Proceedings of the 1996 Winter Simulation Conference
ed. J. M. Charnes, D. J. Morrice, D. T. Brunner, and J. J. Swain

PRINCIPLES OF CONSERVATIVE PARALLEL SIMULATION

David M. Nicol

Department of Computer Science
Dartmouth College
Hanover, NH 03755

ABSTRACT

This tutorial describes considerations in writing parallelized discrete-event simulations. We identify key principles behind various synchronization methods tailored to simulate complex systems.

1 INTRODUCTION

The use of parallel computers to execute discrete-event simulations has been a topic of active research interest for the last fifteen years. The seminal work by Chandy and Misra was done well before parallel computers were in common use, and (perhaps surprisingly), was principally motivated by concerns other than performance. What has become known as the "null-message" algorithm (Chandy and Misra, 1979) was developed as an example of a distributed algorithm to be validated; Chandy and Misra's later work on detecting and breaking deadlock also retained a strong flavor of algorithm validation. Jefferson's seminal Time Warp paper (Jefferson, 1985) had interests in proving the correctness of the method, but importantly was proposed as a way to avoid "blocking" inherent in the Chandy and Misra work, so as to improve upon available parallelism and possibly improve performance.

Since these early times, a great deal of research has gone into new algorithms for synchronization and optimizations for the old ones. So-called "conservative" algorithms have their philosophical base in the Chandy and Misra approach. A defining characteristic of a conservative approach is that no computation is ever done that might possibly be incorrect. A defining characteristic of so-called "optimistic" approaches is that some incorrect computations may be done, but the synchronization system is capable of automatically detecting and correcting such errors. Reynolds made a number of important refinements to this classification (Reynolds, 1988), for our purposes two stand out. Optimism has the components of *aggressiveness*—doing computations that may turn out to be incorrect—and *risk*—exporting the results of an incorrect computation on one processor to other processors. As we will see, it is possible, even desirable, to use a protocol where there is aggressiveness but not risk.

As medium-scale parallel processors become common, it is natural for a simulationist to ask "Should I use parallel simulation?", and if the answer is yes, to ask just how that simulation ought to be constructed. One option is to use a parallel simulation package, the other is to handle the synchronization issues yourself, in your own code. One principle disadvantage of using a package is that it may not work on the architectures available to you. Another is that these (few) packages are products of universities, so you cannot expect much support as you encounter difficulties or discover that the package does not allow you to to model quite the way you had intended. The principle advantage of parallel simulation packages is that the details of synchronization are largely hidden from you. The advantage of writing your own simulator from scratch is that you are fully aware of all the details, and can optimize the synchronization to the problem you are simulating. The disadvantage is that it is definitely more work to build your own synchronization mechanisms, and, depending on the type of synchronization you use, as your model evolves you made need to alter the synchronization strategy. Nevertheless, home-grown parallel simulators are viable for target applications where the semantics are well understood and unlikely to change, e.g., queueing networks, Petri nets.

There is an extremely important application area for roll-your-own parallel simulations emerging, the "High Level Architecture" (HLA) specifications that will determine how to construct a federated distributed simulation from cooperating simulators. The HLA specs principally call for conservative synchronization between individual simulators, and explicitly provide a mechanism for a simulator to report its "lookahead"

to the controlling agent. Consequently, understanding the principles of conservative synchronization for that context will be quite important.

The purpose of this tutorial is to provide those who would build their own simulators with the concepts necessary to approach such a task.

2 DISCRETE EVENT SIMULATION

2.1 The DES Paradigm

Discrete-event simulation is a algorithmic paradigm that is suitable for emulating complex systems whose "state" is discrete, and which changes at discrete instants in time. Queueing networks or Petri nets are examples—their states are vectors of queue lengths or place markings, the state-vector changes when a job departs or a transition fires. It may happen that other information is also needed in the state, e.g., if a queue does not have exponential service then some record of residual service time must be part of the state information.

The key idea behind discrete-event simulation is to focus the computational effort of the program on only those instants where the state changes, and only at those portions of the state that change. This offers large computational savings over "time-stepping" methods that advance the entire state-vector at each of many small time-increments. Some sort of time-stepping is generally required to simulate so-called "continuous systems" whose state evolution behavior is described using differential equations. Whether one uses discrete-event or continuous simulation techniques depends entirely on the nature of the model. It is worth noting in this discussion that continuous simulation models frequently admit to parallelization in a much more straightforward fashion than discrete-event simulations, because the synchronization structure is much simpler.

