Conditional Knowledge as a Basis for Distributed Simulation

K. Mani Chandy
and
Jay Misra

Computer Science Department
California Institute of Technology

5251:TR:87

5251:TR:87

CONDITIONAL KNOWLEDGE AS A BASIS FOR DISTRIBUTED SIMULATION*

(Preliminary Draft)

K. Mani Chandy
Computer Sciences Department 256-80
California Institute of Technology
Pasadena, CA 91125
(on leave of absence from The University of Texas at Austin)

Jay Misra
Computer Sciences Department
University of Texas
Austin, Texas 78712

Abstract

A goal of this paper is to explore different ways of implementing distributed simulations. Distributed simulation grew out of sequential simulation, and it is possible that the way we think about distributed simulation is unduly influenced by its sequential origins. To free ourselves from unnecessary restrictions on the way we design distributed simulations, in this paper we define the distributed simulation problem somewhat differently than in the literature. We propose the concepts of "knowledge" and "conditional knowledge", to help us obtain a general framework to reason about distributed simulations without too close a coupling with any specific implementation method. The framework appears helpful in designing new ways of distributed simulations.

Empirical studies of distributed simulations report widely varying results: some studies report improvements in speed that are almost linearly proportional to the number of computers in the system, while other studies report that distributed simulation is even slower than sequential simulation. The framework proposed in this paper seems to help in explaining the wide differences observed in empirical studies.

Using our framework, we attempt to suggest properties that efficient "general-purpose" distributed discrete-event simulations must have.

The paper assumes little prior knowledge of the literature on simulation or distributed systems. We hope that the paper will serve as a tutorial in addition to providing additional insight.

*supported by ONR Grant Nos. N00014-86-K-0466, N00014-87-K-0510, and the Sherman Fairchild Foundation

1 INTRODUCTION

This paper is an informal exploration of methods for implementing distributed simulators as described in [Chandy and Misra 1979, Chandy and Misra 1981, and Jefferson 1985]. We attempt to explore concepts that appear helpful in thinking about the problem. Our intent is to understand the concepts at an intuitive level rather than to propose a mathematical theory or to describe an implementation. In this sense, the paper describes "work in progress" or "ideas in gestation". Section 2 describes "conditional events" – the basis for sequential simulation. Then "unconditional events" – a basis for distributed simulation is discussed. Two examples are used to illustrate these concepts: the first is an ultra-simplistic model of a battlefield, and the second is a model of a job shop or assembly line [Misra 1986]. The battle field example is used to suggest that the unconditional-event paradigm of distributed simulation may have limitations.

The concept of "knowledge" in distributed simulation is introduced in Section 3. The definition employed here is different from the one employed in the literature on distributed systems [Chandy and Misra 1986, Halpern and Moses 1984]. The concept is extended to "conditional knowledge" in Section 4. The concepts of conditional and unconditional knowledge form the framework which is employed to propose methods of implementing simulations on parallel machines in Section 5. Section 6 is a conclusion in which we use the framework to propose desirable properties of distributed simulators.

2 CONDITIONAL EVENTS AND SEQUENTIAL SIMULATION

Consider a sequential simulation of a network of processes. The computation to be simulated consists of a sequence of "events", where an event is a change in (global) system state. Each event is "initiated" by one process, in the sense that the precondition for the event is determined by the state of precisely one process. However, the occurrence of the event may change the states of an arbitrary number of processes.

A battle between two submarines, for example, can be thought of as a network of two processes where each "sub" is a process. Neither sub knows where the other one is, but it may guess the other's location from information such as sonar readings. Firing a missile, or changing velocity, are examples of events. Whether a missile is fired by a sub, depends only on the state of the sub, and does not depend on the state of the total system. In particular, whether a sub fires a missile, depends on the information it has obtained from its sensors about the location of the other sub, but it does not depend on the actual location of the other sub. The occurrence of an event may change the state of the entire system. Thus, the simulation model allows the firing of a missile by one sub to destroy the other sub instantaneously. (It can be argued that the simulation model is too powerful because missiles take non-zero time to travel. A more general argument is that events cause only "local" as opposed to global state changes. However, an event that is initiated at a single process, but which may change the states of all processes, is a useful concept in building models that are, after all, only approximations of reality In any case it is not necessary to use all the power of the simulator.)

