A NOVEL APPROACH TO SEQUENTIAL SIMULATION

Devendra Kumar

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

TR-85-14 July 1985

ABSTRACT

We present a novel approach to sequential simulation. In this approach we do not require events to be simulated in the chronological order of their occurrence. Instead, at any point in simulation all guaranteed events are simulated right away. This approach reduces the number of event list insertions in a number of simulation systems. In some cases it eliminates the need for event list altogether. It also reduces the number of scheduled events that do not take place, i.e., get cancelled later in the simulation. Sometimes memory requirements may be reduced as well. We illustrate the approach with an example. The approach is based on a distributed simulation algorithm.

This work was supported by Air Force Grant AFOSR 81-0205.

Table of Contents

1. Introduction

A fundamental convention in traditional simulation is that events are simulated in their chronological order of occurrence. The main reason behind this convention is as follows. An event simulated later but with an earlier time of occurrence may affect an event simulated earlier but with higher time of occurrence, making it incorrect. However, we observe that often in practice certain events can be guaranteed to be correct, and they are unaffected by the simulation of other events of earlier times of occurrence. We take advantage of this in our approach to simulation. We do not require events to be simulated in chronological order; rather, whenever an event is guaranteed to occur it is simulated right away.

The main advantage of this approach is that insertions of scheduled events (as in traditional simulation) can be reduced in number. One can often reduce the number of computations of scheduled events that would get cancelled later in traditional simulation. Sometimes this approach also results in reduced memory requirements.

The approach is derived from a distributed simulation algorithm. In distributed simulation, the simulator consists of a set of communicating processes which are assigned to several processors. We have adapted the algorithm to the case of a uniprocessor system. Due to the uniprocessor environment, several simplifications result, and new issues arise (e.g., scheduling of the processes). We have modified the algorithm accordingly.

2. The Approach

A system to be simulated consists of a set of *entities* that interact with each other. At discrete instants of time events occur – each event is caused by an entity and its occurrence affects the future behavior of zero or more entities. For example, an event in a communication network could be the sending of a message from one process to another; in this case the sender and the recipient processes are affected by this event. On the other hand, the event of broadcasting a message affects all the processes in the system.

We briefly review the traditional event driven simulation here. The simulation program maintains a list, called the *event list*, of scheduled events that might occur in the future. At an abstract level, a scheduled event can be defined by a tuple [e,t,i,S] where e is an identifier of the event, t is the time at which event e is supposed to occur, i is the entity that would cause e, and S is the set of entities that would be affected by it.

To simulate an event, the simulation program finds in the event list the scheduled event [e,t,i,S] with the minimum occurrence time t, and causes its entity i to simulate it, then advances the simulation clock to t. The entities in set S may be affected by the event occurrence – some of their old scheduled events may be deleted from the event list and new scheduled events may be inserted into it.

At a given point in simulation, a scheduled event [e,t,i,S] is said to be *guaranteed* if, based upon the events simulated so far, it is determined that this event will definitely be simulated. In other words, simulation of any other events before this scheduled event can not cancel it. As noted above, in general not all scheduled events are guaranteed. This results in the following fundamental convention in traditional simulation — events are simulated in the chronological order of their times of occurrence. In other words, an event is simulated only after all events of earlier times of occurrence have been simulated. (This convention is also followed in *time driven simulation*, and not just in *event driven simulation*, for the same reason.)

We observe, however, that often in practice many scheduled events are indeed guaranteed. For example, consider a FCFS queue with the convention that arrivals of input jobs are simulated in chronological order. Here the scheduled events of service completions are guaranteed to be simulated, since the arrival of new jobs can not cancel them. In our scheme we simulate such guaranteed events right away, instead of first depositing them into the event list and then waiting for the simulation clock to reach that time. More specifically, whenever an entity i computes an event e that is guaranteed to occur at time t, it goes ahead and simulates it. The tuple [e,t] is then deposited in a buffer $B_{i,j}$ for every other entity j affected by the event. An entity j computes its events based on the information it has received from its input buffers $B_{i,j}$, for various entities i.

Since events in the whole system are not being simulated in chronological order, there is no simulation clock maintained by the program. However, for any two entities i and j, all the events that are caused by i and affect j are simulated in chronological order. Thus, when a tuple [e,t] is deposited by i in the buffer $B_{i,j}$, entity j knows the entire history of all events that are caused by i and affect j up to time t. This helps j in its computation of future events.

Obviously, by computing guaranteed events and depositing them in buffers, we are avoiding the corresponding insertions in the event list, as required in traditional simulation.