2.2 World Views

Simulation modelers frequently describe models as having an "event" world view, or a "process" world view. In the event world view the behavior of the model is described in terms of how the simulation model changes when an event is processed. For instance, the code for an event-oriented queueing simulator will identify events such as *DepartQueue*, *ArriveQueue* and with each event will proscribe code that affects the simulation state as a result of the event occurring.

A process-oriented view expresses the logic of the simulation at a higher level of abstraction. In the case of the queueing network it might describe the simulation from the point of view of a job—join a queue, wait for service, join another queue, and so on. The critical difference is that the job's behavior can generally be expressed without reference to the state of the queue, something that is not possible in an event-oriented simulator. Details concerning contention resolution are buried—from the modeler's point-of-view—in what ever process-oriented simulation package is being used.

The choice of world view does have a significant impact on parallelization. By its nature, implementing process orientation is a delicate business. Process-oriented simulations are almost exclusively developed using software packages; to go parallel with a process-oriented view means either to parallelize these packages or write one's own parallelized process-oriented system. The later task can be quite daunting if one is to use optimistic synchronization. Such simulators need to be able to save and restore the state of the *simulator*, as well as the simulation model state. We have had success using conservative techniques with existing process oriented simulators, but in ways limited by the need to not change the base process-oriented simulator.

Most simulations written from scratch are event-oriented. This makes the simulator easier to write, but makes the development of simulation models somewhat more difficult. If one is tailoring the synchronization protocol to the simulation problem, then an event oriented view will almost certainly be the view of choice.

3 SYNCHRONIZATION

Fundamentally, the heart of a discrete-event simulation is an event-list (even process-oriented simulators use these in their internal structure). Causality is maintained in a serial simulator because the next event executed is always the one with least time-stamp in the entire simulation model. This ensures that any event that might alter state information upon which the next event depends has already been executed. A useful way to view a parallel simulation is as a set of cooperating serial simulators, each with its own event list. Now, however, an event executed by one simulator may change model state upon which events in another simulator depend, which is tantamount to one simulator changing the event list of another. One way to ensure causality is to prohibit a simulator from executing its next event until it is sure that no other simulator will cause an earlier event to be inserted, or the next event to be removed. Such an approach is said to be *conservative*. An *optimistic* method will not block a simulator; instead it goes ahead and executes the next event, but first

saves enough state information so that it can return ("rollback") to that point again if it needs to. An optimistic system detects when a "straggler" event (one with a time-stamp that is smaller than the current simulation clock) is received, does a rollback to that time and undoes the effects of wrong computations it might have performed at times larger than that of the straggler message. Another important distinction is that it is possible to have one simulator execute optimistically but temporarily withhold affecting other simulators with its results until it is certain that its inter-processor interactions are correct. Such an approach is said to be "aggressive" but "risk-free".

The main question for a conservative method is how a simulator can determine if its next event is safe to process. There is no single answer, it depends on the simulation model. The general label for one's ability to extract such information from a model is called *lookahead*. The advantage to an aggressive but risk free approach is that it avoids the necessity for explicitly computing lookahead—it allows the computation to *observe* lookahead (in a way we'll make clear later)—but avoids disseminating incorrect information.

4 LOOKAHEAD

Lookahead is the key ingredient for all conservative synchronization methods. A loose definition is that it is the ability of a simulator to predict future behavior with respect to modifying the event lists of other simulators. More precisely, a process p has lookahead with respect to process q if p's simulation clock is at time s, and yet p can determine that under no circumstances will it insert or delete an event from process q's event list with time-stamp $t > s$. Even so, this definition does not capture the many aspects of lookahead, knowledge of which is important when developing a synchronization strategy for a given simulation problem. We now explore these aspects.

4.1 Subtleties of Lookahead

To help motivate a classification of lookahead, we first look at some situations that illustrate different facets of lookahead.

Models of just about any kind of network involve movement of objects (e.g., packets, jobs, parts, patients, vehicles, etc.) and contention among these objects for network resources. The way one process typically affects another is by sending it an object, after that object has acquired and used some resource. A concrete example of this is a queueing network. The network is partitioned among processors; a job contends for service, and after receiving it is routed to another another queue, possibly to one on a different processor.