The central data structures in a sequential simulation are the event list and a variable, "clock", which is a non-negative integer repre-

senting time. The event list contains one entry for each process. An entry e is a tuple (e.name, $e.T$) where e.name identifies the type of the event, and $e.T$ is a non-negative integer representing time, where $e.T >$ clock. The meaning of an entry (e.name, $e.T$) corresponding to a process p is as follows:

For all t' where clock $<= t' < e.T$:
IF all other processes do not initiate events in the interval (clock, t')
THEN no event is initiated by p in the interval (clock, $t' + 1$)
and
IF all other processes do not initiate events in the interval (clock, $e.T$)
THEN p initiates event e.name at time $e.T$.

Note: the interval (clock, t') is the open interval, i.e., it is the interval after clock and before t'. The action taken by a process at time t may depend on actions of processes at times earlier than t, but it does not depend on actions of other processes at time t (where an action is either initiating a specific event or not initiating any event). It follows that the time at which the next event is initiated is the smallest value of $e.T$ over all events e in the event list.

Sequential simulation is a repetition of the following sequence of actions:

- Determine the entries e in the event list with the smallest $e.T$;
- execute these events, i.e., change the system state appropriately;
- recompute the entry in the event list corresponding to each process.

In a sequential simulation of the submarine battle, the event list consists of an entry (e.name, $e.T$) for each sub, where $e.T$ is the time at which the sub next initiates an event provided the other sub does not initiate an earlier event. The simulation proceeds by executing the event corresponding to the smaller of the $e.T$ values, and executing an event may change the states of both subs. The new entry, in the event list, for each sub is then recomputed. The cycle is then repeated.

Consider a direct implementation of the sequential simulation of the submarine battle on a parallel machine with two processors, with one processor for each sub. The first step is to determine which of the two processors has the smaller value of $e.T$. The processor with the smaller $e.T$ communicates the corresponding e.name to the other processor. Both processors determine their next states in parallel, and both processors recompute their new entries in the event list in parallel.

The structure of the computation is as follows:

- determine the minimum over all entries e in the event list of $e.T$,
- broadcast the name of the next event to all processors,
- and each processor determines its new state and event-list entry in parallel.

In general, if an event changes the states of most of the processes, then parallelism can be exploited in the computations of the next states of the processes. On the other hand, if an event changes the states of a very small fraction of the processes, then there is less scope for exploiting parallelism with this approach. Let us next consider the simulation of quite a different system to get a better understanding of the limits of directly mapping sequential simulation programs on parallel architectures.

A simple job-shop consists of a source of jobs followed by a sequence of queues, represent-

ing workstations, followed by a job sink. A diagrammatic representation, where W represents a workstation is given next.

$$\text{source} \rightarrow W \rightarrow W \rightarrow W \rightarrow \textit{sink}$$

Assume that each workstation processes the jobs it receives in the order received. In the simulation there are processes corresponding to the source, the sink and to each of the workstations. An event is a transmission of a job, from the source to a workstation, between workstations, or from a workstation to the sink. Obviously an event in the job-shop changes the state of precisely two processes. A direct implementation of the sequential simulation of a job shop on a parallel architecture is not likely to result in substantial speed-up, because there are at most two processors executing at any time. Even if we were simulating a job shop with a large number of workstations on a parallel architecture with one processor for each workstation, we would not expect to do better than double the speed of a sequential simulation.

The moral of this little discussion is the obvious one—whether a given method of distributed simulation works efficiently depends on the sytem being simulated.

Let us generalize the event list to have two kinds of entries: unconditional entries and conditional entries. The entries e, in the event list, described earlier, are conditional entries. An unconditional entry u, in the event list, is a tuple (u.name, $u.T$) where u.name identifies the event and $u.T$ is a time and $u.T >$ clock. The meaning of the entry is as follows: an unconditional entry (u.name, $u.T$) for a process p means that process p will not initiate an event in the interval (clock, $u.T$) and process p initiates event u.name at time $u.T$; furthermore, the actions taken by process p in the interval (clock, $u.T$] are INDEPENDENT of the actions taken by other processes in the interval. The critical difference between unconditional and conditonal entries is that in the case of a conditional entry (e.name, $e.T$) for a process p, the actions taken by p in the interval (clock, $e.T$) MAY DEPEND on the actions of other processes in the interval (clock, $e.T$). Of course, every unconditional entry may be treated as a conditional entry, but conditional entries cannot (in general) be treated as unconditional entries.

