Distributed Discrete-Event Simulation

JAYADEV MISRA

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

Traditional discrete-event simulations employ an inherently sequential algorithm. In practice, simulations of large systems are limited by this sequentiality, because only a modest number of events can be simulated. Distributed discrete-event simulation (carried out on a network of processors with asynchronous message-communicating capabilities) is proposed as an alternative; it may provide better performance by partitioning the simulation among the component processors. The basic distributed simulation scheme, which uses time encoding, is described. Its major shortcoming is a possibility of deadlock. Several techniques for deadlock avoidance and deadlock detection are suggested. The focus of this work is on the theory of distributed discrete-event simulation.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.4.1 [**Operating Systems**]: Process Management—*deadlocks*; I.6.1 [**Simulation and Modeling**]: Simulation Theory

General Terms: Algorithms, Theory, Verification

Additional Key Words and Phrases: Asynchronous simulation, deadlock detection and recovery, deadlock prevention, message communicating processes, modeling interaction by message communication

INTRODUCTION

This survey presents an entirely new approach to the problem of system simulation. A system simulation is typically carried out as a repetition of the following sequential steps: Fetch one event from a data structure, carry out one step of simulation, and (possibly) update the data structure. Such simulations are practical only when the number of events being simulated is modest.

Recent advances in computer and communication systems have resulted in demands for new tools for their analyses. Mathematical modeling techniques have so far proved inadequate in dealing with these systems, and simulation seems to be the only viable alternative. Unfortunately, simulation is proving to be inadequate because of the sheer magnitude of the problem. For instance, a telephone switch generates about 100 internal messages in completing a local call. Large telephone switches can handle 100 or more calls per second. Thus, simulation of a telephone switch for 15 minutes of real time requires the simulation of nearly 10 million messages, which will require several hours on a very fast uniprocessor.

One alternative is to exploit the cost benefits of cheap micro/minicomputers and high-bandwidth lines by partitioning the simulation problem and executing the parts in parallel. Unfortunately, however, the typical simulation algorithm does not easily

CONTENTS

INTRODUCTION
1. AN OVERVIEW OF SYSTEM SIMULATION
   1.1 System Simulation Problem
   1.2 Distributed Simulation
   1.3 History
2. SEQUENTIAL SIMULATIONS OF SYSTEMS
   2.1 Physical Systems
   2.2 What Is Simulation?
   2.3 The Sequential Simulation Algorithm
3. DISTRIBUTED SIMULATION: THE BASIC SCHEME
   3.1 A Model of Asynchronous Distributed Computation
   3.2 Basic Scheme for Distributed Simulation
   3.3 Partial Correctness of the Basic Distributed Simulation Scheme
   3.4 Features of the Basic Distributed Simulation Scheme
4. DISTRIBUTED SIMULATION: DEADLOCK RESOLUTION
   4.1 Overview of Deadlock Resolution
   4.2 Deadlock Resolution Using *Null* Messages
   4.3 Correctness of the Simulation Algorithm
   4.4 Discussion
   4.5 Demand-Driven *Null* Message Transmission
   4.6 Circulating Marker for Deadlock Detection and Recovery
5. SUMMARY AND CONCLUSION
ACKNOWLEDGMENTS
REFERENCES

partition for parallel execution. An entirely new approach to simulation for multiprocessors is required. This survey presents such an approach.

The text is organized in five sections. Section 1 describes the need for distributed simulation; it gives a quick survey of the system simulation problem, the sequential simulation algorithm and its shortcomings. The scope of the paper and a history of distributed simulation are also included in that section. In order to make the paper self-contained, basic notions of sequential simulation are introduced and explained in Section 2. A proof that the sequential simulation algorithm works correctly is given in that section; surprisingly, the author could not find such a proof in any simulation book. It is then shown why this scheme cannot be readily parallelized. Section 3 introduces the basic distributed simulation scheme, which is shown to be partially correct. It is shown that this scheme may result in deadlock. Several different approaches for deadlock resolution are discussed in Section 4. Section 5 contains a summary and possible directions for future investigation.

We believe that distributed simulation offers a promising approach to speeding up simulation. The basic theory has been developed; it remains to experiment with various alternative heuristics to ensure that substantial performance gains over sequential simulation can be achieved. The problem of deadlock and its resolution are at the core of the performance issue. There is some indication that reasonable performance gains may be expected at least for simulations of certain classes of queuing networks [Peacock et al. 1979a, 1979b; Quinlivan 1981]. However, several large-scale studies, with a number of different heuristics for deadlock resolution, are needed before any claims about performance can be made. We hope that this paper will spur interest in such studies.

This paper does not introduce a new simulation language, because distributed simulations can be written using sequential simulation languages for simulating the physical processes, and message communication languages for describing interactions among component machines. We also avoid a number of traditional issues in simulation: pseudorandom number generation, statistical analysis of the outputs, etc. Methods developed in these areas for sequential simulation still apply [Fishman 1978]. Our goal in this paper is to show how the body of actual simulation can be distributed among a set of interacting machines.

1. AN OVERVIEW OF SYSTEM SIMULATION

1.1 System Simulation Problem

We consider the problem of simulating physical systems, also called *networks*, that consist of one or more *physical processes*. Each physical process operates autonomously, except to interact with other physical processes in the system. The interaction is by *messages*. Contents of a

message sent by a (physical) process depend on the characteristics of the process (its initial state, its rules of operation) and the messages that the process has received so far.

We describe the problem and the terminology more precisely in the next section. We note that many real systems can be modeled in terms of processes and messages as described above. For example, in a computer system, CPU, disks, memory, and job entry terminals may be thought of as processes; the CPU may interact with a disk by sending it messages requesting or releasing disk space; a job entry terminal may interact with the CPU by sending it messages, which are in fact jobs or tasks to be executed. Detailed examples are given in the next section.

Typical steps in constructing and using a simulation program consists of

(1) starting with a real system and understanding its characteristics,
(2) building a model from the real system in which aspects relevant to simulation are retained and irrelevant aspects are discarded,
(3) constructing a simulation of the model that can be executed on a computer (simulations other than computer programs are not considered here), and
(4) analyzing simulation outputs to understand and predict the behavior of the real system.

In addition, the model and the simulation must be verified and may be refined during steps (2) and (3), perhaps iteratively, if they do not meet the expectations. In this paper, we look at only one step—step (3)—of the entire simulation process. What is typically called a model in step (2) is actually our physical system; we show how to go from a physical system to a computer program for simulation that is distributed and hence may be concurrently executed on several machines. We do not consider the problem of constructing a physical system description from the real system, nor do we consider how to analyze simulation outputs to predict the behavior of the real system. Stated another way, we show how to construct an asynchronous system (the simulator running on asynchronous machines) from a synchronous system (the physical system, running in real time). We further restrict ourselves to discrete-event simulations; we assume that *events* in the physical system—in our case, message transmissions—happen at discrete points in time.

1.1.1 Traditional Approach to System Simulation

Traditionally, discrete-event system simulations have been done in a sequential manner. A variable *clock* holds the time up to which the physical system has been simulated. A data structure, called the *event list*, maintains a set of messages, with their associated times of transmissions, that are scheduled for the future. Each of these messages is guaranteed to be sent at the associated time in the physical system, provided the sender receives no message before this message transmission time. At each step, the message with the smallest associated future time is removed from the event list, and the transmission of the corresponding message in the physical system is simulated. Sending this message may, in turn, cause other messages to be sent in the future (which then are added to the event list) or cause previously scheduled messages to be canceled (which are removed from the event list). The clock is advanced to the time of the message transmission that was just simulated.

This form of simulation is called *event driven*, because events (i.e., message transmissions) in the physical system are simulated chronologically and the simulation clock is advanced after simulation of an event to the time of the next event. There is another important simulation scheme, time-driven simulation, in which the clock advances by one tick in every step and all events scheduled at that time are simulated. We do not discuss time-driven simulation in this paper.

1.1.2 Drawbacks of Sequential Simulation

The nature of the event-list mechanism dictates a sequential simulation, since in each cycle of simulation only one item is removed from the event list, its effects

simulated, and the event list, possibly, updated. This is unfortunate; the algorithm cannot be readily adapted for concurrent execution on a number of processors, since the event list cannot be effectively partitioned for such executions. We contend that the sequentiality inherent in the event-list structure is a major impediment to the widespread use of simulation. Complex computer and communication systems of the future will be intractable mathematically and therefore will have to resort to simulation for their performance evaluations. Current simulation techniques will prove inadequate for these systems because, with current technology, only a modest number of events can be simulated. It is necessary to take a radically new approach to simulation that will utilize the power and cost benefits of small computers and high-bandwidth communication lines.

1.2 Distributed Simulation

Distributed simulation offers a radically different approach to simulation. Shared data objects of sequential simulation—the clock and event list—are discarded. In fact, there are no shared variables in this algorithm. We suggest an algorithm in which one machine may simulate a single physical process; messages in the physical system are simulated by message transmissions among the machines. The synchronous nature of the physical system is captured by encoding time as part of each message transmitted between machines. We show that machines may operate concurrently as long as their physical counterparts operate autonomously; they must wait for message receptions to simulate interactions of the corresponding physical processes.

Distributed simulation offers many other advantages in addition to the possible speedup of the entire simulation process. It requires little additional memory compared with sequential simulation. There is little global control exercised by any machine. Simulation of a system can be adapted to the structure of the available hardware; for instance, if only a few machines are available for simulation, several physical processes may be simulated (sequentially) on one machine.

Several distributed simulation algorithms have appeared in the literature. They all employ the same basic mechanism of encoding physical time as part of each message. The basic scheme they use may cause deadlock. Various distributed simulation algorithms differ in the way they resolve the deadlock issue.

1.3 History

Sequential simulation has a long history; Franta provides a discussion of a number of prominent simulation languages and their relative merits [Franta 1977]. Among the many simulation packages introduced recently, we mention DEMOS, SAMOA, and MAY [Birtwistle 1979; Lonow and Unger 1982; Bagrodia et al. N.d.]. DEMOS is a discrete-event modeling package implemented in SIMULA [Dahl et al. 1970]. It provides an extensive list of features for event scheduling, data collection, and report generation. SAMOA uses Ada as the base language [U.S. DoD 1982]. MAY provides a very small set of constructs for message communication; these features have been used to build an extensive library for simulations of computer and communication networks. The minimality of MAY makes it possible for it to be implemented even on personal computers.