Consider a resource mapped to process p, such that after an object uses that resource it migrates to contend for a resource that is mapped to a different process q. The nature of the resource allocation scheme affects p's lookahead. If allocation is non-preemptive (that is, once a resource is allocated to an object it is not released until that object has completed its service), then p knows that an object in service will remain in service without the possibility of another object acquiring that same resource, and possibly releasing it and migrating to q sooner than would the original process. Another factor is the mechanism by which the resource is released and the object migrates. In a queueing network, a job frequently receives a known amount of service, and its departure time is known at the point the job enters service. In other models it is possible that the object holds the resource until told to release it. p's ability to look ahead relies then on being able to predict when such release directives are issued.

There are further complications. It might be that an object does not know where it is migrating until its point of departure. This happens, for instance, if a job leaves one queue and joins the shortest queue from among a set, at the point of departure. It might be that at the point the object begins its service p knows what the state of the object will be when it migrates, it might be that it does not (for instance, if the object carries with it information relating to the simulation state at the time of departure).

Another type of lookahead arises when prior to simulating the timing of activities, certain measurements are made that yield lower bounds on delays until those activities occur. For example, in the trace-driven simulation of a shared memory multiprocessor, one can analyze an individual processor's trace file to determine the numbers of local references between local references. By assuming that each local reference is resolved as quickly as possible one can compute a lower bound on inter-global-reference delays. As a process simulates the memory behavior described by a trace, it can look ahead over the remaining local references until the next global one (which presumably affects another simulation process).

As a final example of the subtleties of lookahead, we observe that lookahead may be "sampled", in the sense of generating random numbers in order to obtain lookahead, and "observed". An example of the former type arises in stochastic simulations, where random variables may be sampled before they are actually needed, for the purpose of computing lookahead. A common instance of this is to sample the service time of the next job to receive service at a queue,

before the job actually arrives. If the queue is non-preemptive, we know that at least the pre-sampled service time must elapse before the next job departs the queue. A case of observed lookahead occurs in the parallelized direct-execution simulation of computer programs. The program is executed between points where it interacts with other simulators, and the number of instructions executed in such an interval are measured. To exploit this we can interleave execution of the computer program to get the measurements, and the timing simulator.

4.2 Dimensions of Lookahead

We now classify dimensions of lookahead, knowledge of which will prove useful when we develop synchronization protocols.

4.2.1 Time / Content Lookahead

Our informal definition of lookahead focuses on the temporal aspect, and has nothing to say about the content of the next event p may send to q. Consider: p may know simply that it will not affect q at a time less than t, or, it may know that it *will* affect q at time t, and it knows how it will do so. The latter case is what we call "content" lookahead; its presence clearly indicates that p has a better ability to predict future behavior. Intuitively, process q may be able to use content lookahead in a more advanced way than purely time lookahead.

Another possibility is that p knows that it will affect q at time t, but will not know exactly how it will do so until later—perhaps only at time t. From q's perspective, knowledge that *something* specific will occur at time t may be handled differently from knowledge that nothing will happen before time t. The difference is subtle, but may have an impact on how q is implemented.

4.2.2 Bounded / Exact Time Lookahead

There is a difference between p knowing that it will not affect q before time t, and p knowing that it will not affect q before time t and that it *will* affect q at time t. The former case is "bounded-time lookahead" and the latter case "exact-time lookahead". The distinction has an effect on the way that q deals with the information. In fact, if p has content lookahead and exact-time lookahead, then the lookahead information suffices to actually deliver the forecast event to q.

4.2.3 Directed / Semi-directed / Undirected Lookahead

Our loose definition of lookahead was given in terms of p knowing how it might affect q in the future. This is an example of what we'll call "directed" lookahead. It might be the case that p knows that it will not affect some set of processes before time t. For instance, if a server routes a job to the queue with shortest length among a subset of queues at the instant of departure, then its lookahead is limited to that set of queues. This we call "semi-directed" lookahead. Finally, if p knows only that it will not affect *any* process before time t, we say that it has "undirected" lookahead.