We introduce a special event called the null event, which does not change the state of the system. Thus if there is an unconditional entry (u.name, $u.T$) for a process p, in the event list, where u.name is the null event, then it means that process p does not execute any "real" event in the interval (clock, $u.T$], where (clock, $u.T$] is the interval that excludes clock, but includes $u.T$.

Consider the source in the job-shop example. The events it initiates correspond to the jobs that arrive at the shop. Presuming that job arrivals do not depend on the manner in which jobs are processed in the shop, the actions taken by the source do not depend on the actions of other processes. Therefore, all the entries in the event list corresponding to the source are unconditional events. Furthermore, the source may produce a stream of unconditional entries corresponding to the times at which a stream of jobs arrives at the shop. There is no reason for the execution of the simulator of the source to depend on the execution of the simulator of any other process (provided that there is enough memory to store the unconditional entries).

Now consider a workstation in the job-shop. Assume that with each job is a work-order that specifies the amount of processing time that the job requires at the workstation. If a workstation is processing a job at some time, then

we know the time at which the workstation will complete processing the job, and this time is independent of the actions taken by other processes. Therefore, all entries representing the movement of jobs are unconditional.

Are there any conditional entries in a sequential simulation of a job-shop? Yes: A process simulating an idle workstation posts a conditional entry (null, infinity) to indicate that IF the workstation does not receive any more jobs THEN the workstation will not output any more jobs.

One way of simulating a job-shop on a parallel machine is as follows. The event list contains a stream (a queue) of unconditional entries from each process. Each queue of entries in the event list corresponds to a queue of jobs for a workstation. The processor simulating a workstation removes the entry at the head of its input queue, and processes it, i.e., adds an entry to its output queue. For example, if the entry at the head of the input queue is (u.name, t) where u.name is a work-order that specifies that the job gets t' units of processing time at the workstation, and if the workstation is idle at time t, then the processor simulating the workstation puts (u'.name, $t + t'$) in its output queue, where u'.name describes the remaining work-order for that job. No synchronization is necessary. If at time t, the workstation has t'' units of processing to complete before it can get to the newly arrived job, then the processor outputs (u'.name,$t + t' + t''$).

This method is faster than executing the simulation on a sequential processor. Indeed, this simulation is ideal for a pipeline architecture with one stage in the pipeline simulating one process in the job-shop. Once the pipe is full, a great deal of concurrency is exploited.

Now let us attempt to use unconditional entries, exclusively, for the battlefield problem. How far can we predict the actions taken by one sub, that are independent of the actions taken by the other sub? One presumes that in a battlefield, one sub monitors the other continuously, and therefore the only prediction that we can make about one sub, given ONLY the state of that sub, is what actions it will take in the very next instant. In other words, an unconditional entry is limited to the form (u.name, $u.T$) where $u.T$ = (clock+1). To predict the behavior of one sub over a longer horizon, we need to have information about both subs.

The unconditional-entry approach, applied to the battlefield, results in the following type of computation: The event list consists of two entries, one unconditional entry per sub. Each of the entries is of the form (u.name, $u.T$) where $u.T$ = (clock +1). Each process computes its state at time clock+1, then the clock is incremented by 1, and each process determines its new unconditional entry (u.name, $u.T$), where unfortunately, $u.T$ = (clock+1). The resulting computation is a time-driven simulation: each step of the computation corresponds to incrementing the clock by 1. Simulating the battlefield, using unconditional entries alone, on a parallel processor may result in slower execution than discrete-event simulation on a sequential processor.

What have we learned from these examples? The approach of using conditional entries alone, appears appropriate for parallel discrete-event simulation of our example of the battlefield. The approach of using unconditional entries alone, appears appropriate for parallel simulation of our example of a job-shop. Time-driven simulation is inherently parallel, and indeed it may be the simulation method of choice because, in many systems, the state changes continuously over time rather than at discrete instants.

