sends a message that causes e', or if there is some sequence of events, '$\langle e_1, \ldots e_n \rangle$', such that e causes e_1, for each $i: 1 \leq i \leq n-1$ e_i causes e_{i+1}, and e_n causes e'. Suppose that an event 'e_1' from process 'P_1' and an event 'e_2' from process 'P_2' both occur at time t. Then these two events are independent unless e_i (for $i = 1$ or 2) either sends a message with timestamp t to P_j (for $j = 1$ or 2, $j \neq i$) or causes an event that sends a message with timestamp t to P_j.

In order to ensure that the simulation progresses, distributed simulation algorithms generally require certain conditions on the structure of the model to be satisfied. For example, [Jefferson 1985] assumes that whenever a process sends a message, the timestamp of that message is greater than the time at which the message is sent. This ensures that any two simultaneous events from different processes are independent. In this case, no tie-breaking rule is needed to ensure that the model is well defined.

[Misra 1986] assumes that a model meets a weaker condition. To explain this condition, we define a *cycle* of processes to be a sequence of processes, '$\langle P_1, \ldots P_n \rangle$', such that for each $i: 1 \leq i < n$, P_i can send messages to P_{i+1} and P_n can send messages to P_1. P_1 is said to be P_n's *successor* in the cycle, and for each $i: 1 \leq i < n$, P_{i+1} is said to be P_i's successor in the cycle. [Misra 1986] assumes that in every cycle of processes, there is at least one process, 'P_k', such that whenever P_k sends a message to its successor in that cycle, the timestamp of that message is greater than the time at which the message is sent. When a model meets this condition, [Misra 1986] refers to the model as having the *predictability property*. We assume in the remainder of this paper that every model has the predictability property.

We say that process 'P_1' can *immediately influence* process 'P_2' if P_1 can send a message to P_2 with timestamp equal to the time at which that message was sent. Thus, a model has the predictability property, if and only if there is at least one process in every cycle of processes that cannot immediately influence its successor in that cycle. Suppose that we construct a directed graph in which each vertex represents a process from a given model specification and in which there is an edge from process P_1 to process P_2 if and only if P_1 can immediately influence P_2. If a model has the predictability property, there are no cycles in this graph. In that case, this graph defines a partial, asymmetric ordering on processes. (That is, for any two distinct processes, either one process precedes the other under the ordering, or the two processes are unrelated under the ordering, but there is no pair of processes such that each process precedes the other.) We refer to this ordering on processes as the *influence ordering*. Note that if two processes are unrelated by the influence ordering, then any pair of events from those two processes is independent.

The reason for defining the influence ordering is that in order for the simulation of an event 'e_1' from process 'P_1' to have any effect on the simulation of an event 'e_2' from process 'P_2', e_1 must either have a smaller time of occurrence than e_2 or P_1 must precede P_2 under the influence ordering. This follows from the fact that in order for the simulation of e_1 to have any effect on the simulation of e_2, e_1 must cause a message with timestamp less than or equal to 't' to be sent to P_2, where t is the time at which e_2 is scheduled to occur. If e_1 occurs at time t, then the message must also have timestamp t, and so P_1 must precede P_2 under the influence ordering.

Because of this, we can enforce the dependency ordering on simultaneous events in a sequential simulation by enforcing the influence ordering on processes. That is, we can enforce the dependency ordering on simultaneous events in a sequential simulation by simulating simultaneous events from different processes in order according to the influence ordering on the processes to which those events belong.

Note that the use of the influence ordering as a tie-breaking ordering actually gives a stronger ordering on simultaneous events than that given by the dependency ordering. That is, event e_1 from process P_1 and event e_2 from process P_2 may be independent even if P_1 precedes P_2 under the influence ordering. In that case, the use of priorities based on the influence ordering forces e_1 to be simulated before e_2, even though they are actually independent. Since the two events are independent, however, they may be simulated in either order and so the correctness of the simulation is not effected.

4. STATICALLY ASSIGNING PRIORITIES TO PROCESSES

In this section, we give an algorithm that can be used to assign priorities to processes in such a way that whenever one process precedes a second process under the influence ordering, the first process is assigned a higher priority than the second process. However, this algorithm must be used to assign priorities to processes before the simulation begins, and so the algorithm cannot be used if new processes may be created during the simulation. The situation where processes are created dynamically during the simulation is discussed in section 5. We also note that in order to use this algorithm, we need to determine, at the start of the simulation, which processes can immediately influence which other processes. Finally, we make the (reasonable) assumption that there are only a finite number of processes in the model.