4.2.4 Conditional / Unconditional Lookahead

An important class of methods rely on exploiting something called "conditional lookahead". The idea is that at time s, process p knows that it will not affect process q before time t, *provided* that the state of p does not change before time t. An example is a job in a preemptive queue, nominally scheduled to depart at time t. If no other higher priority job enters before time t and interrupts it, the lookahead is to t. The lookahead in this case is unconditional if the queue is non-preemptive.

5 USING LOOKAHEAD

Synchronization protocols vary in how they use the lookahead that applications provide. One distinction is whether a protocol is synchronous or asynchronous (or both). This distinction is drawn based on how the lookahead is used and distributed. In an asynchronous protocol, when process q is specifically told that p might affect it at time t, process q then blocks at t until told otherwise. A synchronous protocol involves some sort of global reduction synchronization. Processes that participate in the reduction provide lookahead values to it; the result of the reduction governs how far a process may advance before blocking. Synchronous protocols find a way to combine lookahead information from multiple sources.

Some synchronous protocols use the notion of conditional lookahead. For instance, the YAWNS (Nicol et al. 1989, Nicol 1993) protocol works using two principle ideas. First, that every message sent by one process to another is pre-sent in time and content. For instance, if a job enters service at a non-preemptive queue and we know its service time and routing destination, then at the point it enters service its arrival at the next queue may be reported. The second idea is that a process be able to examine its state and determine a lower bound on the time of the

next message it sends, *conditioned on the assumption that no further messages are received.* Each process is thus computing conditional lookahead. Each offers that conditional lookahead to a global min-reduction. The value produced by the reduction is a simulation time up to which all processes may advance asynchronously of all others, without concern for receiving a message in its past.

Another way lookahead is combined is through "lookahead propagation networks". This can be effective when the underlying simulation model is of a network where one can view the nodes as adding delay to received objects, then propagating the objects. The fundamental idea is best described with an example. Say we have a queueing server that is presently idle but has pre-sampled the service time of the next arrival. If the server has no idea of when next an arrival might occur, it must assume the worst, that an arrival will come immediately. That estimate can obviously be improved if a way is found to bound the arrival time of the next arrival. The lookahead propagation network does this. The idea is to compute minimal length paths through a network that has the topology of the simulated network; nodes in the lookahead propagation network are weighted with the value of the service time to be given to the next arrival. Receiving a lower bound on the arrival time of a job from any source, an idle node adds to this bound its presampled service delay and offers to its successors the sum as a bound on when next it will propagate a job to them.

6 PROTOCOLS

As a concrete application of these ideas, we now look at some synchronization protocols and observe how they differ in their use and requirements of lookahead.

6.1 Null Message Protocol

The original protocol (Chandy and Misra, 1979) (discovered independently by Byrant, 1977) describes a distributed simulation in terms of "logical processes" (LPs), and "channels" between those processes. It is assumed that the channels are static; one might imagine a directed graph where LPs are nodes and channels are edges. One LP affects another by sending it a message over a channel. The message has a time-stamp on it, the time at which it affects the receiver. All such communication is point-to-point. It is assumed that messages from one LP to another appear at the receiver in the order they were sent. A distinction is made between having the message show up at the receiver, and having that LP "accept" the message, e.g., pull it out of the channel and incorporate it into its own state. An LP can detect when an input channel is empty.

An LP can be certain that it does not execute an event out of order so long as it never accepts a message with a time-stamp less than that of the next internal event the LP has to perform. Consequently, if any of its input channels are empty, the LP must block. Without further structure, this blocking rule makes it quite easy to deadlock; consider—a channel may be declared between two LPs, but through the vagrancies of random sampling no message is ever sent through that channel. To avoid this specific problem one may use so-called "null-messages". The rule becomes that when an LP accepts a message of any kind with time-stamp t, it posts a null-message on each of its output ports for which the next message time is not known. One might naively put time-stamp t on such null-messages, so that a null-message is interpreted as a declaration from sender to receiver that the sender has advanced to time t and so implicitly will send no subsequent message with time-stamp less than t. This is still not enough to avoid deadlock, lookahead needs to be incorporated. The net effect is that the null-messages sent in response to an accepted message have time-stamps greater than the accepted message. The increase in time-stamps, say v, reflects how good the lookahead is at the LP. The value v depends, of course, on the model under simulation. Intuitively, the increase in time-stamp value reflects a priori knowledge that *any* response by the LP to *any* accepted message will require at least v units of simulation time to "propagate" through the LP. As we have discussed earlier, that value may be derived from pre-sampling random variables or by knowledge that any such delay (even stochastic) exceeds some fixed value v.