The simulation program cycles through the entities – each entity computes guaranteed events and deposits them in the corresponding buffers. Each entity also discards from its input buffers those elements that are no longer needed for future computations.

What happens when no entity can compute a guaranteed event? In such a situation we revert back to the event list mechanism – the next scheduled events are computed and the event with the minimum occurrence time is simulated. Subsequently, guaranteed events are simulated till the above situation arises again. Thus in the total simulation, the simulator keeps alternating between "compute and simulate guaranteed events" and "compute and simulate the scheduled event with the minimum occurrence time" phase. (Henceforth we will call these phases: A and B, respectively.)

Advantages Of The Approach

In our approach we simulate guaranteed events right away. This avoids the corresponding insertions in the event list. Insertions in the event list may be quite time consuming, since the elements in the list need to be maintained in the order of increasing time values.

Consider a scheduled event in traditional simulation that gets cancelled later during the simulation. Obviously, the time involved in its computation and insertion in the event list goes wasted. This can happen, for example, in a priority queue where the arrival of a job of higher priority will preempt the current job in service. In our approach the number of such cases can be reduced. This stems from the fact that events need not be simulated in chronological order. For example, for a priority queue guaranteed output events may be computed by first simulating input events up to an appropriately higher time.

Sometimes our approach may result in reduced memory requirements. For example, in a tandem network of FCFS queues, in traditional simulation we need enough buffers to hold all the jobs present in the system at any time. In our approach, we may simulate the complete progress of one job, then simulate the complete progress of the next job, etc. Thus we would require memory to hold one job only.

3. An Example

We illustrate our simulation method by considering a simple system – a driver's license office, shown schematically in figure 1 below.

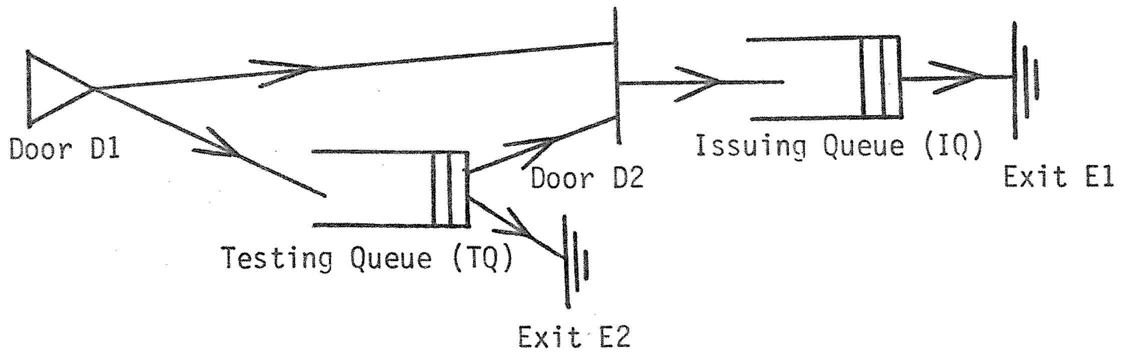

Figure 1: A Driver's License Office

Applicants for a license enter the office via door D1. There are two kinds of applicants – those who currently hold a license and simply want to renew it, and those who currently don't have a license and wish to get one. Applicants for a renewal enter a "license issuing area" via door D2. There is an issuing officer who takes their application, verifies the information therein, takes a photograph, accepts the license fee, and issues a license. Having received the license, the applicant leaves the driver's license office via exit E1.

Applicants without a license go to a "testing area" where a testing officer gives them a driving test. Some of them fail the test, and leave the testing area via exit E2. The successful applicants go to the license issuing area via door D2. From here on, they go through the same activities as described above for the license renewal applicants.

All the applicants arriving at the license issuing area form a waiting line (called the issuing queue or IQ). The issuing officer deals with one applicant at a time, in the FCFS order. Similarly, all applicants going to the testing area form a waiting line (called the testing queue or TQ), and the testing officer gives them the test in the FCFS order.

The problem is to simulate the events that occur in this system. An event is the arrival of an applicant at D2, IQ, TQ, E1 or E2.