We define the *influence diagram* of a model to be a graphical representation of the influence ordering, in which processes are vertices and in which there is a directed edge from one process to a second process if and only if the first process can immediately influence the second process. Note that the influence diagram is an acyclic directed graph and that one process precedes a second process under the influence ordering if and only if there is a directed path from the first process to the second process in the influence diagram. In this case, we say that the first process is a *predecessor* of the second process. If one process can immediately influence a second process, then we refer to the first process as an *immediate predecessor* of the second process.

We define the *depth* of any vertex 'v' in an acyclic directed graph to be one plus the length of the longest path to v from any vertex with no immediate predecessors in the graph (that is, a vertex with no entering edges). Thus, for example, a vertex with no immediate predecessors has depth one. Note that a vertex has depth d if and only if all of its immediate predecessors have depth less than or equal to $d-1$ and at least one of its immediate predecessors has depth $d-1$.

If we consider a priority of one to be the highest possible priority, and a priority of two to be the second highest priority, etc., then we can assign a priority of one to all processes with depth one in the influence diagram, since those processes have no predecessors under the influence ordering. Since all immediate predecessors of processes with depth two have priority one, we can then assign a priority of two to all processes with depth two. Similarly, we can assign a priority of d to all processes with depth d in the influence diagram.

An algorithm that sets the priority of each process equal to its depth in the influence diagram is given in figure 1. We refer to this algorithm as the *depth finding algorithm*. The depth finding algorithm requires a list, 'L_0', of all processes in the model specification, and uses an integer counter associated with each process. The first loop in the algorithm initializes the integer counter of each process to equal the number of processes that can immediately influence that process. A new list, 'L_1' is then constructed that contains all processes that cannot be immediately influenced by other processes.

The main body of the depth finding algorithm is contained in the second loop. At the beginning of each iteration of the loop, the list L_i contains all processes that have not yet been assigned priorities and all of whose predecessors have depth less than i in the influence diagram. This is true at the beginning of the first iteration of the loop because no process in L_1 has any predecessor. This remains true because processes are placed in

```
Let L_0 be a list of all processes in the model
Initialize every process's counter to 0

For each P ∈ L_0 do
  For each P' that P can immediately influence,
    Add one to P''s counter

Let L_1 be the list of all processes whose counters equal 0
Let i = 1

Repeat until L_i = ∅
  Let L_{i+1} = ∅
  For each P ∈ L_i do
    For each P' that can be influenced by P do
      Subtract one from P''s counter
      If P''s counter is zero
        Then add P' to L_{i+1}
      Set P's priority to i and remove P from L_i
  Increment i

If any process has not been assigned a priority
 Then the influence ordering is not acyclic.
```

Figure 1. The Depth Finding Algorithm

L_{k+1} only during the k^{th} iteration of the loop, and processes are placed in L_{k+1} only when all of their predecessors have depth less than or equal to k in the influence diagram.

Because of this, on each iteration of the loop every process in L_i can be assigned priority i. The body of the loop therefore assigns a priority to each process, 'P', in L_i and removes P from L_i. However, before removing P from L_i, the counter associated with each process that can be immediately influenced by P is decremented.

When the counter of any process reaches zero, then all of that process's immediate predecessors have been assigned a priority, and so have depth less than or equal to i. The process is therefore placed in L_{i+1}. At the end of each iteration of the loop, i is incremented.

The loop is repeated until L_i is found to be empty at the beginning of the i^{th} iteration. If the influence ordering is acyclic, then at this point every process has been assigned a priority. To see this, suppose that, at the beginning of a loop, L_i is found to be empty and a process has not yet been assigned a priority. Then that process's counter must be greater than zero (or it would have been placed in a list) and so that process must have an immediate predecessor that has not yet been assigned a priority. That immediate predecessor must also have an immediate predecessor which has not yet been assigned a priority. We can continue finding immediate predecessors that have not yet been been assigned priorities indefinitely. Since there are only a finite number of processes, we will eventually find a cycle in the influence ordering. Therefore, if the influence ordering is acyclic, then when L_i is found to be empty every process must have been assigned a priority.