Now consider our earlier classifications of lookahead. Null-message type protocols are seen to use time-lookahead as only the message time-stamps are involved in the sequencing. These protocols also use bounded-time-lookahead in that the values placed on null-messages are only bounds. The protocols use semi-directed lookahead, provided that the LP-channel graph is not a complete graph—a null message is sent to all receivers of output channels from an LP, but it is sent only to those receivers. Finally, null-message type protocols use unconditional lookahead.

6.2 Appointments

Imagine a simulation with the same type of static LP-channel structure as is assumed by null-message protocols, but with additional assumed intelligence on the part of LPs. We now require an LP to maintain on each of its output channels an "appointment"

time (Nicol and Reynolds, 1984; Nicol 1988). One can think of the transferral of an appointment from sender to receiver as a null-message, but with some critical differences. First, the semantics of message acceptance becomes different from null-messages. In the null-message approach a message is not accepted from a channel before the LP is prepared to process that null-message, at its posted time-stamp. In an appointment-based protocol the receiver understands the appointment as a promise by the sender not to transmit any message with larger time-stamp. For its part the receiver tacitly promises not to advance its simulation clock beyond any appointment. Consequently the receiver may accept the appointment at any time, and bases its blocking decisions on accepted appointment values rather than unaccepted null-message times. The practical import of this is that at the point it accepts an appointment, the information there contained may allow the receiver to improve upon an appointment it provides to yet another LP. For example, suppose process p gives process q an appointment with time-stamp t. Process q is a stochastic queueing server, and knows that it routes all jobs accepted from p to some other process r. p's knowledge of a lower bound on the arrival time of the next job from p may allow it to immediately improve its appointment with r, even if the appointment time lies far into the future.

Whereas (Nicol and Reynolds, 1984) introduced the concept of appointments, it was only later in (Nicol, 1988) where they were actually used. The proposed appointments calculation procedure also introduced the idea of using a "shadow network", or as we have termed it here, a lookahead propagation network. Shortest-path computations on the the shadow network were used to push lookahead forward across LPs.

According to our classification, appointment-based protocols differ from null-message-based protocols in their ability to use content-lookahead (if available), and directed lookahead. A very important difference is that the null-message protocol can be implemented using very little model-specific information. An aggressive appointment protocol may involve some complex calculations and logic on the part of the LPs. The tradeoff is increased protocol complexity in exchange for sometimes markedly increased performance.

6.3 PUCS

The PUCS protocols (Parallel Uniformized Continuous time Simulator) were developed specifically for simulating continuous-time Markov chains (CTMC) (Heidelberger and Nicol, 1993). These protocols are essentially appointment-based; their appointments exploit the mathematically structure of CTMCs.

The simplest form of PUCS requires that each sending process identify for each process to which it may send messages a maximum "rate" at which those messages may be dispatched.

A typical example is a queue p with n servers, each with service rate μ. Jobs leaving p are routed to queue q with probability α. The fastest possible rate at which p routes jobs to q is when all of its servers are busy; that rate is $n \times \alpha \times \mu$.

Using uniformization, the mathematical structure of CTMCs allow the simulation to be performed (at least conceptually) in two phases. In the first phase, each process generates a synchronization schedule for all other processes it may affect, and conveys that schedule to them. If the maximum rate at which p affects q is λ_{pq}, then the spacing of synchronization points from p to q is random, with an exponential distribution having rate λ_{pq}. These schedules govern synchronization in the second phase, for the synchronization points are appointments. When process p reaches an appointment time t that it sent to process q, p randomly decides to actually affect q, or not. It does so by measuring the actual transition rate at which it dispatches messages to q, given p's state at t, call this rate a_{pq}. Then, with probability a_{pq}/λ_{pq} it chooses to undergo a state transition that affects q; with complimentary probability it does not, and merely sends q a message reporting a "pseudo" event instead. In the example of a multi-server given earlier, if k out of n servers are busy at time t, then an actual state transition occurs at t with probability k/n.

From the point of view of the lookahead classification scheme, PUCS has bounded-time-lookahead, and directed lookahead. It is properly seen as a specific way to generate and manage appointments, for a specific problem class.