It seems apparent that a simulation method that combines conditional and unconditional entries is worth studying because it is likely to be efficient for a wider class of problems. Before exploring this possibility let us define "knowledge in distributed simulations", and use the concept to gain insight into the problem of distributed simulation.

3 KNOWLEDGE AND SIMULATION

Our goal, in this section, is to gain an understanding of what makes for efficient distributed simulations. However, to gain this understanding, we shall define some terms formally. For brevity, the system that is being simulated is referred to as "the system". The processes in the system are called physical processes (or PPs) to distinguish them from processes in the simulator. A system is a set of PPs and a set E of events. Set E includes the null event. A PP is a set S of process states, an initial state in that set, an initial event (which is an element of E), a next- state function f, and a next-event function g, where:

$$f :: S \times D \rightarrow S,$$

where D is the powerset of E,

$$g :: S \times D \rightarrow E.$$

The meaning of f is as follows. If the PP is in state s at some time t, and the set of events d occurs at time t, then the state of the PP, at time $t+1$ is $f(s,d)$. The meaning of g is as follows. If the PP is in state s at time t, and the set of events d occurs at time t, then the PP initiates event $g(s,d)$ at time $t+1$. Note that $g(s,d)$ may be the null event. A PP is in its initial state and initiates its initial event at time 0.

The PPs we have defined are deterministic: their future behaviors are functions of their histories. In the real world, PPs may be nondeterministic; for example a PP may toss a coin to determine whether to initiate one event or another. In a simulator, randomness is modeled by employing a pseudo-random number generator. Given the value of the seed of a pseudo-random number generator, the sequence of numbers generated is deterministic. Given the values of the seeds, each execution of a simulator is deterministic. In analogy, we define a PP in terms of a single behavior, though in reality it may exhibit many behaviors.

For example, a partial specification of a submarine in a simplistic video game is as follows.

$$S = \{\text{on}, \text{off}\}$$

$$E = \{\text{null0}, \text{fire0}, \text{null1}, \text{fire1}\}$$

where null0, fire0 means that sub0 initiates the null event, fires a missile respectively; null1, fire1 are analogous events for sub1.

$$f(\text{off},d) = \text{off for all } d$$

$$f(\text{on}, \{\text{null0}, \text{null1}\}) = \text{on}$$

$$f(\text{on}, \{\text{null0}, \text{fire1}\}) = \text{off}$$

$$f(\text{on}, \{\text{fire0}, \text{null1}\}) = \text{on}$$

$$f(\text{on}, \{\text{fire0}, \text{fire1}\}) = \text{on}$$

.

Thus the submarine we are defining has the property that once it is "off" it remains "off", and if it is "on" it remains "on" except for the case that it does not fire and the other sub does. (Presumably, if both subs fire simultaneously, the missiles destroy each other without damaging the subs, which seems to occur quite often in video games.)

A system state is a tuple; its components are process states. The behavior of the system

is an array $B[0 \ldots N]$, where N is the time-horizon of the simulation; the elements of the array are pairs (r, d) where r is a system state, and d is a set of events. The meaning of a behavior is as follows. If $B[j]$ is (r, d) then the state of the system at time j is r, and the set of events that occur at time j is d. At time 0, each process is in its initial state and initiates its initial event. Since the PPs are deterministic, the system admits only one behavior. We leave it to the reader to develop an algorithm to compute the system behavior B given the specifications of the PPs; the obvious solution is to compute $B[j+1]$ after computing $B[0 \ldots j]$. This solution corresponds to a time-driven simulation.

Our problem is to design a simulator to determine the system behavior rapidly. There are no restrictions on our design. For instance we permit our programs to fill in several elements of array B in parallel, or to start with filling in the middle element.

A simulation is an execution of the simulator. A simulation is a sequence of computational steps and we do not interpret the meaning of a computational step. A simulation may be the sequence of computational steps carried out on a sequential machine, or an interleaving of computational tional steps carried out on a distributed machine or the sequence of synchronous operations carried out by a parallel synchronous machine. A point in the simulation is an integer j, which represents an instant in the unfolding of the simulation in which the first j steps of the simulation have been executed and the remaining steps have not been executed.