It would not be difficult to use this algorithm manually to assign priorities to processes in a given simulation model. However, it might also be desirable to automate the assignment of priorities to processes. This would probably require some kind of static analysis of the model to determine which processes can immediately influence other processes. We are also lead to ask how computationally expensive the depth finding algorithm would be compared to the computational expense of the simulation.

The time required by the depth finding algorithm is proportional to, at most Nk, where 'N' is the total number of processes in the model and where 'k' is the maximum number of processes that any one process can immediately influence. To see this, first note that every process appears in L_i for at most one value of i, and appears in L_i exactly once. The main loop is repeated once for every non-empty L_i. On the i^{th} iteration of the main loop, the outer for-loop ("For each $P \in L_i$") is repeated once for every process in L_i. Thus, the outer for-loop is repeated at most once for every process in the model specification. Furthermore, on every iteration of the outer for-loop, the inner for-loop is repeated no more than k times. Thus, the time required for the depth finding algorithm is proportional to, at most, Nk.

Let 'e' be the average number of events to be simulated for each process. e is likely to be very large if every process exists throughout the simulation, and so e is likely to be much larger than k. It seems reasonable to assume that the time required for the simulation is approximately proportional to Ne. We therefore conclude that the cost of using the depth finding algorithm at the beginning of a simulation is not significant compared to the cost of the simulation. That is, using the depth finding algorithm to statically assign priorities to processes before the simulation begins would not significantly slow down the simulation.

5. DYNAMICALLY ASSIGNING PRIORITIES TO PROCESSES

We now discuss how the influence ordering could be enforced in a sequential simulation when processes can be created and destroyed during the simulation. We assume that whenever a process is created, it can be determined which processes can immediately influence that process and which processes can be immediately influenced by that process. We must first extend the definition of 'immediately influence'. If P_1 can create a process that can send a message to P_2 with timestamp equal to the time at which the process is created, then an event from P_1 that occurs at time t can cause a message to be sent to P_2 that has timestamp t. In that case, we would need to assign a higher priority to P_1 than to P_2. Therefore, we extend the definition of 'immediately influence' given in section 3 by stating that, in such cases, P_1 immediately influences P_2.

Since processes are created and destroyed during the simulation, the influence ordering may change during the simulation. Neither [Jefferson 1985] nor [Misra 1986] discuss simulations in which proceses can be created and destroyed. However, it seems likely that if a distributed simulation algorithm requires a model to have the predictability property, then in order to use that distributed simulation algorithm with a dynamic model (that is, one in which processes can be created or destroyed) that model must have the predictability property at all times. However, it might be possible for the influence ordering to change in such a way that a process that is, at one time, a predecessor of a second process is, at a later time, a successor of that process. Thus, if priorities are assigned to processes at one point during the simulation, they may have to be changed later in the simulation.

The most obvious way to dynamically assign priorities to processes is to use the depth finding algorithm given in section 4 to assign a new priority to each process whenever a new process is created. This would increase the cost of creating a new process, 'P', by an amount proportional to $N_P k_P$, where

'N_P' is the number of processes in existence when P is created and where 'k_P' is the maximum number of processes that any one process can immediately influence when P is created. Note that $k_P < N_P$ so that $N_P k_P < N_P^2$. The cost of using the depth finding algorithm whenever a process is created therefore depends on the number of processes that exist when each process is created. This number depends entirely on the model, but it seems likely that using the depth finding algorithm whenever a process is created would significantly slow down the simulation if the total number of processes created is large.

A better approach might be to assign priorities to processes only when simultaneous events are actually encountered during the sequential simulation. So long as simultaneous events are relatively rare, this would not significantly slow down the simulation. However, if simultaneous events occur relatively frequently, the overhead of assigning priorities to processes could again be significant.

A third approach to assigning priorities in a dynamic simulation is to assign a priority to each process as it is created, but to avoid re-assigning priorities to all processes whenever possible. The rest of this section is devoted to a discussion of how this could be done.