The idea of distributed simulation was proposed by Chandy in 1977 in a series of lectures at the University of Waterloo, and independently by R. E. Bryant. Papers by Chandy and Misra [1979], Chandy et al. [1979], and Bryant [1977] contain the basic ideas of distributed simulation, the problem of deadlock, and schemes for deadlock resolution. Peacock et al. [1979a, 1979b] and Holmes [1978] have proposed mechanisms for avoiding deadlock by periodic use of *probe messages*. Empirical work by Peacock et al. has shown that their method is indeed viable: The time needed for simulation of a class of queuing networks steadily decreases when the number of processors available for simulation increases. Empirical investigations by Seethalakshmi and Quinlivan showed that the method is also efficient for acyclic physical systems and that performance can be substantially improved if there is adequate space for

buffering messages [Seethalakshmi 1979; Quinlivan 1981].

Chandy and Misra have subsequently suggested a scheme for deadlock detection and recovery [Chandy and Misra 1981]. Reynolds suggested using common memory among neighbors to avoid deadlock [Reynolds 1982]. A notable departure from these schemes is one proposed by Jefferson based on *virtual time* [Jefferson 1985]. A performance analysis of this scheme appears in Lavenberg et al. [1983]. The virtual time approach is still being developed and it is a little premature to include it in this survey.

Bezivin and Imbert propose an approach in which each process in the simulator maintains a local time, and an overall global time is maintained by a central process [Bezivin and Imbert 1983]. Christopher et al. propose precomputing minimum wait time along all paths in a network so that delay information may be propagated rapidly among nonneighboring processes [Christopher et al. 1983].

Kumar has combined some recent work in deadlock and termination detection with the basic simulation scheme [Kumar 1986; Misra 1983]. Behaviors of these algorithms on a wide class of practical simulation problems are currently being investigated, both analytically and using empirical techniques.

2. SEQUENTIAL SIMULATIONS OF SYSTEMS

This section introduces the problem of system simulation. A precise definition of simulation is given. The sequential simulation algorithm using the event-list structure is presented and proved. It is shown why the sequential simulation scheme cannot be readily adapted for parallel execution.

2.1 Physical Systems

We consider physical systems, also called *networks*, consisting of a finite number of *physical processes* (abbreviated as pp's). Each pp represents some component of the real system to be simulated. For instance, in a computer system, the CPU, each disk, each memory bank, and each job entry terminal may be thought of as a pp. In traditional simulation terminology, each pp is described by a set of *events* and each event has an associated time of occurrence. Furthermore, there is a *dependency* relation among all events in the system; if the pair of events (e, e') is part of the dependency relation, we say that e' depends on e. Dependency relation captures our intuitive understanding of the order in which events must occur in the system; no event can occur unless all the events on which it depends have already occurred. Clearly, we must then require that the dependency relation not be cyclic, that is, it should be an irreflexive partial order; furthermore, the time associated with an event e' must be no less than the associated time of any event e on which it depends.

We next give an example that clarifies the notion of events and dependencies.

Example 2.1 (Car Wash)

The following example is a variation of one appearing in Birtwistle et al. [1973]. A car wash system consists of an attendant and two car washes, abbreviated CW1 and CW2. Cars arrive at random times at the attendant. The attendant directs cars to CW1 or CW2 according to the following rule: If both car washes are busy, that is, washing cars, any arriving car is queued at the attendant; if exactly one car wash is idle, the car at the head of the queue, if any, is sent to that idle car wash; if both car washes are idle, the car at the head of the queue, if any, is sent to CW1. CW1 spends 8 minutes and CW2, 10 minutes in washing a car. Given some distribution of car arrivals, it is necessary to compute the average amount of time a car spends at the car wash (including the washing time) and the average length of the queue that builds up at the attendant. We do not compute the above statistics; we simply show the sequence of events and message transmissions in two different views of the car wash problem.

The entire system can be described by listing all possible events—all possible car arrivals and their subsequent washings—and dependencies between them. We restrict ourselves to describing part of this system.

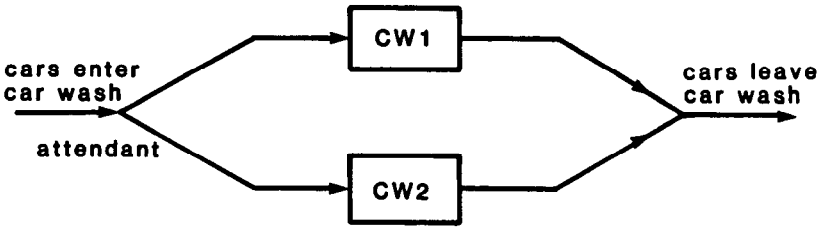

Figure 1. Schematics of car flow.

The schematic diagram of the flow of cars is given in Figure 1.

Initially both CW1 and CW2 are idle. Assume that 6 cars, C1 through C6, arrive at the attendant at times 3, 8, 9, 14, 16, 22. An event in this system is either a car arriving at some point, that is, at the attendant, CW1, or CW2, or a car leaving the car wash. We assume that the driving time from the attendant to CW1 or CW2 is zero. Also, the washing of a car begins as soon as it arrives at CW1 or CW2. The chronological sequence of events is given in Table 1.

Dependencies among events is shown in the directed graph of Figure 2; a directed line from event e_1 to event e_2 denotes that event e_2 depends directly on event e_1.

If an event e' depends on event e, then simulation of e must precede simulation of e'. Conversely, if e, e' are *independent*, that is, there is no dependency relation between them, then they may be simulated concurrently or, equivalently, in arbitrary order. Thus, two independent events, such as event 8 (C4 arrives at the attendant) and event 12 (C3 leaves car wash) are independent and hence can be simulated concurrently.

We find it convenient to dispense with the notion of event; we model a physical system as a set of pp's that operate autonomously to change their own states and that interact by *sending and receiving messages*. Such a model is possible because if event e' at process q depends on event e at process p, then process p may send a message to process q after it completes execution corresponding to event e, and q, upon receiving this message (and other messages corresponding to other dependencies of e') may carry out the actions necessary for implementation of e'. Message transmission delays are zero, that is, any message sent at time t is received by the intended recipient at t. (Recall that we are describing a physical system, not the computer system on which the simulation is to run.) If it is necessary to model delays in the real-world system (viz., driving time from attendant to a car wash in the last example), then either the sender of a message idles for some time before sending the message or the recipient of a message idles for some time after receiving the message; another possibility is to model the communication medium as a process incorporating the delay.

Table 1. A Sequence of Events in the Car Wash

Event number	Time	Event
1	3	C1 arrives at the attendant
2	3	C1 arrives at CW1
3	8	C2 arrives at the attendant
4	8	C2 arrives at CW2
5	9	C3 arrives at the attendant
6	11	C1 leaves car wash
7	11	C3 arrives at CW1
8	14	C4 arrives at the attendant
9	16	C5 arrives at the attendant
10	18	C2 leaves car wash
11	18	C4 arrives at CW2
12	19	C3 leaves car wash
13	19	C5 arrives at CW1
14	22	C6 arrives at the attendant
15	27	C5 leaves car wash
16	27	C6 arrives at CW1
17	28	C4 leaves car wash
18	35	C6 leaves car wash

Example 2.1 (continued)

We now present the car wash viewed as a message-passing system. The car wash system has 5 pp's: the source, which generates

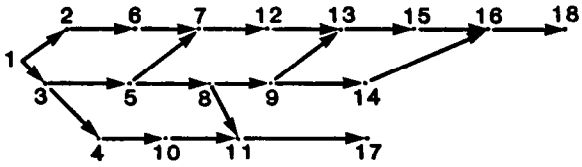

Figure 2. Schematics of events in a car wash.

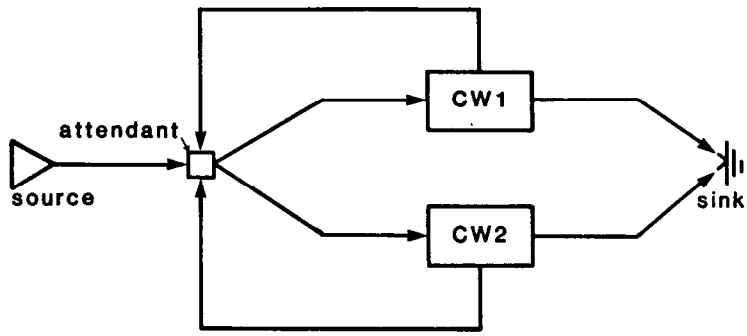

Figure 3. Schematics of message flow in the car wash system.

cars at the prescribed times, the attendant, CW1, CW2, and the sink (exit). The schematic diagram of message communications among these pp's is given in Figure 3.

Note that we have possible message flow paths from CW1 and CW2 to the attendant. This is because the attendant must know when a car wash becomes idle. (In this particular problem, the attendant can keep track of the times at which the last cars were sent to CW1 and CW2 and, since the washing times are fixed, can deduce the times at which CW1 and CW2 will next become idle. This means that the attendant is simulating CW1 and CW2. In general it will not be possible, or preferable, to do so in a simulation.) A complete list of messages for this example is shown in Table 2, with corresponding event numbers from Table 1. Each message has a sender, a receiver, and message content. In our case the content is either a car number or the status (idle) of a car wash.

This example shows how to model event interactions by message transmissions. In particular, if an event at one pp causes events to happen at several other pp's, we shall have to model such event dependencies by several message transmissions. Second, the chronological order of simulations of events in sequential simulation (described later) guarantees that every event simulation precedes the simulation of events that depend upon it. Our approach in distributed simulation dispenses with chronological simulations of events.

There are two conditions that are met by every physical system imaginable: *realizability* and *predictability*. We assume that both these conditions hold for all physical systems we consider.

Realizability. A message sent by a pp at time t is a function of its initial state, t, and the messages it has received up to and including t.

Realizability says merely that a pp cannot guess any message it will receive in the future. Note that we admit the possibility of a message that is received at t affecting a message that is sent at t. An example of a pp in which this instantaneous cause–effect is seen is given below.