In an actual simulation, one would normally compute the interarrival times at the door D1, and service times for the queues IQ and TQ by sampling from certain prespecified probability distributions. Similarly, one would determine the type of an applicant (whether applying for a renewal or otherwise) and whether an applicant taking the driving test fails, by sampling from prespecified probabilities. However, *for* *the ease of exposition*, we will assume a deterministic system with the following characteristics:

1. Applicants arrive at the door D1 at times 100, 200, 300, ... Service times at IQ and TQ are 105 and 500 respectively.

2. Applicants 10, 20, ... need to take the test; others are applying for a renewal.

3. For the applicants taking the test – the first one fails the test, next one passes, third one fails, and so on.

4. We assume that only 20 applicants enter the system.

We assume that the only information of interest about an applicant is his id. Hence the element e in a tuple [e,t] would refer to the applicant's id. We will use the following order in which the entities are reached by the simulation program to compute their guaranteed events. The phase of computing guaranteed events (i.e., phase A) consists of an alternating sequence of two subphases. In subphase 1, we follow the progress of applicants from the door D1 to exit E2 or door D2 (as the case may be), one applicant at a time. This subphase is over when an applicant reaches door D2. In subphase 2, we follow the progress of applicants from the door D2 to exit E1, one applicant at a time. This subphase is over (and subphase 1 starts) when no more applicants can progress from door D2. At the start of subphase 1, if no applicants can progress, then we enter phase B of the algorithm. In phase B, obviously we have to consider only the scheduled event for the door D2.

Below we show the sequence of actions taken by the simulation program. Specifically, we show the sequence in which tuples are computed for various buffers, and the computation of scheduled events in phase B. In the following, a quadruple (e,t,i,j) is used to state that tuple [e,t] is deposited in the buffer $B_{i,j}$. We refer to an entity i by its symbolic name D1, D2, E1, E2, IQ, or TQ, instead of an integer. The buffers are referred to in the similar way. Applicant ids are assumed to be A1, A2, ...

1. **Tuples buffered:**

   (A1,100,D1,D2), (A2,200,D1,D2), ..., (A9,900,D1,D2).

   This simulates the arrivals of applicants A1, ..., A9 at door D2 at times 100, ..., 900. During this simulation period, entity D2 can not compute its guaranteed next output events, since it has to output the tuples in buffer (D2,IQ) in the chronological order. If it deposits the tuple (A1,100), it doesn't know if there would be a tuple deposited on buffer (TQ,D2) later with a time component less than 100.

2. **Tuples buffered:**

   (A10,1000,D1,TQ), (A10,1500,TQ,E2).

   This simulates the arrivals of applicant A10 at TQ and E2 at times 1000 and 1500 respectively.

3. **Tuples buffered:**

   (A11,1100,D1,D2), (A12,1200,D1,D2), ..., (A19,1900,D1,D2).

This simulates the arrival of applicants 11, ..., 19 at door D2 at times 1100, ..., 1900. Note that entity D2 still cannot compute an output tuple.

4. **Tuples buffered:**

(A20,,2000,D1,TQ), (A20,2500,TQ,D2)

This simulates the arrivals of applicant A20 at TQ and D2 at times 2000 and 2500 respectively.

5. **Tuples buffered:**

(A1,100,D2,IQ), (A1,205,IQ,E1), (A2,200,D2,IQ), (A2,310,IQ,E1), ..., (A9,900,D2,IQ), (A9,1045,IQ,E1), (A11,1100,D2,IQ), (A11,1205,IQ,E1), (A12,1200,D2,IQ), (A12,1310,IQ,E1), ..., (A19,1900,D2,IQ), (A19,2045,IQ,E1).

This simulates the arrivals of applicants A1, ..., A9 and A11, ..., A19 at IQ and E1. Note that after computing the above tuples, D2 can not compute its next output tuple since it doesn't know the simulation time of next tuple to arrive in the buffer (D1,D2). At this point D1 also can not compute an output (since it has simulated all 20 applicants). Hence the algorithm enters its phase B. As mentioned before, we need compute only the next scheduled event for D2. This scheduled event is the tuple (A20,2500).

6. **Tuples buffered:**

(A20,2500,D2,IQ), (A20,2605,IQ,E1).

This simulates the arrivals of applicant A20 at IQ and E1. At this point the algorithm again enters phase B. Since there is no scheduled event for D2, simulation ends.

4. Further Improvements

We mentioned in section 2 that the simulation program cycles through various entities to compute their guaranteed events. It is possible that when a particular entity is reached, it has nothing to output. This wastes the time involved in reaching this entity and checking this condition for the entity. Depending on the particular system being simulated, one could possibly define an order in which to reach the entities such that the number of such cases is reduced. This order could be defined either statically or dynamically. For example, in a tandem network of FCFS queues, one may use the same order of entities in which a job arrives at them (statically defined order). In simulating a tree network rooted at a source process that generates jobs, one may follow the progress of one job from the source to a sink, then follow the next job, etc. (dynamically defined order). Now we define a heuristic to reduce the number of such cases in general. We keep a list of "potentially active" entities. Any entity currently not on this list is guaranteed not to be able to compute a guaranteed event. The simulation program reaches the entities by going through this list. When an entity has computed all its guaranteed events, it is removed from the list. When does an entity enter the list? One heuristic would be – whenever it receives an input. In a specific application, one could possibly define more appropriate boolean conditions for the specific entities. It would be helpful to keep a boolean variable for each entity to check whether it is on the list currently; it should be checked before evaluating the above boolean condition.