A simulator consists of one or more processes, called simulator processes (or SPs) to distinguish them from physical processes (or PPs) in the system that is being simulated. There need be no correspondence between SPs and PPs. In a sequential simulation there may be only one SP. In a two processor machine there may be two SPs. We place no restrictions on SPs because our goal is to obtain a general understanding of all simulators, rather than to understand a particular implementation of a distributed simulator. (Note: in most implementations there is one SP corresponding to each PP, but there is no reason to restrict our design to have a correspondence between processes in the system that is being simulated and processes in the simulator.)

Let b be a predicate on the system behavior. For example, b may be: $B[5]$ is (r, d) where d includes the event "fire1", and the state of PP 0 in r is "on".

Define a relation "knows" between SPs, predicates on system behavior and points in the computation, as follows:

q knows b at j, where q is an SP, b is a predicate on the behavior, and j is a point in the simulation, means that b can be deduced given only the state of q at j, and the specification for q.

EXAMPLE: Consider the example of the time-driven simulation of the submarine battle. Assume that there is one SP for each sub. An SP has the specification of one sub, but has no information about the other. If the clock has value t, and an SP has obtained the event that is executed by the other sub at time t, then by employing its function f its state at time $t+1$ can be determined, and by employing its function g the event it initiates at time $t+1$ can be determined. Therefore the SP knows its state at time $t+1$ and it knows the event it initiates at time $t+1$. Unfortunately, the SP does not know anything about the behavior at times after $t+1$ because that depends on the specification of the other sub. To gain knowledge about times later than $t+1$ the SP must receive information from the other SP.

Therefore, the concept of knowledge gives us an idea of the amount of information transfer required in this case. Informally speaking, in this example a great deal of information transfer is required because a process has limited knowledge.

EXAMPLE: Next consider the job-shop example. Assume that there is one SP for each workstation and each SP has the specification of the corresponding workstation, but does not have the specification of any other workstation. Assume that at some point in the simulation an SP simulating a workstation has the following information: a job requiring 10 units of service arrives at the empty workstation at time 5, and a job requiring 20 units of service arrives at the workstation at time 6. At this point in the simulation the SP knows that the workstation outputs no jobs in the interval (5,15), it outputs one job at 15 and the following one at 35. Thus the SP can carry out this computation without receiving information from other SPs.

EXAMPLE: Consider the example of the battle again, except that one SP has the specification for both subs. In this case, the SP knows the entire system behavior initially. The SP needs no additional information. Thus we draw a distinction between what an SP knows and what it has computed.

In designing efficient parallel simulators, we often tradeoff what an SP knows with the amount of computation it does. In a sequential simulation, the single SP knows the entire behavior initially, and an execution of the simulator consists of the SP computing what it knows. In a time-driven simulation, with one SP per PP, an SP only knows the behavior at the next value of the clock. Again speaking very informally, in an efficent simulation, by the time an SP computes what it knows, additional information arrives to increase its knowledge. If an SP's knowledge stays ahead of what it has computed then the SP need not become idle waiting for information. Some of the tuning that is done in making distributed systems more efficient—for instance in "packaging" the simulation of several PPs on the same processor—consists of specifying SPs so that an SP's knowledge stays ahead of its computation. This form of packaging SPs has a susbstantial impact on the performance of a distributed simulator.

Next we shall extend our definition of knowledge to "conditional knowledge" and then attempt to use our insight to design distributed simulators.

4 CONDITIONAL KNOWLEDGE

Consider the example of the submarine battle once again. Suppose the specification of sub0 is that it will fire its first missile at time 10 if it is not destroyed earlier. Suppose there is an SP that has the specification of sub0, and has no other information. What does the SP know when the simulation is initiated? According to our definition of "knows", when the simulation is initiated, the SP knows the state of sub0 at time 0 and the event initiated by sub0 at time 0. The SP does not know the state of sub0 at time 1 because it does not know the event initiated by the other sub at time 0. In particular the SP does not know whether sub0 fires it first missile at time 10. What else does the SP know initially? The SP knows

"sub0 is not destroyed in the interval (0,10)
implies
sub0 fires its first missile at time 10".