Example 2.2 (Instantaneous Message Transmission)

Consider a pp that acts as a merge point for several pp's. Schematically, such a pp, A, is shown in Figure 4. Messages arriving at A, either from the top or from the

Table 2. A Sequence of Message Transmissions in the Car Wash System

Message number	Event number	Time message sent	Message sender	Message receiver	Content
1	—	0	CW1	Attendant	Idle
2	—	0	CW2	Attendant	Idle
3	1	3	Source	Attendant	C1
4	2	3	Attendant	CW1	C1
5	3	8	Source	Attendant	C2
6	4	8	Attendant	CW2	C2
7	5	9	Source	Attendant	C3
8	6	11	CW1	Sink	C1
9	—	11	CW1	Attendant	Idle
10	7	11	Attendant	CW1	C3
11	8	14	Source	Attendant	C4
12	9	16	Source	Attendant	C5
13	10	18	CW2	Sink	C2
14	—	18	CW2	Attendant	Idle
15	11	18	Attendant	CW2	C4
16	12	19	CW1	Sink	C3
17	—	19	CW1	Attendant	Idle
18	13	19	Attendant	CW1	C5
19	14	22	Source	Attendant	C6
20	15	27	CW1	Sink	C5
21	—	27	CW1	Attendant	Idle
22	16	27	Attendant	CW1	C6
23	17	28	CW2	Sink	C4
24	—	28	CW2	Attendant	Idle
25	18	35	CW1	Sink	C6
26	—	35	CW1	Attendant	Idle

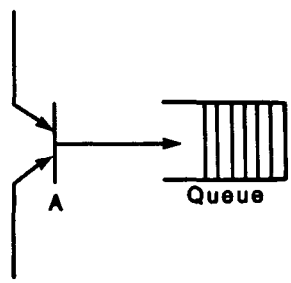

Figure 4. A merge point pp.

bottom, are instantaneously sent to the queue on the right. Therefore, a message sent by A at t depends upon messages received at t. It may be argued that pp A cannot be physically constructed. However, this pp may represent a real-world entity, where the interval between reception and transmission of a message is small enough to be ignored altogether in the modeling process. Such merge points are often used in queuing network descriptions of systems.

Predictability. Suppose the physical system has cycles, that is, a set of processes $pp_0, \ldots, pp_{n-1}$, where pp_i sends messages to pp_{i+1} (and perhaps to other pp's) and receives messages from pp_{i-1} (and perhaps other pp's).[1] Suppose that the message, if any, sent by pp_i at some time t depends on what pp_i receives at t, for all i; then we have a circular definition where the message received by every pp at t is a function of itself. In order to avoid such situations, we require that *for every cycle and t there is a pp in the cycle and a real number* $\epsilon, \epsilon > 0$, *such that the messages sent by the pp along the cycle can be determined up to* $t + \epsilon$, *given the set of messages that the pp receives up to and including t.*

Predictability guarantees that the system is "well defined" in the sense that the output of every pp up to any time t can be computed given the initial state of the system.

[1] All arithmetic in pp subscripts is modulo n.

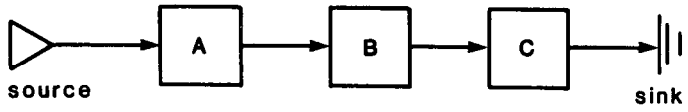

Figure 5. Schematic diagram of the example assembly line.

Table 3. Job Generation Times and Service Times in the Example of Figure 5

	Jobs			
Work station	1	2	3	4
Job generation times				
Source	5	7	30	32
Service times				
A	4	10	1	5
B	12	15	2	7
C	2	3	1	4

We next consider some typical simulation examples and show that they satisfy the realizability and predictability conditions.

Example 2.3 (Car Wash–Realizability and Predictability)

We consider the car wash problem introduced in Example 2.1. Each pp's output at time t depends only upon the messages it has received up to t. Of particular interest is the behavior of the attendant. If it receives an "idle" message from either of the car washes at time t and the queue is not empty at t, then it sends a message at t. Therefore, the realizability condition is satisfied. The predictability condition is satisfied because each cycle contains one of CW1 or CW2 and, given the input to CW1 (CW2) up to t, we can predict the output from it up to $t + 8$ ($t + 10$).

Example 2.4 (Assembly Line)

An assembly line consists of a series of n work stations. Jobs enter the assembly line at work station 1; when a job has been serviced at work station i, it proceeds to work station $(i + 1)$, $i = 1, 2, \ldots, n - 1$; a job leaves the system after being serviced at work station n. Service times at different work stations are random variables; jobs may be queued at a station awaiting service. A work station takes one job from its input queue when it is free, services that job, and then sends it to the queue of the following work station. All work stations service the jobs in a first come, first served (FCFS) basis. It is desired to find the expected number of jobs in the queue of each work station and the expected waiting time for jobs at each work station.

Specifically, consider an assembly line consisting of three work stations, A, B, and C, which services four jobs identified as 1, 2, 3, and 4. Schematically, the assembly line is shown in Figure 5.

The times at which the source generates jobs and the service time of each work station for each job are given in Table 3.

The source (call it work station 0), the sink (call it work station 4), and each work station are pp's. pp_i sends messages to pp_{i+1}, $i = 0, 1, 2, 3$. The source sends messages (which represent jobs) to work station 1 at times 5, 7, 30, and 32. If a job $j, j > 1$, arrives at a work station at time t, then its service at this work station begins either immediately (at t) if the work station is then idle, or it begins immediately after the departure of the $(j - 1)$th job from the work station. Let A_j be the time of arrival of job j at some work station, let D_j be the time of departure of job j from this work station, and let S_j be the service time for job j at this work station. Then we have

$$D_0 = 0;$$

$$D_j = \max(A_j, D_{j-1}) + S_j, \qquad j = 1, 2, \ldots.$$

Using the service times and generation times of jobs given in the previous table, we can construct the departure times from work stations, that is, times at which messages are sent, as in Table 4.

Each work station's output at time t depends only on the jobs it has received up to t, and therefore the realizability condition is satisfied. The predictability condition is trivially satisfied since there is no cycle in the physical system.

Table 4. Times at Which pp's Send Messages in the Example of Figure 5

	Message			
pp	1	2	3	4
Source	5	7	30	32
A	9	19	31	37
B	21	36	38	45
C	23	39	40	49

Example 2.5 (A Computer System)

Imagine a computer installation that consists of a central processing unit (CPU) and two peripheral processors, proc1 and proc2. Jobs enter the CPU, spend some time there, and then branch to one of the peripheral processors with some given probability. Upon completion of processing at the peripheral processor, a job may leave the system or return to the CPU with some probability. The schematic diagram of the system is shown Figure 6.

This system has pp's for the source, the sink, merge points M_1 and M_2, branch points B_1 and B_2, the CPU, proc1, and proc2. Each message represents the transfer of a job from one pp to another. The realizability property holds because no pp bases its behavior on anticipation of the future. Probabilistic decisions at B_1, B_2 cause no difficulty because the inputs to B_1, B_2 up to time t determine their outputs up to time t (though the outputs may be different at different times owing to the probabilistic nature). We can realistically assume that each processor spends nonzero time in processing a job. Therefore the system also has the predictability property.

This concludes our discussion of using physical systems to model real-world systems. From now on we assume that we are dealing with physical systems with the properties of realizability and predictability. Now we define the meaning of simulation for such physical systems.

2.2 What Is Simulation?

We wish to build a simulator, or a *logical system* consisting of *logical processes* (abbreviated lp), to simulate a physical system. We use "simulation" in a rather strict sense: We say that a logical system correctly simulates a physical system if it is *possible* for the logical system to predict the exact sequence of message transmissions in the physical system. That is, if $t_1, t_2, \ldots, t_i, \ldots$ are the times at which the messages $m_1, m_2, \ldots, m_i \ldots$ are transmitted in the physical system and $t_1 \leq t_2 \cdots \leq t_i \leq \cdots$, then the logical system should be able to output the sequence $\langle (t_1, m_i), (t_2, m_2), \ldots, (t_i, m_i), \ldots \rangle$.

The logical system may not actually print the sequence $\langle \cdots (t_i, m_i) \cdots \rangle$. All that is desired is that it should be *possible* to do so from the logical system.

Clearly a physical system is a simulation of itself. We wish to construct logical systems that may *not* operate at the same speed as the physical system. Our goal is to construct a logical system out of a machine or machines where the speeds of processors and communication links (if any) are arbitrary. In other words, we wish to duplicate the behavior of a synchronous physical system using asynchronous logical components.

It should be observed that we can carry out the typical functions of simulation—analyze data, predict performance or future behavior, generate reports, etc.—using the logical system. We do not address these issues in this paper; we merely observe that since it is possible to create the sequence of physical message transmissions in the logical system, all interactions can be reconstructed and analyzed.

Example 2.6 (Message Transmission in the Assembly Line Example)

A simulation of the assembly line of Example 2.4 should be able to predict the following message sequence. This sequence is derived from Table 4. In the following, a message consists of (sender id, receiver id, message content). We write a 4-tuple (t, s, r, m) to denote that at time t, pp s sends a message to pp r with content m.

⟨(5, source, *A*, 1), (7, source, *A*, 2),
(9, *A*, *B*, 1) (19, *A*, *B*, 2), (21 *B*, *C*, 1),
(23, *C*, sink, 1), (30, source, *A*, 3),
(31, *A*, *B*, 3), (32, source, *A*, 4),
(36, *B*, *C*, 2), (37, *A*, *B*, 4), (38, *B*, *C*, 3),
(39, *C*, sink, 2), (40, *C*, sink, 3),
(45, *B*, *C*, 4), (49, *C*, sink, 4))

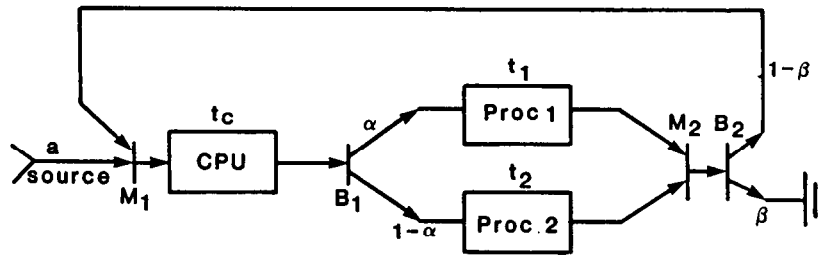

Figure 6. Schematic diagram of job flow in a computer system that has a CPU and two peripheral processors: a, mean time between arrival of jobs from the outside source, a random variable; t_c, mean time spent by a job at the CPU, a random variable; t_1, mean time spent by a job at the peripheral processor 1 (proc1), a random variable; t_2, mean time spent by a job at the peripheral processor 2 (proc2), a random variable; α, probability of a job going to proc1; β, probability of a job exiting the system; M_1, M_2, merge points; B_1, B_2, branch points.