Consider an instant when phase A is over, i.e., no guaranteed events can be computed for any entity. Which entities should compute their next events? In general not all of them. For a particular system certain entities may be known not to compute the scheduled event with minimum time. A FCFS queue is one such example. We need not consider such entities in computing the scheduled events.

Earlier we suggested that in phase B we compute all the scheduled events afresh, i.e., the scheduled events computed in the current occurrence of phase B are not saved to be used in the next phase. Sometimes, several scheduled events computed in phase B remain valid even in the next occurrence of phase B. Here we suggest a heuristic to take advantage of this. One may keep an event list of the scheduled events computed during phase B. One would also keep a list of those entities whose scheduled events must be computed at the next occurrence of phase B. (If these entities have elements in the event list then they must be removed from the event list in the next occurrence of phase B.) This list is similar to the "potentially active" list mentioned above. As before, appropriate conditions may be defined to decide when an entity should be inserted in this list. Also, a boolean variable may be kept for each entity to check if it is currently on this list.

5. Discussion and Conclusions

We have presented a new approach to sequential simulation. In this approach we do not require that events are simulated in their chronological order. This is a major point of deviation from traditional simulation. This approach results in reduced number of insertions in the event list. It can sometimes reduce the number of computations of scheduled events that do not actually take place. Also, it can reduce memory requirements. These advantages would depend on the specific system being simulated. For better performance, one has to take decisions regarding the following issues: (i) In what order are the entities reached to compute their guaranteed events, (ii) When phase B begins, which entities should be reached to compute their scheduled events, and (iii) Should one keep an event list to hold scheduled events previously computed, so that some of them could be used in the next occurrence of phase B. We have suggested heuristics for these issues.

Our approach is based on a distributed simulation algorithm. In distributed simulation, usually each entity is simulated by an autonomous process and various processes are mapped onto processors. In order to achieve a high degree of parallelism, the processes are asynchronous and there is no global simulation clock. Processes synchronize with each other by sending and receiving the tuples [e,t]. In general, deadlocks may arise resulting from a cyclic waiting among the processes. One method of handling the deadlock problem is to let the simulator deadlock, to detect deadlock, and to recover from it [Chandy 81, Kumar 85c]. We applied this algorithm to the case of a uniprocessor system. In such a system, deadlock is easy to detect (when no entity can compute a guaranteed event, i.e., the list of "potentially active" entities is empty).

Several other algorithms have been proposed for distributed simulation [Chandy 79a, Peacock79, Jefferson82, Misra 84, Kumar 85c], but they involve too many overhead messages and we expect that this may severely degrade performance on a uniprocessor system.

We have shown in this paper that an algorithm developed for distributed simulation could also be useful for sequential simulation and it can suggest a new approach in this environment. We expect similar lessons to be learnt in other application areas of parallel computing.

Acknowledgements

My special thanks to Professor M. Gouda for his help in the presentation of this work, including several fruitful discussions and comments on earlier drafts of the paper. I am also thankful to Professors J. Misra and K. M. Chandy for several discussions, encouragement, and financial support for this work.

References

[Bagrodia 83] R. Bagrodia, "May: A Process Based Simulation Language", Master's Report, Dept. of Computer Sciences, University of Texas at Austin, Austin, Texas 78712, 1983.

[Birtwistle 73] G. M. Birtwistle, O. J. Dahl, B. Myhrhaug, and K. Nygaard, "Simula Begin", Auerbach Publishers Inc., Philadelphia, Pennsylvania, 1973.

[Birtwistle 79] G. M. Birtwistle, "DEMOS: A System For Discrete Event Simulation", Macmillan Press, 1979.

[Bryant 77] R. E. Bryant, "Simulation of Packet Communication Architecture Computer Systems", Technical Report MIT, LCS, TR-188, Massachusetts Institute of Technology, November 1977.

[Chandy 79a] K. M. Chandy, V. Holmes, and J. Misra, "Distributed Simulation Of Networks", *Computer Networks* 3(1):105-113, Feb. 1979.

[Chandy 79b] K. M. Chandy and J. Misra, "Distributed Simulation: A Case Study In Design And Verification of Distributed Programs", *IEEE Transactions on Software Engg.*, SE-5(5):440-452, September 1979.