6.4 TNE

The *Time-of-Next-Event-Algorithm* protocols (Groselj and Tropper, 1988) compute "link times" between LPs, a link time being a lower bound on the time of the next message sent from one LP to another. A link time is clearly an appointment. It computes link times using (i) knowledge of the time of the next event in each LP, (ii) knowledge of a minimum delay added by an LP to any message that it processes, (iii) knowledge of existing link times, (iv) a shortest path algorithm. Variants of TNE have focused on how to propagate shortest path information across processors. The TNE algorithms are thus seen to computing appointments based on the topology of the LP

network and the lookahead of minimum delays across LPs. With respect to our classifications, TNE algorithms have bounded-time lookahead, directed lookahead, and do not have content lookahead. The lookahead computed is unconditional.

6.5 Bounded Lag

Like the other conservative mechanisms described so far, the Bounded Lag protocol (Lubachevsky,1989) assumes a static LP interconnection topology. Like some of the other conservative mechanisms, it relies upon a lookahead propagation network. Two things distinguish the Bounded Lag algorithm from others. First, it is synchronous. It does a global lookahead calculation, involving all processors, to identify for each LP a point in simulation time up to which it may safely execute events. That point is identified through analysis of the lookahead propagation network to identify for each LP the earliest time at which it might be affected in the future by another LP, OR, a lower bound on that time, the "bounded-lag". The key idea here is that for any LP, given some lag B, an analysis of the LP interconnection network and lookahead at each of the LPs can determine a sphere-of-influence, a set of LPs that might possibly affect the LP within B simulation time units. If none can, then B units from the present time serves as the LPs upper limit. What is critical here is that the lag allows one to bound the effects of all LPs outside of the sphere-of-influence, without explicitly considering them.

The lookahead discussed in the Bounded Lag papers focuses on minimum propagation delays between LPs (equivalent to minimum service times at queues), and so-called opaque periods which are localized periods during which the time of an LP's next action is insensitive to the receipt of any further messages. This is like a non-preemptive server.

This protocol has bounded-time lookahead, both directed and undirected lookahead (the lag is undirected). Like the appointments protocol it does not specify precisely how the opaque periods or through-LP lookaheads are defined, so content-lookahead might be exploited (although no examples of that have have appeared in the bounded lag literature). Applications of the bounded-lag algorithm have always used unconditional lookahead, but the method is robust enough that conditional lookahead can be transformed into unconditional lookahead in the course of the lookahead phase.

6.6 Conditional Events

The "Conditional Events" paper (Chandy and Sherman, 1989) recognized that in ordinary discrete-event simulation, without knowledge of model-specific information, all events on the event-list are "conditional" in the sense that any one of the *except the one with least time-stamp* might be removed as a consequence of executing an event with smaller time-stamp. Of course, with more model specific information we might identify events on the event list that are unconditional, and we might identify events that do not affect existing events on the event list.

In a parallel simulation context a process has a set of events; some are known to be unconditional, some are known not to affect existing events on this processor's event list, some are conditional. However, the one with least time-stamp is not unconditional, at least not before we can establish that the process will not be affected by another at an earlier time.

Chandy and Sherman showed how to transform unconditional events into conditional events. Each process offers to a global min-reduction the least time-stamp among all its conditional events. The reduction delivers the least such to all processes, and this defines a simulation time up to which all processes may simulate concurrently.

Details of how one identifies conditional or unconditional events are model dependent. The next protocol we examine, YAWNS, gives some refinement to the concept.

The conditional events approach is synchronous; its lookahead is bounded, conditional and undirected. Content-lookahead might be employed in identifying unconditional events on a model specific basis.

6.7 YAWNS

The YAWNS (Yet Another Windowing Network Simulator) protocol (Nicol et al., 1989; Nicol, 1993) is an application of conditional event approach. Its principle contribution is to show by mathematics and implementation the utility of a simple conservative tightly synchronized approach in situations where there is ample parallel workload, and lookahead of a specific type.

The key ideas behind YAWNS were described earlier. The specific resolution YAWNS provides to the conditional event proposal is identification of unconditional events (service completions), and conditional events (next messages out from a process).