2.3 The Sequential Simulation Algorithm

Two major data objects used by the sequential simulation algorithm are the *clock* and *event list*, which are described as follows:

Clock. A real-valued variable. It gives the time up to which the corresponding physical system has been simulated; that is, all messages (t, m) sent in the physical system with $t <$ *clock*, can be deduced from the logical system at any point in its execution.

Event list. A set of tuples of the form (t_i, m_i), where $t_i \geq$ *clock* and m_i is a message. (We assume that the identities of the sender and the receiver are parts of the message.) A tuple (t_i, m_i) is in the event list means that, in the physical system, *if the sender of m_i receives no message at any t, clock $\leq t < t_i$, then it sends m_i at t_i and sends no other message at any time t, clock $\leq t \leq t_i$.*

It is required that for every pp_i there be at least one event-list entry (t_i, m_i) in which pp_i is the sender. If a pp sends no message in the future, unless it receives further messages, the corresponding event-list entry will be (∞, m), where the message content in m is arbitrary. A similar entry, (∞, m), will always be in the event list for a pp that has terminated.

Example 2.7 (A Snapshot in Sequential Simulation of the Assembly Line)

In simulating the assembly line of Example 2.4, a possible value of *clock* and corresponding entries in the *event list* are shown as follows:

clock: 9
event list {(19, *A*, *B*, 2), (21, *B*, *C*, 1),
(∞, *C*, sink, —),
(30, source, *A*, 3)}

This snapshot of the simulation corresponds to the point in the physical system where the source has produced jobs 1 and 2, and job 1 has been processed at *A* and sent to *B*. The source has one more job scheduled for production; *A* has scheduled to send job 2 to *B* at time 19, provided *A* receives no more jobs between 9 and 19; *B* has scheduled to send job 1 to *C* at time 21, provided it receives no more jobs before then; *C* has scheduled no message because it has received no jobs.

It should be noted that each entry (t, m) in the event list is *conditional.* An entry (t, m) may not actually occur in the physical system, because this message transmission *may be* canceled if the sender of m receives a message prior to t. In fact, it is impossible to construct general-purpose sequential simulations without canceling events from the event list. For example, cancellation is required in the simulation of a system with preemption: Scheduled departure of a job from a server for some future time may have to be canceled (and recomputed) owing to the arrival at the server of a job that preempts the previous job.

Let (t, m) be an entry in the event list where t is smaller than t' for every other (t', m') in the event list. *We can then guarantee that the first message to be sent at or after the current value of clock is message m and that it is sent at time t.* This is the content of the following theorem, upon which sequential simulation is based.

Theorem 1

Let (t, m) be an entry in the event list such that $t < t'$ for every other entry (t', m') in the event list. Then the message m is transmitted at time t in the physical system and no other message is transmitted at t'', where $clock \leq t'' < t$.

Proof. If message m is not transmitted at t, it must be because some other message is transmitted at or before t (and at or after *clock*), which causes the sender of m to cancel transmission of m. Consider the first message m' to be so transmitted; it must be transmitted at t' where $clock \leq t' \leq t$. The sender of m' could not have received any message between clock and t', because such a message would be the first message transmitted after clock. Then (t', m') must be an entry in the event list, because the sender of m' sends its message without receiving any other message after the current clock value and before t'. Since $t' \leq t$, it contradicts our choice of (t, m). Hence the result. □

2.3.1 Simulations of Simultaneous Events

We assumed in Theorem 1 that there is a unique tuple (t, m) in the event list, where t is smaller than t', for all other (t', m'). In a sequential simulation, two message transmissions that happen simultaneously in the physical system, that is, at the same time t, must be simulated in some order. Simulating them in arbitrary order can lead to problems, as in the following: pp A plans to send a message m to pp B at time t; pp B is an alarm clock that is scheduled to go off, that is, to send a message m' to pp A at time t, unless it receives a message from pp A *before or at* t. In the physical system, pp B will *not* send m' to pp A. However, if these message transmissions are simulated sequentially in arbitrary order, a possible simulation may result in pp B sending m' to pp A. This example illustrates that events should be simulated in the order of their dependencies (m' is dependent on m in this example). Simulation in the order of dependencies also guarantees chronological order. Certain sequential simulation languages, such as GPSS [Franta 1977], provide the user with facilities for defining orderings among simultaneous events. In this case, information defining orderings must be kept with tuples in the event list. Distributed simulation is based on the dependency order and hence avoids this problem.

A tuple (t, m) in the event list is a *smallest tuple* if $t \leq t'$ for every (t', m') in the event list, and, if $t = t'$, then message m' does not precede m {this has to be deduced from additional facts stored with m' and m}. Note that there may be several smallest tuples and they may be simulated in arbitrary order.

The simulation algorithm, given below, works as follows. In each step a smallest tuple is removed from the event list, its effects are simulated (causing possible additions to and deletions from the event list), and the clock is advanced to the time associated with this message transmission. This algorithm is given in a pseudoprogramming notation below.

2.3.2 The Sequential Simulation Algorithm (See Figure 7)

The correctness of this algorithm should be obvious from our previous discussions. Note that the sequential simulation algorithm is capable of producing the sequence of message transmissions in the physical system; it simply prints (t, m), whenever it removes (t, m) from the event list.

Example 2.8 (A Sequence of Snapshots in the Simulation of the Assembly Line)

We consider the assembly line example and show in Table 5 a partial sequence of event lists and clock values.

Initialize::
  clock := 0;
  event list := {(t_i, m_i) | message m_i will be sent at t_i unless the sender of m_i receives a message before t_i; one such entry exists for each pp as the sender}.
Iterate::
  while termination criterion is not met **do**
    remove a smallest tuple (t, m) from the event list;
    simulate the effect of transmitting m at time t;
    {This may cause changes in the event list.
    Note however that any addition or deletion,
    (t', m') to the event list will have $t' \geq t$.}
    clock := t
  endwhile

Figure 7. The sequential simulation algorithm.

Table 5. Partial Sequence of Event Lists and Clock Values

Clock	Event list	Smallest tuple
0	⟨(5, source, A, 1), (∞, A, —, —), (∞, B, —, —), (∞, C, —, —)⟩	(5, source, A, 1)
5	⟨(7, source, A, 2), (9, A, B, 1), (∞, B, —, —), (∞, C, —, —)⟩	(7, source, A, 2)
7	⟨(30, source, A, 3), (9, A, B, 1), (∞, B, —, —), (∞, C, —, —)⟩	(9, A, B, 1)
9	⟨(30, source, A, 3) (19, A, B, 2), (21, B, C, 1), (∞, C, —, —)⟩	(19, A, B, 2)

Notes on Parallel Execution. It should be obvious that to process more than one tuple at once, say, both (t, m) and (t', m'), we must be sure that these two events are independent, that is, that execution of one will not in any way affect the execution of the other. This requires us to know more about the cause–effect relationship among messages. We consider these issues in the next section and develop a basic scheme for distributed simulation.

3. DISTRIBUTED SIMULATION: THE BASIC SCHEME

In this section we introduce a model of distributed computation and show how a simulation may be carried out by a set of communicating processes. We limit our discussion here to a basic scheme, one which can result in deadlock. More sophisticated schemes that resolve deadlock are discussed in the next section.

3.1 A Model of Asynchronous Distributed Computation

A distributed system consists of a finite number of processes and *directed channels* connecting some pairs of processes. To distinguish these processes from physical processes, we call them *logical processes* or *lp*'s. Each lp may execute sequential code and two special commands: *receive* and *send*. In a send, an lp names an outgoing channel and a message that is to be sent along that channel. Execution of the send results in the message being deposited on the named outgoing channel; the sender then proceeds with the execution of its code. Each message takes an arbitrary but finite time to reach its destination. Messages sent along a channel are delivered in the sequence in which they are sent. In a receive command, an lp names one or more incoming channels from any one of which it wishes to receive a message. An lp wishing to receive may have to wait until a message arrives along one of the incoming channels. Note that our communication protocol is extremely simple and can be implemented on many existing machine architectures.

A set of lp's D is *deadlocked* at some point in the computation if all of the following conditions hold: (1) every lp in D is either waiting to receive or is terminated; (2) at least one lp in D is waiting to receive; (3) for any lp_i in D that is waiting to receive from some lp_j, lp_j is also in D, and there is no message in transit from lp_j to lp_i.

It follows then that none of the lp's in D will carry out any further computation since they will remain waiting for each other.

3.2 Basic Scheme for Distributed Simulation

To simulate any given physical system, we construct a distributed logical system as follows. We will associate one lp with each

pp; lp_i will simulate the actions of pp_i. If pp_i can send messages to pp_j, there is a *channel* from lp_i to lp_j.

An lp can simulate the actions of a pp up to time t if the lp knows the initial state and all messages that the corresponding pp receives up to time t. This is because, from the *realizability* property, no future message (message received by the pp after time t) can affect the pp's behavior at t. We note further that an lp may be able to simulate a pp beyond time t, even though it knows its input messages only up to time t, as shown in the following example.

Example 3.1 (An lp May Predict the Future)

Consider a typical nonpreemptive first come, first served (FCFS) server, which spends exactly 10 units of time servicing each job. Assume that a job arrives at time t when this server is idle. From this information about input messages up to time t, we can predict the behavior of the server up to time $t + 10$: It will produce no output between times t and $t + 10$, but it will output a message at $t + 10$, sending the job that has been serviced to its next destination.

From these observations, we can construct an algorithm for distributed simulation. We note that the times at which pp's send messages must be encoded into the message that the lp's send: *If messsage m is sent by pp_i to pp_j at time t, message (t, m) will be sent by lp_i to lp_j at some point during simulation.*

We make a *chronology* requirement: If an lp sends a sequence of messages $\langle \cdots (t_i, m_i), (t_{i+1}, m_{i+1}) \cdots \rangle$ to another lp, then $t_i < t_{i+1} \cdots$. The implication of this requirement is that if lp_i receives (t, m) from lp_j, then it knows *all* messages that pp_i receives from pp_j up to and including time t, because any future message will have a higher t component.

Define the *channel clock value* of a channel to be the t component of the last message received along that channel; the channel clock value is 0 if no message has been received along that channel. Clearly, every lp_i knows all messages received by the corresponding pp_i up to time $T_i = \min_j \{t_j\}$, where t_j's are the channel clock values of all incoming channels to that lp, and the minimum is taken over all these incoming channels. We call T_i the clock value of lp_i. Hence, lp_i can safely simulate pp_i up to T_i; that is, it can deduce every message that pp_i sends up to time T_i. Also, lp_i *may* be able to deduce pp_i's message transmissions beyond T_i. In any case, lp_i will send messages corresponding to all the messages it can deduce for pp_i. The basic simulation algorithm followed by lp_i is sketched next; we assume that all messages are sent at $t > 0$ in the physical system.