[Chandy 81] K. M. Chandy and J. Misra, "Asynchronous Distributed Simulation Via a Sequence of Parallel Computations", *Communications of the ACM*, Vol. 24, No. 4, pp.198-205, April 1981.

[Chandy 82] K. M. Chandy and J. Misra, "A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems", *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Ottawa, Canada, August 1982.

[Chandy 83] K. M. Chandy, J. Misra, and L. Haas, "Distributed Deadlock Detection", *ACM Transactions on Computing Systems*, Vol. 1, No. 2, pp. 144-156, May 1983.

[Christopher 83] T. Christopher, et. al., "Structure of a Distributed Simulation System", in *Proceedings of the 3rd International Conference on Distributed Systems*, Ft. Lauderdale, Florida, 1983.

[Dahl 70] O. J. Dahl, B. Myhrhaug, and K. Nygaard, "Simula 67 Common Base Language", Norwegian Computing Centre, Oslo, Norway, 1970.

[Fishman 78] G. S. Fishman, "Principles of Discrete Event Simulation", A Wiley-Interscience Publication, John Wiley and Sons, New York, New York, 1978.

[Franta 77] W. R. Franta, "Process View of Simulation", Elsevier Computer Science Library, Operating and Programming Systems Series, P.J. Denning (ed.), Elsevier North Holland Publisher, 1977.

[Gligor 80] V. Gligor and S. Shattuck, "On Deadlock Detection in Distributed Data Bases", *IEEE-TSE*, Vol. SE-6, No. 5, September 1980.

[Holt 72] R. C. Holt, "Some Deadlock Properties of Computer Systems", *Computing Surveys*, Vol. 4, No. 3, pp. 179-196, September 1972.

[Jefferson 82] D. R. Jefferson and H. A. Sowizral, "Fast Concurrent Simulation Using The Time Warp Mechanism, Part I: Local Control", Technical Report, The Rand Corporation, Santa Monica, California, December 1982.

[Kleinrock 76] L. Kleinrock, "Queueing Systems, Volume II: Computer Applications", Wiley-Interscience, John Wiley & Sons, Inc., 605 Third Avenue, New York, N.Y. 10158, 1976.

[Kobayashi 81] H. Kobayashi, "Modeling and Analysis: An Introduction to Systems Performance Evaluation Metodology", The Systems Programming Series, Addison-Wesley Publishing Company, Menlo Park, California, Oct. 1981.

[Kumar 85a] D. Kumar, "A Class of Termination Detection Algorithms For Distributed Computations", Technical Report, Department of Computer Sciences, University of Texas at Austin, Austin, Texas 78712, May 1985.

[Kumar 85b] D. Kumar, "A High Speed Distributed Simulation Scheme And Its Performance Evaluation", in preparation.

[Kumar 85c] D. Kumar, "Distributed Simulation", Ph.D. Thesis (in preparation), Department of Computer Sciences, University of Texas, Austin, Texas 78712.

[Lamport 78] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, Vol. 21, No. 7, July 1978.

[Lonow 82] G. Lonow and B. Unger, "Process View of Simulation In ADA", in *1982 Winter Simulation Conference*, pages 77-86, 1982.

[Misra 83] J. Misra, "Detecting Termination of Distributed Computations Using Markers", *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Montreal Canada, August 17-19, 1983.

[Misra 84] J. Misra, "Distributed Simulation", *IEEE Tutorial on Distributed Simulation*, 1984.

[Obermarck 82] R. Obermarck, "Distributed Deadlock Detection Algorithm", *ACM Transactions on Database Systems*, Vol. 7, No. 2, pp.187-208, June 1982.

[Peacock 79] J. K. Peacock, J. W. Wong, and E. G. Manning, "Distributed Simulation Using A Network of Processors", *Computer Networks* 3(1):44-56, Feb. 1979.

[Sauer 78] C. H. Sauer, "Characterization And Simulation of Generalized

Queuing Networks", Research Report RC-6057, IBM Research, Yorktown Heights, NY, May 1978.

[Sauer 81] C. H. Sauer, and K. M. Chandy, "Computer Systems Performance Modeling", Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1981.

[Sauer 83] C. H. Sauer and E. A. MacNair, "Simulation of Computer Communication Systems", Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1983.

[Seethalakshmi 79] M. Seethalakshmi, "A Study And Analysis of Performance of Distributed Simulation", Master's Report, Dept. of Computer Sciences, University of Texas, Austin, Texas 78712, May 1979.