6.8 The Event Horizon

As a final topic we examine an approach that resembles the YAWNS approach, except that the method

is not completely conservative. The concept of "event horizon" (Steinman, 1991) codifies the logic behavior conditional events: given all processes are synchronized at t, the event horizon is the simulation time stamp of the next message to pass between any two processes. Strictly conservative applications of conditional event approaches compute lower bounds on the event horizon. Yet, as pointed out in (Steinman, 1991) that isn't necessary. So long as one saves state and withholds sending messages until it is certain that they should be sent, one can *compute* up to the event-horizon. From a synchronization point, each process executes events (saving state) up to the point where its time of next event exceeds the time-stamp on any message the process generated but hasn't yet delivered. That time-stamp is the processes local event horizon. A min-reduction on all local event horizons yields the global event horizon. All messages generated by events with time-stamps no greater than the horizon are delivered; as a result, a processor may receive a message with a time-stamp smaller than its local event horizon; that process will roll back. But, importantly, such rollbacks are entirely local to the process. They do not propagate. Construction of a simulation using this technique is considerablely simpler than one that goes fully optimistic.

While the event horizon idea is not strictly conservative, it is a natural and useful extension to ideas explored in the conservative context. It should be considered in cases where predicted lookahead is difficult to acquire.

7 SUMMARY AND CONCLUSIONS

This tutorial has highlighted key ideas useful for writing one's own parallel discrete-event simulation.

REFERENCES

Bryant, R.E. 1977. Simulation of packet communication architecture computer systems. *MIT-LCS-TR-188, Massachusetts Institute of Technology.*

Chandy, K.M. and J. Misra. 1979. Distributed simulation:a case study in design and verification of distributed programs. *IEEE Trans. on Software Engineering*, 5, 440-452.

Chandy, K.M. and Sherman, R. 1989. The conditional event approach to distributed simulation. In: *Proceedings of the 1989 Conference on Parallel and Distributed Simulation*, 93-99, SCS Simulation Series.

Groselj, B. and Tropper, C. 1988. The time of next event algorithm. In: *Proceedings of the 1988 Conference on Parallel and Distributed Simulation*, 25-29, SCS Simulation Series.

Heidelberger, P. and D.M. Nicol. 1993. Conservative parallel simulation of continuous time Markov chains using uniformization. *IEEE Trans. on Parallel and Distributed Systems*,4.

Jefferson, D.R. 1985. Virtual time. *ACM Transactions on Programming Languages and Systems*, 3, 404-425.

Lubachevsky, B.D. 1989. Efficient distributed event-driven simulations of multiple-loop networks. *Communications of the ACM*, 32, 111-123.

Nicol, D.M. and Reynolds, P.F. Jr. 1984. Problem oriented protocol design. In: *Proceedings of the the 1984 Winter Simulation Conference*, 471-474.

Nicol, D.M. 1988. Parallel discrete-event simulation of FCFS stochastic queueing networks. In: *Proceedings of the ACM/SIGPLAN PPEALS 1988. Parallel Programming: Experiences with Applications, Languages and Systems*, 124-137. ACM Press.

Nicol, D.M., Micheal C., and Inouye, P. 1989. Efficient aggregaton of multiple LP's in distributed memory parallel simulations, In: *Proceedings of the 1989 Winter Simulation Conference*, 680-685.

Nicol, D.M. 1993. The cost of conservative synchronization in parallel discrete-event simulations. *Journal of the ACM*, 40, 304-333.

Reynolds, P.F. Jr. 1988. A spectrum of options for parallel simulation. In: *Proceedings of the 1988 Winter Simulation Conference*, 325-332.

Steinman, J.S. 1991. SPEEDES: synchronous parallel environment for emulation and discrete event simulation, In: *Proceedings of the 1991 Conference on Parallel and Distributed Simulation*, 95-103, SCS Simulation Series.

AUTHOR BIOGRAPHY

DAVID M. NICOL received the Ph.D. in Computer Science from the University of Virginia in 1985 and is presently an Associate Professor of Computer Science at Dartmouth College. He is an area editor for the ACM's *Transactions on Modeling and Computer Simulation* and an associate editor for the *INFORMS Journal on Computing.* He has served as the 1989 Program Chairman and the 1990 General Chairman of the Workshop on Parallel and Distributed Simulation (PADS), has served on the PADS Steering Committee, and in 1996 was the Program Chairman for the ACM Sigmetrics Conference. His interests are in parallel simulation, performance analysis, and algorithms for mapping parallel workload.