3.2.1 Basic Distributed Simulation Algorithm for lp_i (See Figure 8)

Note. The lp's that have no incoming channels are called *source lp*'s. Each source lp also follows this algorithm: It simply sends messages until the simulation completion criterion is met. A *sink lp* simply receives messages and otherwise does not affect the simulation.

Example 3.2 (Distributed Simulation of the Assembly Line)

Let us review the assembly line example (Example 2.4). In the following, we have one lp each for the source, the sink, work station A, work station B, and work station C.

See Table 3 for the job generation and processing times.

Figure 9 shows the messages sent by each lp; an arrow from (t, m) to (t', m') means that sending of (t, m) precedes sending of (t', m').

Note in this example that the source can send its messages to A without waiting for any input; A can send the ith message to B only after receiving the ith message from the source, etc. Two messages on different lp's between which there is no sequence of arrows are independent and hence may be transmitted simultaneously in the simulator. For instance, (32, Source, A, 4), (31, A, B, 3), (36, B, C, 2), (23, C, Sink, 1) can possibly be transmitted simultaneously. The five lp's form a pipeline through which each job passes. If the speeds of the

Initialize:: $T_i := 0$ {All messages received by pp_i up to T_i, are now known to lp_i}
while simulation completion criterion is not met **do**
{simulate pp_i up to T_i by doing the following}::
for each outgoing channel, compute the sequence of messages
$\langle (t_1, m_1), (t_2, m_2) \cdots (t_r, m_r) \rangle$, where $t_1 < t_2 \cdots < t_r$ and, pp_i sends m_j at time t_j along this channel;
send each message in sequence along the appropriate channel;
{Note: *All* messages sent by pp_i up to T_i can be deduced by lp_i and sent; also *some* messages to be sent beyond T_i may be predicted by lp_i and sent. Only new messages that have not been sent before are sent. Also note that some or all of these message sequences may be empty.}
{receive messages and update T_i until T_i changes value}::
$T_i' := T_i$;
while $T_i' = T_i$ **do**
wait to receive messages along all incoming channels;
upon receipt of a message, update lp_i's internal state and recompute T_i, the minimum over all incoming channel clock values
endwhile
endwhile

Figure 8. Basic distribution algorithm for lp_i.

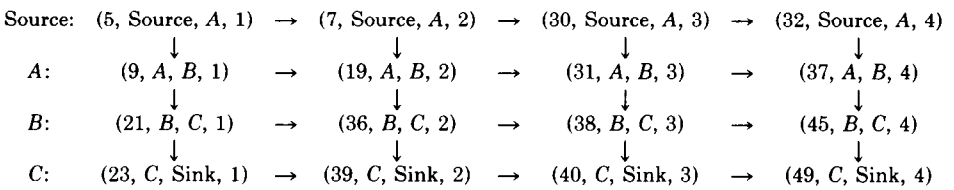

Figure 9. Messages sent by each lp.

lp's are approximately equal, and the transmission delays between lp's are approximately equal, then the pipeline should work at full efficiency; one job is input and one job is output per cycle after an initial delay of three cycles.

This is about the simplest simulation example one can think of. We study a harder example next.

Example 3.3 (A Primitive Computer system) (See Figure 10)

We have one lp each for the source, the CPU, Proc1, Proc 2, *M*, *B* and the sink. For this example, assume that jobs arrive at the CPU from the source every 5 time units starting at time 3, that jobs spend 1 unit at the CPU, that jobs alternately go to Proc1 and Proc2 from *B*, and that a job spends 5 units at Proc1, 18 units at Proc2, and no time at *B* or *M*. We show the sequence of messages and their dependencies in Figure 11. (To simplify the diagram, we have not shown the arrows between messages at a pp.)

Note the behavior of the lp corresponding to *M*. Assume that it first receives (27, Proc2, *M*, 2) from the lp corresponding to Proc2. This is possible if, for instance, the lp corresponding to Proc2 were considerably faster than the one corresponding to Proc1. Then the lp for *M* can only infer that it will not receive any other message from the lp corresponding to Proc2 with time component smaller than 27. However, it cannot assert anything about messages from Proc1; it can thus simulate pp *M* only up to time 0. Suppose that it next receives (45, Proc2, *M*, 4); it must still wait. The next input is, say (9, Proc, *M*, 1). Then the lp corresponding to *M* can assert that it knows all inputs that *M* receives up to time 9 and hence predict all of *M*'s outputs at least up to 9; therefore, it can output (9, *M*, Sink, 1), since jobs spend no time at *M*. The rest of the outputs of *M* are easy to see. Finally, note that *M* cannot output

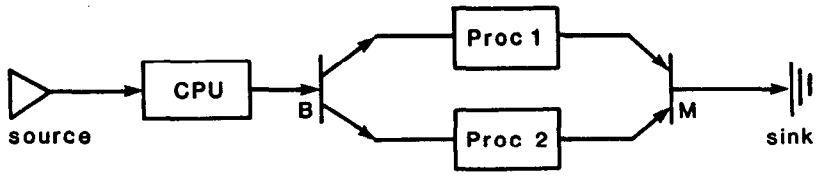

Figure 10. A primitive computer system.

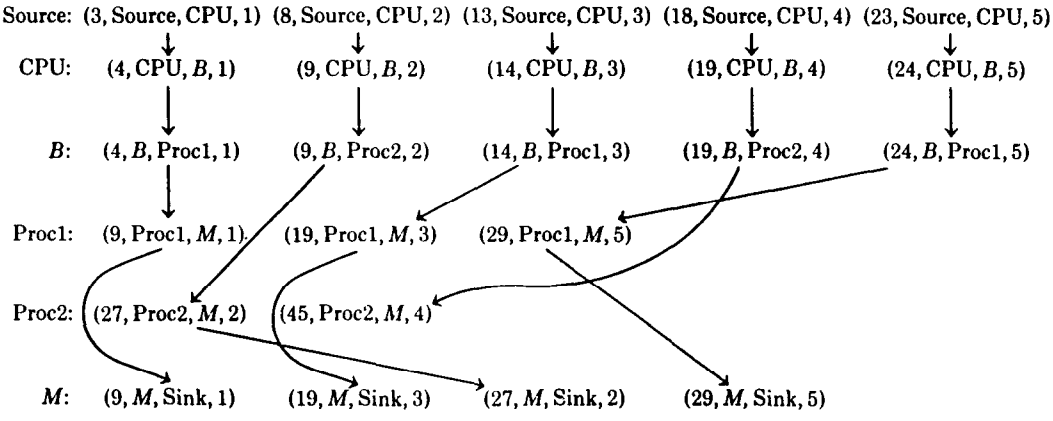

Figure 11. Sequence of messages and their dependencies.

(45, *M*, Sink, 4) at the very end, because it does not know whether it will receive a message with a *t* component lower than 45 from the lp corresponding to Proc1. An extra message with a *t* component exceeding 45 must be sent from Proc1 to *M* to "flush out" this message. We discuss this issue later.

3.3 Partial Correctness of the Basic Distributed Simulation Scheme

Correctness of a distributed simulation algorithm consists of two parts: (1) If a message *m* is transmitted in the physical system at time *t*, then (*t*, *m*) is transmitted in the simulator; (2) if (*t*, *m*) is transmitted in the simulator, then message *m* was transmitted at time *t* in the physical system. These statements are not quite true for the basic distributed simulation scheme just presented. As we observed in the last example, job 4 is sent at time 45 from *M* to the sink in the physical system, but the corresponding message is never sent in the simulator. Therefore, we can prove only one part of the correctness condition stated above: Whatever is transmitted in the simulator actually happens in the physical system. We are postponing discussion of the converse statement—if message *m* is transmitted at time *t* in the physical system, then (*t*, *m*) is transmitted in the simulator—to the next section.

Define a simulation to be *correct* at some point if it meets the following two conditions: (1) If message *m* is sent at time *t* along channel *e* in the physical system, and *t* is less than or equal to the channel clock value of channel *e* at this point in simulation, then (*t*, *m*) has been sent along channel *e* in the simulation; (2) if (*t*, *m*) has been sent in the simulation, then message *m* is sent at time *t* in the physical system.

We note that, in a simulation that is correct at some point, every lp must have received a *correct input sequence* along every incoming channel, that is, every message on this channel that has been transmitted in the physical system up to this channel clock value has been received along this channel in the simulation, and

vice versa. We assume that every lp *correctly simulates* the corresponding pp; that is, *any message sent by an lp is correct provided that all messages it has received prior to sending this message are correct.* Clearly a simulation is correct if and only if every lp has sent *correct output sequences* along every outgoing channel. Theorem 2 follows by applying induction on the number of messages transmitted in the simulation.

Theorem 2

Simulation is correct at every point.

Proof. Simulation is obviously correct, by definition, when no message has been transmitted in the simulation. Assume that a simulation is correct up to some point. The next messsage in the simulation is sent by some lp_i. Since simulation is correct prior to this message transmission, lp_i has received correct input sequences so far. From our assumption that lp_i correctly simulates pp_i, the output sequences of lp_i, including the last message sent, are correct. Every other lp has sent correct sequences so far, from the inductive hypothesis. Hence the simulation is correct following the last message transmission. □

In a similar manner, we can derive the following result.

Theorem 3

All messages sent by one lp to another are chronological in their time components.

3.4 Features of the Basic Distributed Simulation Scheme

3.4.1 The Problem of Deadlock

Theorem 2 tells us only that whatever is transmitted in the simulator corresponds to a message in the physical system. As we have noted earlier, not all messages in the physical system are transmitted in the simulator using the basic simulation scheme. In fact, the next example shows a system in which no message is transmitted to a subsystem in the simulator.

Example 3.4 (A Deadlocked Subsystem in a Distributed Simulation)

Consider a physical system in which the source sends messages to a branch point B, and B routes the messages to Proc1 or Proc2. After some finite time, each message is sent from Proc1 or Proc2 to a merge point M, after which it enters a network N (see Figure 12). Consider the case in which B sends *every* message to Proc1. Then in the simulation, the lp corresponding to M will never receive a message from Proc2. Hence the channel clock value for the channel (Proc2, M) will remain at 0 and the lp for M will never send a message. The subsystem N will thus never receive a message.