We use the term *priority value* to refer to the numeric value of the priority assigned to a given process. Recall that if one process has a smaller priority value than a second process, the first process is considered to have a higher priority than the second process. We also refer to the processes that exist at the beginning of the simulation as the *initial processes.* At the beginning of the simulation, priorities must be assigned to initial processes using the depth finding algorithm.

Let $D = \lfloor (p_{\max} - p_{\min})/(d_0 + 1) \rfloor$, where '$p_{\max}$' is the largest possible priority value, '$p_{\min}$' is the smallest possible priority value, 'd_0' is the greatest depth of any initial process in the influence ordering, and '$\lfloor x \rfloor$' is the greatest integer that is less than or equal to x. The priority value of every initial process is multiplied by D. This should be done to leave as much room as possible between consecutive priorities so that priorities may be assigned to as many newly created processes as possible. $\lceil D/2 \rceil$ (the least integer greater than or equal to $D/2$) is then added to each priority. This leaves as much room as possible to assign priorities to newly created processes that have higher or lower priority than any initial process.

Whenever a process 'P' is created during the simulation, P must be assigned a priority. This is to be done by computing the greatest priority value, 'p_1', of any of process that can immediately influence P, and the least priority value, 'p_2', of any process that can be immediately influenced by P. Note that the time required to compute p_1 is proportional to the number of processes that can immediately influence P and the time required to compute p_2 is proportional to the number of processes that P can immediately influence. If $p_1 < p_2$ and $p_1 < \lfloor (p_2 - p_1)/2 \rfloor$, then we assign priority $\lfloor (p_2 - p_1)/2 \rfloor$ to P. (Recall that lower numeric values correspond to higher priorities.) This ensures that P has lower priority than any process that can immediately influence P and has higher priority than any process that it can immediately influence.

If, however, $p_1 \geq p_2$, then the influence ordering has changed (due to the creation and destruction of processes) and there is no way to correctly assign a priority to the new process relative to the priorities that have been assigned to existing processes. If $p_1 = \lfloor (p_2 - p_1)/2 \rfloor$, then even if $p_2 > p_1$, there is no integer value between p_2 and p_1 that can be assigned as a priority to the new process. In either case, all priorities must be recomputed using the depth finding algorithm. We then multiply those priorities once again by $D = \lceil (p_{\max} - p_{\min})/(d_{\max} + 1) \rceil$, where '$d_{\max}$' is the maximum depth of any process in the new influence ordering, and add $\lfloor D/2 \rfloor$ to each priority. We are then ready to continue the simulation and assign priorities to new processes as they are created.

6. SUMMARY

We studied the handling of simultaneous events in distributed simulation and concluded that the results of a distributed simulation are the same as those of a sequential simulation in which a particular assignment of priorities is used to resolve time ties. We described this assignment of priorities and gave an algorithm that can be used to compute these priorities before the simulation begins. This would be necessary, for example, if a sequential implementation and a distributed implementation of the same language were being developed, or if a sequential simulation were being used to validate or to evaluate a distributed simulation of the same model. In that case, some mechanism would be needed to ensure that the two implementations handled simultaneous events in exactly the same way. Finally, we showed how to dynamically assign priorities to processes in a simulation in which processes can be created and destroyed during the simulation, and we discussed the costs and the limitations of this method.

ACKNOWLEDGMENT

This work was partially supported by the CASE (Computer Applications and Software Engineering) Center at Syracuse University, Syracuse, New York.

REFERENCES

Cota, B. A. and R. G. Sargent (1989), "An Algorithm for Parallel Discrete Event Simulation Using Common Memory," In *Proceedings of the 22nd Annual Simulation Symposium*, A. Rutan, Ed. ACM, 23-31.

Cota, B. A. and R. G. Sargent (1990), "A New Version of the Process World View for Simulation Modeling," CASE Center Technical Report no. 9003, Syracuse University, Syracuse, NY.

Jefferson, D. R. (1985), "Virtual Time," *ACM Transactions on Programming Languages and Systems*, 7, 3, 198-206.

Misra, J. (1986), "Distributed-Discrete Event Simulation," *ACM Computing Surveys*, 18, 1, 19-65.

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.

Zeigler, B. P. (1977), *Theory of Modelling and Simulation*, Wiley, New York.

Zeigler, B. P. (1984), *Multifacetted Modelling and Discrete Event Simulation*, Academic Press, London.