We define a ternary relation "conditionally-knows" between an SP, two predicates on the

system behavior and a point in the simulation:

q conditionally-knows b given b' at j where q is an SP, b and b' are predicates on the system behavior, and j is a point in the simulation, means

q knows $(b' \Rightarrow b)$ at j. In our submarine example, the SP conditionally-knows "sub0 fires its first missile at time 10" given "sub0 is not destroyed in the interval (0, 10)" at 0, i.e., at initiation of the simulator. The concept of conditional knowledge is not a new concept at all; but we dignify knowledge of predicates of the form "b' implies b" with a special name, because this form of knowledge plays an important role in simulation.

Conditional knowledge may not appear useful in itself. However, as a simulation proceeds, conditional knowledge can, sometimes, be converted to knowledge. How can conditional knowledge be converted to knowledge? Obviously, if q conditionally-knows b given b' at j, and q knows b' at j, then q knows b at j. Informally speaking, conditonal knowledge and some knowledge can yield more knowledge. Also conditional knowledge and more conditional knowledge can yield knowledge. For example, if an SP conditionally-knows that sub0 fires its first missile at time 10 given that it is not destroyed earlier, and the SP also conditionally- knows that sub1 fires its first missile at time 20 given that it is not destroyed earlier, then the SP also knows that sub0 fires its first missile at time 10 (assuming that a sub is destroyed only if the other fires). Indeed, this conversion of conditional knowledge to knowledge forms the basis for sequential simulation.

Now let us use our insight to explore some methods of distributed simulation.

5 EXPLORING METHODS FOR DISTRIBUTED SIMULATION

Our goal here is to suggest that the concepts of knowledge and conditional knowledge may be useful in the design of distributed simulations. We do not imply that the algorithms proposed here are better than others or even that they are original. Another goal here, is to use the concept of knowledge, coupled with that of computation, to get a VERY ROUGH idea of the efficiency of an approach without doing a great deal of empirical testing.

5.1 Each SP starts with a different initial state

Consider a simulator in which there are two SPs where $SP0$ has the specification for all PPs, and $SP1$ has the specification for all PPs, except that in place of the initial states and events, $SP1$ has some other values—let's call these values X. $SP1$ carries out a simulation starting with initial values X. Suppose $SP1$ has computed a sequence of (states, event-sets) tuples—$(r0, d0), (r1, d1), (r2, d2)$—where (r0,d0) corresponds to X. Now $SP1$ conditionally-knows $B[j+1..j+2] = (r1, d1), (r2, d2)$ given that $B[j] = (r0, d0)$, where B is the system behavior. In other words, $SP1$ conditionally-knows the behavior at $j+1$ and $j+2$ given that the behavior at j is the initial value employed by $SP1$. Here $SP0$ knows the entire behavior—it needs no information from $SP1$. The following approach to distributed simulation can be taken. Both SP's compute in parallel. If $SP0$ has computed a j such that $B[j] = (r0, d0)$, then it sends the value of j to $SP1$. Upon receiving this value, conditional knowledge in $SP1$ is converted to knowledge, and this knowledge has already been com-

puted by $SP1$.

This example illustrates that it may be efficient for an SP to compute conditional-knowledge because the values computed may turn out to be knowledge.

5.2 Messages with conditional and unconditional information

Consider a closed loop of two workstations. The output of each station feeds the input of the other. Each station has a single server. There are a fixed number of jobs in the system and each job cycles repeatedly through the two work- stations. There are two priorities of jobs. A job may switch priority when it completes service at a station. Whether a job switches priority is determined by the workstation at which it completes service.

The simulator has two SPs, one for each workstation. Suppose an SP knows that at time t a workstation is serving a high-priority job that requires t' additional units of service; then the SP knows that the workstation will output the job at time $t + t'$.

Now suppose that an SP knows that at time t a workstation is serving a low-priority job that requires t' additional units of service, and no high-priority job arrives at the station at time t. Then, if $t' > 1$ and all jobs require at least one unit of service, then the SP knows that the work- station does not output a job at time $t + 1$; this is because it cannot output the low-priority job that it is serving at least until $t + t'$, and if a high-priority job arrives at time $t + 1$, it cannot output that for at least one additional time unit. The SP also conditionally-knows that the next job out- put by the workstation is at time $t + t'$ given that no high- priority job arrives at the workstation in the interval $(t, t + t')$.