We show another example in which deadlock arises owing to a circular pattern of waiting among the lp's.

Example 3.5 (Cyclic Waiting in a Distributed Simulation) (See Figure 13)

Consider a network of three processes and a source, shown schematically in Figure 13. The number on each channel is the channel clock value; that is, the last message sent from x to y and received by y had a t component of 20, and so on. Suppose that none of x, y, z will now send a message unless they receive a message, that is, they can predict no future messages.

We can see that z will not send a message unless x first sends a message to y. Hence x need not wait for z; it can process the next message from the source. However, none of the lp's corresponding to x, y, z have this global knowledge; they only have local knowledge of the behavior of each individual pp. Therefore, x cannot proceed unless it receives from z, z cannot proceed unless it receives from y, and y cannot proceed unless it receives from x, leading to a deadlock.

3.4.2 Simulation Snapshot

In a sequential simulation, it is possible to assert that the simulator has completed simulation up to the time given by the clock: every pp must have been simulated up to this point in time. We cannot make

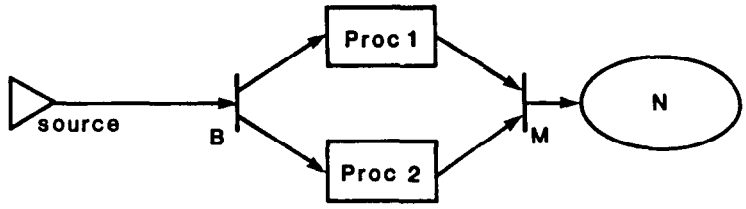

Figure 12. A distributed simulation that does not progress.

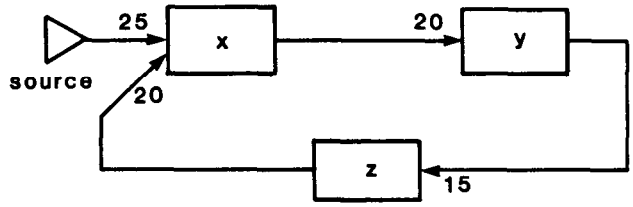

Figure 13. A distributed simulation that deadlocks.

a similar statement for distributed simulation, because each lp may have simulated the corresponding pp to a different point in time. For instance, in the example of the primitive computer system (Example 3.3), we can assert at the end that the lp's have simulated the corresponding pp's as follows: (Source: 23)(CPU: 24), (B: 24), (Proc1: 29), (Proc2: 45), (M: 29).

Let T, *the clock value of the simulator*, be the minimum of all lp clock values. We can assert that at any point in the simulation, the physical system has been simulated up to the simulator's clock value, even though some individual lp's may have simulated the corresponding pp's far beyond T.

3.4.3 Encapsulation of Physical Processes by Logical Processes

In distributed simulation, the radical departure from sequential simulation is the lack of any global control. (We show deadlock resolution without global control in the next section.) Since a pp is simulated entirely by one lp, various different simulations of a pp can be attempted by substituting different lp's for it. Furthermore, the correctness of simulation can be checked one lp at a time—the proof of correctness is naturally partitioned among lp's, that is, we show that each lp correctly simulates the behavior of the corresponding pp. We have shown that, if each lp behaves correctly, the simulation as a whole behaves correctly. This observation may lead to major simplifications in designing complex simulations. In fact, distributed simulations can be implemented using existing sequential simulations; instead of reporting to a central event-list manager, an lp sends messages. In all other respects, the core of the simulation remains unchanged.

4. Distributed Simulation: Deadlock Resolution

We have seen in the last section that the basic distributed simulation scheme may lead to deadlock even in acyclic networks. In this section we present several different approaches to resolution of deadlock. We comment on some of the most viable approaches to deadlock resolution.

4.1 Overview of Deadlock Resolution

In all the examples we have seen so far, the simulator clock value (i.e., the minimum of all lp clock values) remains at some final value T forever. If T is smaller than the point up to which we need to run the simulation, we have to apply some other

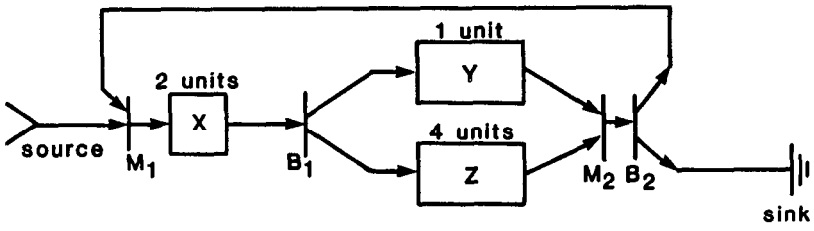

Figure 14. A physical system with loop.

scheme to advance the simulation. For instance, in the example of the primitive computer system (Example 3.3), the lp corresponding to M cannot proceed any further unless it is told that Proc1 will never send it a message. In Example 3.5, lp x must be told that it will never receive any input along zx until x first sends a message. The first scheme we describe, using *null* messages, is effectively an implementation of this idea [Bryant 1977; Chandy and Misra 1979]. We also discuss some other schemes that avoid deadlock using different kinds of overhead messages.

4.2 Deadlock Resolution Using *Null* Messages

We postulate a new kind of message to be used by the simulator: (t, *null*) sent by lp_i to lp_j means that pp_i sends no message to pp_j between the current channel clock value of the channel from lp_i to lp_j and t; therefore, any future message from lp_i to lp_j will have a t-component exceeding t. Clearly *null* messages have no counterpart in the physical system. A *null* message is used to announce absence of messages. Absence of messages in a physical system at time t is recognized by no message being transmitted at that time. Unfortunately, the basic scheme of the last section cannot guarantee absence of messages to an lp without sending it an actual (*nonnull*) message having a higher t-component value.

We now propose modifications to the basic algorithm of Section 3 to incorporate *null* messages. Let us first review the basic distributed simulation scheme of the last section. T_i denotes the clock value of lp_i. Whenever lp_i receives a message, it properly updates T_i, and, if T_i changes in value, then lp_i advances the simulation of pp_i up to T_i. At this point lp_i predicts for each outgoing channel, a sequence of messages that the pp_i would have sent. Thus lp_i typically generates $\langle (t_{j1}, m_{j1}), (t_{j2}, m_{j2}), \cdots \rangle$ for transmission to lp_j, for every j to which it has outgoing channels. Some of these sequences may be empty, in which case no message is sent to the corresponding lp. Suppose that lp_i can further predict that after transmission of this message sequence pp_i *will not* send any more messages to pp_j until time t_j. Then, in the new scheme, *lp_i sends (t_j, null) to lp_j after sending the genuine message sequence.* Since lp_i knows the state of the corresponding pp up to time T_i, it can predict all messages (that are to be sent) and absence of messages, at least up to T_i. Therefore, every outgoing channel will have a last message on it with time component equal to or greater than T_i. Note that, in any iteration, only the last message sent along a channel may be a *null* message.

Reception of a *null* message is treated in the same manner as the reception of any other message: It causes the lp to update its internal state, including the clock value, and (possibly), to send messages.

Suppose it is required to simulate the physical system up to some time T. Then every source must send messages until the t component of the last message equals T; if no *nonnull* message exists with this property, then finally, (T, *null*) should be sent.

Example 4.1

Consider the physical system shown schematically in Figure 14.

We study the progress of one possible simulation run of this physical system. The

source sends out jobs that are processed at X for 2 time units. Jobs are routed alternately to Y and Z from B_1. Y processes a job for 1 unit and Z for 4 units. Every job loops through the system twice; that is, the first time a job arrives at B_2, it is sent back to M_1, and on the second arrival at B_2 it is sent to the sink.

Table 6 shows a succession of message transfers, where each horizontal row is a time slice and each entry corresponds to a single activity of one of the processes. It is evident that several activities may happen concurrently.

4.3 Correctness of the Simulation Algorithm

The partial correctness results of the last section still apply. The only difference now is the presence of *null* messages. We define the simulation to be *correct* at some point, if it is correct according to the definition of Section 3 after ignoring *null* messages.

Theorem 4

Simulation is correct at every point.

Proof. The proof is almost identical to the previous proof and hence omitted here. □

The next theorem shows the power of adding null messages: We show that we have a deadlock-free system that can simulate a physical system up to any desired time.

Theorem 5

Assume that every source process sends messages until the t-component of a message equals T. Then every lp will simulate the corresponding pp, at least up to T.

Proof. Consider the point at which the simulation terminates, that is, at which all messages that have been sent have been received and no lp has any outstanding message to send. The following observation is critical: For every lp (except a source lp) there exists an incoming channel to that lp whose channel clock value is less than or equal to the channel clock value of every outgoing channel from that lp. This observation follows because (1) an lp that has received messages at least up to t along every input channel must have sent messages (t', m'), $t' \geq t$, along every outgoing channel; (2) every message that has been sent has been received when simulation terminates. Note that (1) could not be asserted in the basic scheme because an lp need not send out messages with higher t-component values than the input messages.

We now claim that the channel clock value for every channel is at least T. If not, consider a channel e_1 for some lp, whose channel clock value is t_1, with $t_1 < T$. According to the above observation, there exists a channel e_2, which is an incoming channel to this lp, such that e_2's channel clock value is t_2, where $t_2 \leq t_1$. Continuing in this manner, we can construct a sequence of channels, $e_1, e_2, \ldots, e_i, \ldots$ such that for all i, e_{i+1} is a predecessor channel of e_i and $t_{i+1} \leq t_i$, and we have $t_1 < T$. Since the physical network is finite, eventually we either (i) get to a source lp, or (ii) have a cycle of channels. In the first case, since every source lp sends messages until the t component of the last message sent is T, we cannot have channel clock value of any outgoing channel of a source lp smaller than T. In the second case, all channel clock values in the cycle are equal to t_1 and $t_1 < T$. From the predictability property (Section 2), for this cycle and this t_1, there exists a pp, say pp_j, whose outputs can be determined beyond t_1, given its inputs up to t_1. Hence, lp_j has some messages to send, which contradicts our assumption that the simulation has terminated. Therefore, the channel clock value of every channel is at least T, and hence the simulation clock value is at least T.

We have implicitly used the fact that, for any finite T, only a finite number of messages may be transmitted in the logical system. This is derived from the predictability property, in which the parameter ϵ, $\epsilon > 0$, is a fixed quantity. A more rigorous proof of this boundedness property may be found in Chandy and Misra [1979]. □

4.4 Discussion

It is interesting to note that the simulator never deadlocks; If the physical system deadlocks, the simulator continues

Table 6. Message Transmissions in the Simulation of Example 1

Step	Source Snds	M_1 Rec	M_1 Snds	X Rec	X Snds	B_1 Rec	B_1 Sends Y	B_1 Sends Z	Y Rec	Y Snds	Z Rec	Z Snds	M_2 Rec	M_2 Snds	B_2 Rec	B_2 Snds M_1	B_2 Snds Sink
1					(2,null)					(1,null)		(4,null)					
2						(2,null)											
3							(2,null)	(2,null)					(1,null)				
4									(2,null)		(2,null)		(4,null)				
5	$(2,J_1)$									(3,null)		(6,null)		(1,null)			
6	$(3,J_2)$	$(2,J_1)$											(6,null) (3,null)		(1,null)		
7		$(3,J_2)$												(3,null)		(1,null)	(1,null)
8	$(6,J_3)$	(1,null)													(3,null)		
9			(1,null)													(3,null)	(3,null)
10		(3,null)	$(2,J_1)$														
11			$(3,J_2)$	(1,null)													
12		$(6,J_3)$		$(2,J_1)$	(3,null)												
13					$(4,J_1)$	(3,null)											
14				$(3,J_2)$			(3,null)	(3,null)									
15					$(6,J_2)$	$(4,J_1)$			(3,null)		(3,null)						
16						$(6,J_2)$	$(4,J_1)$	(4,null)		(4,null)		(7,null)					
17							(6,null)	$(6,J_2)$	$(4,J_1)$		(4,null)		(4,null)				
18										$(5,J_1)$	$(6,J_2)$	(8,null)		(4,null)			
19												(10,J_2)	$(5,J_1)$				
20														$(5,J_1)$	(4,null)		
21													(7,null)			(4,null)	(4,null)
22		(4,null)											(8,null)		$(5,J_1)$		
23			(4,null)										(10,J_2)			$(5,J_1)$	(5,null)
24		$(5,J_1)$		(4,null)													
25			$(5,J_1)$		(6,null)												
26				$(5,J_1)$													

computation by transmitting *null* messages with increasing t values. This correctly simulates the corresponding physical situation, in that, while time progresses, no messages are transmitted in the physical system and the simulator terminates with every clock value at least at T. The simplicity of this scheme is one of its most attractive points. It requires small coding changes to send out null messages. Furthermore, the requirement of unbounded buffers between two lp's is not really necessary. The same results hold if there are only a finite number of buffer spaces between every lp_i and lp_j, and lp_i has to wait to send if all buffer spaces are currently full. The proof that there is no deadlock in this situation is essentially contained in Chandy and Misra [1979].

The metric of interest in performance calculations is the *turnaround time*, that is, the amount of time it takes to complete the simulation, rather than processor utilization, that is, the fraction of time the processors are utilized. In fact, one would expect the processors to be lightly utilized. The other parameter of interest, line bandwidth, has not received adequate attention.

Empirical studies show that this scheme is quite efficient for acyclic networks [Seethalakshmi 1979]. Several factors seem to affect the efficiency in general networks:

(1) *Degree of Branching in the Network.* Consider a network with one source and one sink. The number of distinct paths between the source and the sink is a (rough) measure of the amount of branching in the network. *Null* messages tend to get created at branches and they may proliferate at all successive branches (if not subsumed). So, one would expect that the fewer the number of branches, the better the performance. Empirical studies seem to confirm this [Seethalakshmi 1979]. Theoretically, optimum efficiency is achieved for a tandem network (the assembly line example of Section 2, Example 2.4), and excellent results are obtained for low-branching-type networks. In general, acyclic networks exhibit reasonably good performance levels.

Experiments were carried out by Peacock et al. [1979a, 1979b] on networks of various topologies. Their conclusions are: "For some topologies of queueing networks models, this approach results in a speedup in the total time to complete a given simulation. However, for other topologies, especially those with loops, the speedup may not be significant." They also investigated several different ways of partitioning the physical network so that more than one pp may be implemented on one lp.

(2) *Time-Out Mechanisms to Prevent Null Message Transmission.* A slight modification to the scheme of this section may save a considerable number of message transmissions. A *null* message (t, m) has no effect if it is followed by another message (t', m'), $t' > t$. Therefore, it may be efficient to delay transmissions of *null* messages in the hope that future messages received by an lp would make it unnecessary to transmit them at all. Clearly, the amount of time τ that an lp waits before transmitting a null message is of importance. If $\tau = 0$, we have the algorithm as stated in this section. If $\tau = \infty$, *null* messages are never transmitted, and then we have the basic algorithm of Section 3, which may lead to deadlock. Other values of τ are of potential interest, but no empirical studies have been performed for other values.

(3) *Amount of Buffering on Channels.* The number of buffer spaces on channels seem to have substantial effects on performance [Quinlivan 1981; Seethalakshmi 1979]. When the number of buffer spaces was reduced to 0, senders had to wait until the receivers were ready to receive, and a considerable amount of time seemed to be spent in waiting. The number of buffer spaces was then increased and the following rule was used to annihilate null messages: Any message put in the buffer *after* a *null* message (and therefore with a higher t component) annihilates any *null* message ahead of it still in the buffer. The annihilation rule is somewhat similar to the time-out mechanism. It was found that, in the simulation of a certain class of queuing networks, the performance improved rapidly until the number of buffer spaces on a channel approached 10, increased less rapidly until about 20, and remained

essentially unchanged thereafter. However, these numbers cannot be applied directly to other problems; we expect these numbers to depend on the type of problem and the speeds of processors and lines.

Next, we describe two different schemes that limit the number of *null* message transmissions. In the query–reply scheme (4.5), no *null* message is transmitted along a *channel*(y, z) until z demands to have (y, z)'s clock value increased; y may then be forced to send a *null* message incrementing the channel clock value of (y, z). In the circulating marker scheme (4.6), a single marker is used to carry *null* messages. It is not clear that either of these schemes is superior to the basic scheme, where *null* messages are transmitted after a proper time-out. Only empirical investigations can settle these issues.

4.5 Demand-Driven *Null* Message Transmisision

Suppose that, on the basis of time-out, lp z asks lp y to advance the clock value of (y, z). Such an advance is always possible if (y, z)'s current clock value is less than y's clock value (which is the minimum of channel clock values of (x, y), for all x). In this case y sends a message, possibly null, advancing (y, z)'s clock value. However, if (y, z)'s clock value equals y's clock value, then no advance may be possible. In this case y has to advance its own clock first, making the same kind of request of all lp x for which (x, y)'s clock value equals y's clock value. Hence, effectively the request has been propagated by y. Only when y's clock value increases beyond (y, z)'s clock value can y send a message to z incrementing (y, z)'s clock value. The propagations of requests may form a cycle, in which case a deadlock is detected.

We sketch the algorithm below for an lp y. A *query* is a message that is sent by one lp to another to request that a channel clock value be advanced; queries will be propagated, in general. Hence, we assume that a query contains the path it has traversed and the channel clock values of all channels along the path; query $(p_0\ p_1 \cdots p_n)$ denotes a query initiated by p_0 and sent from p_i to p_{i+1}, for all i, $0 \le i < n$. In addition, the query contains t_i, clock value of the channel (p_{i+1}, p_i), for all i, $0 \le i < n$. It is obvious that $t_j \ge t_{j+1}$ and $0 \le j < n - 1$.

A *reply* for a query contains the query and a new clock value for the last channel in the query; that is, for a query $(p \cdots y\ x)$, there is a new (larger) clock value for channel (x, y). Additionally, a reply may contain one or more messages. In the following we use "y has *query* $(p \cdots y)$" as a Boolean proposition that is set true or false in the algorithm. Actions of lp y are described by the following rules.

(1) Initiating a query ::

if time-out **then** y has a *query*(y).

(2) Upon receiving a reply to *query* $(p \cdots y\ x)$ ::

advance channel clock value of $(x\ y)$ and receive messages, if any, in the reply; y has *query* $(p \cdots y)$.

(3) lp y has or receives *query* $(p \cdots z\ y)$ {where the sequence may have only one element y} ::

if y appears more than once in the query
then detect deadlock {recovery from deadlock is treated below}
else if lp y can advance the channel clock value of $(y\ z)$
then send reply to z (including the query, the new channel clock value and messages, if any)
else send query $(p \cdots z\ y\ x)$ to every x for which clock value of channel $(x\ y)$ = clock value of lp y (unless such a query has been sent and no reply is yet received).

A query eventually leads to either detection of a deadlock or increase of some channel clock value; we refer the reader to Chandy and Misra [1982] and Chandy et al. [1983] for the essential ideas in the proof of this claim.

Consider the situation in Example 3.5. A query initiated by y is received by x, propagated to z, and then propagated to y, which detects deadlock. If the query were initiated by x, it is sent to z and is propagated by z to y; then y replies, advancing the channel clock value of (y, z) to 20, z

sends the query to y again, and y sends the query to z, which detects deadlock.

Resolution of deadlock is surprisingly difficult. In the above example x can detect deadlock, but it cannot, in general, advance its own clock to 25, the channel clock value of the channel outside the deadlocked set. This is because the physical process x may send out a message, say at time 22, if it receives no message between 20 and 22 along either input channel. This is certainly conceivable, for example, if x is an alarm clock that is set at time 20 to go off at time 22 unless it is canceled before that time. Therefore, lp x's clock value cannot be advanced to 25. Resolution of deadlock may be accomplished by determining the minimum of the "next event times": For every lp in the deadlocked set, the time at which a message will be sent (provided no further message is received up to then) is determined, and the clock value of the lp with the minimum clock value is advanced to this time. This calculation may be carried out in a centralized or decentralized fashion; in fact, the query may carry the next event time for each lp it has seen, in which case the detection of deadlock can also determine which lp may restart and at what clock value.

4.6 Circulating Marker for Deadlock Detection and Recovery

A suggestion has been made in Chandy and Misra [1981] to let the basic simulation scheme deadlock, detect deadlock, and recover from it. We now discuss two methods for deadlock detection and recovery.

Consider a marker that continuously circulates in a network. It follows a cycle of channels such that it traverses every channel of the network sometime during a cycle. Such a cycle exists if the network is strongly connected; new channels may be added to the network to make it strongly connected. The marker is merely a special type of message. It initially starts at some lp. If an lp receives the marker, its obligation is to send the marker (along its designated route) within a finite time of being idle (i.e., not having anything more to send). We let the marker carry some information for deadlock detection, as described below.

Each lp has a 1-bit flag to show whether the lp has received or sent a message since the last departure of the marker from the lp. We say that an lp is *white* if it has neither received nor sent a message since the last departure of the marker from the lp; the lp is *black* otherwise. Initially all lp's are black. The marker declares deadlock when it finds that the last N lp's that is has visited were all white when it arrived at the lp, where N is the number of channels in the network. The algorithm is correct if messages between two lp's, including the marker, are received in the order sent; see Misra [1983] for a precise description and proof of this result.

We can use this scheme to detect and recover from deadlock. The marker, in addition to keeping the number of white lp's it has seen since it last saw a black lp, carries the minimum of "next-event-times" for the white lp's it visits: Each white lp can report the time of the next event, assuming it receives no further messages, to the marker, and the marker merely keeps track of the smallest of these, and the corresponding lp. When the marker detects deadlock, it knows the next event time and the lp at which this next event occurs. Therefore, it can restart that lp. Alternately, a central process may broadcast (send messages to all lp's) to advance their clocks to the next event time in the system.

The overhead messages in this case are for marker transmissions. If deadlocks are infrequent, the marker may move slowly. In this case the deadlock may be detected some time after it occurs, but the proportion of overhead messages to genuine messages will be low.

An elegant variation of this deadlock detection scheme has been discovered by Chandy [unpublished notes] and refined by Kumar [1986]. As before, there is a marker that visits the lp's. However, it visits them in an arbitrary fashion, with the only requirement being *that it visit each lp eventually.* It collects the following information from an lp when it visits it: (1) status of the lp (an lp is *idle* if it will send no more messages unless it first receives a

message; it is *nonidle*, otherwise), and (2) number of messages received along every input channel and number sent along every output channel of the lp. This information overwrites any previous information collected from that lp. Note that the information collected by a marker may become obsolete if an lp receives messages, becomes nonidle, and/or sends messages after the marker collects the information from the lp. Yet, the marker can declare deadlock if the information it has collected shows that every lp is idle and, for every channel, the number of messages sent equals the number of messages received.

These schemes have to be modified for detecting deadlocks within a subnetwork in the logical system; we can determine, through preprocessing, the subnetworks that may deadlock, and then we can assign markers to these subnetworks.

5. SUMMARY AND CONCLUSION

In this section, we summarize the discussions about distributed simulation, its status, problems, and future research directions. We hope to have demonstrated that distributed simulation may be applied in every situation in which sequential discrete-event simulation may be applied. Our examples have been predominantly from the area of computer systems, since a queuing network description of a computer is a physical system. However, our physical systems encompass a large variety of real-world applications. Implementation of distributed simulation is possible in any language that allows creation of message communicating processes.

The assignment of logical processes to physical processors should follow the guideline that the message traffic among processors be as low as possible. Message communication may be accomplished either through a common memory (messages are deposited in a common memory by the sender and removed by the receiver) or by other interaction mechanisms among processors. The important criterion is how loosely coupled the processors are. If two processors are tightly coupled, that is, if the logical processes on these processors exchange a large number of messages, then the processors must also exchange at least that many messages, and the message traffic will be heavy. If processors are loosely coupled, they can operate autonomously, that is, without communicating with other processors, for longer periods of time. It is also easier to avoid deadlock among a set of logical processes if they are simulated on one processor, because a centralized scheduler, employed for message communication, can also detect deadlock.

Static partitioning of the physical network among a fixed number of processors requires preprocessing prior to simulation. Preprocessing is useful for many other reasons, too. In the circulating marker algorithm, preprocessing is needed to determine a (static) cyclic path for the marker. Preprocessing could also be used to partition the lp's such that the amount of branching is reduced and cycles are mostly contained within one processor. Preprocessing can determine other simulation parameters for time-out, sizes of buffers on channels, etc. This is an area that has been extensively studied for sequential simulations. It needs to be studied again for distributed simulation, since the problems are somewhat different in nature.

We have sketched several variations of the basic scheme for deadlock resolution. There is little evidence yet of the superiority of any one scheme. The large number of heuristics suggests that some combination may be appropriate for particular problem domains. For instance, if we use a set of uniform processors, among which message communication is expected to be regular, we can expect that deadlock will rarely arise, and therefore a (slowly) circulating marker scheme would be preferable. Also, the marker can be used to collect statistical information about the simulation itself, and hence the simulation parameters, such as time-outs, can be dynamically changed.

We have not discussed specific hardware architectures that can support simulation. There has not been enough experimentation with distributed simulation to know where it spends most of its time, and whether any architectural improvement would be useful for all distributed

simulation problems. At present, any architecture that supports processes and communication among them would be appropriate.

Currently, the most important problem in distributed simulation is the empirical investigation of various heuristics on a wide variety of problems to establish (1) which heuristics work well for which problems and on which machine architectures, (2) how to partition the physical system among a fixed set of processors, and (3) how to set simulation parameters such as time-outs and buffer sizes.

ACKNOWLEDGMENTS

This research has been supported by a grant from the Air Force Office of Scientific Research under grant AFOSR 85-0252.

I am deeply indebted to my friend and colleague, K. M. Chandy of The University of Texas, Austin, for his help, advice, and stimulating ideas. I am thankful to the Computer Systems Lab at Stanford University, and in particular to Susan Owicki, for providing a proper environment for the initial preparation of this paper. I am grateful to Doug DeGroot, formerly of IBM, Yorktown Heights, and Devendra Kumar of The University of Texas, Austin, for their editing efforts and helpful comments on the first draft of this manuscript. Unusually thorough reviews by two anonymous referees and the help and guidance of Dick Muntz, associate editor of *Computing Surveys*, are deeply appreciated.

REFERENCES

BAGRODIA, R., CHANDY, K. M., AND MISRA, J. N.d. A message-based approach to discrete-event simulation. *IEEE Trans. Softw. Eng.*, to appear.

BEZIVIN, J., AND IMBERT, H. 1983. Adapting a simulation language to a distributed environment. In *Proceedings of the 3rd International Conference on Distributed Computing Systems* (Ft. Lauderdale, Fla.). IEEE, New York, pp. 596–603.

BIRTWISTLE, G. 1979. *DEMOS: A System for Discrete Event Simulation.* Macmillan Press, New York.

BIRTWISTLE, G. M., DAHL, O. J., MYHRHAUG, B., AND NYGAARD, K. 1973. *Simula Begin.* Auerbach, Philadelphia.

BRYANT, R. E. 1977. Simulation of packet communication architecture computer systems. Tech. Rep. MIT, LCS, TR-188, Massachusetts Institute of Technology, Cambridge, Mass.

CHANDY, K. M., AND MISRA, J. 1979. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. Softw. Eng. SE-5*, 5, 440–452.

CHANDY, K. M., AND MISRA, J. 1981. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM 24*, 4 (Apr.), 198–205.

CHANDY, K. M., AND MISRA, J. 1982. A distributed algorithm for detecting resource deadlocks in distributed systems. In *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (Ottawa, Ontario, Canada). ACM, New York, pp. 157–164.

CHANDY, K. M., MISRA, J., AND HOLMES, V. 1979. Distributed simulation of networks. *Comput. Netw. 3*, 105–113.

CHANDY, K. M., MISRA, J., AND HAAS, L. M. 1983. Distributed deadlock detection. *ACM Trans. Comput. Syst. 1*, 2 (May), 144–156.

CHRISTOPHER, T., EVENS, M., GARGEYA, R. R., AND LEONHARDT, T. 1983. Structure of a distributed simulation system. In *Proceedings of the 3rd International Conference on Distributed Computing Systems* (Ft. Lauderdale, Fla.). IEEE, New York, pp. 584–589.

DAHL, O. J., MYHRHAUG, B., AND NYGAARD, K. 1970. *Simula 67 Common Base Language.* Norwegian Computing Centre, Oslo, Norway.

FISHMAN, G. S. 1978. *Principles of Discrete Event Simulation.* Wiley, New York.

FRANTA, W. R. 1977. *Process View of Simulation.* Elsevier Computer Science Library, Operating and Programming Systems Series, P. J. Denning, Ed. Elsevier North-Holland, New York.

HOLMES, V. 1978. Parallel algorithms on multiple processor architectures. Ph.D dissertation, Computer Science Dept., Univ. Texas at Austin, Austin, Tex.

JEFFERSON, D. R. 1985. Virtual time. *ACM Trans. Prog. Lang. Syst. 7*, 3 (July), 404–425.

KUMAR, D. 1986. Ph.D dissertation (in preparation). Computer Science Dept., Univ. Texas at Austin, Austin, Tex.

LAVENBERG, S., MUNTZ, R., AND SAMADI, B. 1983. Performance analysis of a rollback method for distributed simulation. In *Performance '83*, A. K. Agrawala and S. K. Tripathi, Eds. North Holland, New York, pp. 117–132.

LONOW, G., AND UNGER, B. 1982. Process view of simulation in Ada. In *1982 Winter Simulation Conference* (San Diego, Calif., Dec. 6–8). IEEE, New York, pp. 77–86.

MISRA, J. 1983. Detecting termination of distributed computations using markers. In *Proceedings of the 2nd ACM Principles of Distributed Computing* (Montreal, Ontario, Canada). ACM, New York, pp. 290–293.

PEACOCK, J. K., WONG, J. W., AND MANNING, E. G. 1979a. Distributed simulation using a network of processors. *Comput. Netw. 3*, 1, 44–56.

PEACOCK, J. K., WONG, J. W., AND MANNING, E. G. 1979b. A distributed approach to queuing network simulation. In *Proceedings of the Winter Simulation Conference* (San Diego, Calif.). IEEE, New York, pp. 399–406.

QUINLIVAN, B. 1981. Deadlock resolution in distributed simulation. Master's thesis, Computer Science Dept., Univ. Texas at Austin, Austin, Tex.

REYNOLDS, P. 1982. A shared resource algorithm for distributed simulation. In *Proceedings of the 9th International Symposium on Computer* Architecture (Austin, Tex.). IEEE, New York, pp. 259–266.

SEETHALAKSHMI, M. 1979. A study and analysis of performance of distributed simulation. Master's thesis, Computer Science Dept., Univ. of Texas at Austin, Austin, Tex.

U.S. DoD 1982. *Reference Manual for the ADA Programming Language.* U.S. Department of Defense.

Received February 1985; final revision accepted May 1986