Let us design the simulator so that the SPs exchange information about both knowledge and conditonal knowledge. For example assume that $SP0$, $SP1$ are simulating workstations $W0$, $W1$ respectively. Assume that $SP1$ conditionally-knows that $W1$ outputs no high-priority job in the interval (t, t') given that $W0$ outputs no high-priority job in that interval. Also assume that $SP0$ sends a message to $SP1$ after receiving which, $SP1$ conditionally-knows $W0$ outputs no job in $(t+1, t+t')$ given $W1$ outputs no high-priority job in that interval. Upon receipt of the message, $SP1$ knows that $W1$ outputs no high-priority job in the interval (t, t').

In this example, the exchange of both conditional and unconditional information is helpful in allowing the simulation to progress.

This approach can be extended to an arbitrary network. In the worst case, this approach reduces to a sequential simulation. Therefore, this approach has the advantage of significant speed-up on parallel machines in some applications and of being no worse than sequential simulation for all applications.

5.3 Deadlock

It is interesting to interpret deadlock, from the point of view of knowledge. A process waits when it has computed everything it knows. All processes wait—i.e., remain deadlocked—if each process has computed everything it knows. "Breaking deadlock" means that some external process supplies information to the set of deadlocked processes that results in at least one of them gaining knowledge, and the process then continues computation. The [Chandy, Misra 1981] paper can be interpreted in this way.

5.4 Optimistic Computations

Many ways of implementing optimistic computations [Jefferson 1985] become apparent when we try to understand distributed simulation from the point of view of conditional knowledge.

A process that has computed everything it knows may wait to receive additional information, or it may proceed to compute conditional knowledge. A pessimistic computation is one where a process only computes what it knows. An optimistic computation is one in which a process may also compute what it conditionally-knows. (Pessimistic computations may also use conditional knowledge, because—as described in Section 5.2—conditional knowledge may be converted to knowledge.) The advantage of optimistic computations is that an SP never waits.

Consider a simulator in which all conditional knowledge that is computed is clearly identified as conditonal knowledge (as distinguished from knowledge). For example if a simulator of a job-shop determines that a work- station W outputs a job at time t given that it receives no high-priority job in the interval (x, t'), then this result is stored in memory exactly in the form of conditional knowledge, i.e., in the form b given b' where b is "W out- puts a job at time t" and b' is "W receives no high- priority job in the interval (x, t)". Now suppose that as the simulation unfolds, it is determined that W receives a high-priority job in the interval (x, t). Does the conditional knowledge have to be discarded?

There is no reason to discard conditional knowledge except the practical reason that there is insufficient memory to store the conditional knowledge that has been computed. It is not as though conditional knowledge becomes "false" and therefore must be erased to restore the system to a correct state. In our example, it is always the case that W outputs a job at time t given that it receives no high-priority job in the interval (x, t), even if W does receive a high-priority job in the interval.

What are the disadvantages of optimistic computations? In many simulators, an SP does not record everything it knows. For instance, an SP may not recored the entire state of a workstation because it may not be necessary to determine the entire behavior—it may be sufficient to determine some attributes of the behavior, such as "what is the average time spent by jobs in the job-shop"? When an SP starts computing conditional knowledge, it must distinguish conditional knowledge from knowledge; in particular, it may have to save the entire state of the processes that it is simulating, because this state is knowledge rather than conditional knowledge. Saving the state (and then copying the saved state) may be a considerable overhead. The extent of the overhead depends on the system being simulated. The overhead may outweigh the advantages of not waiting.

5.5 Processing Conditional Knowledge

Conditional knowledge can be processed in many ways. Consider the following example. The system to be simulated consists of three processes $u.i$, $i = 0, 1, 2$, connected in a cycle, where the events initiated by process $u.i$ may modify the states of processes $u.i$ and $u.(i + 1) mod 3$. There are three SPs: $SP.i$, $i = 0, 1, 2$, corresponding to the $u.i$. Suppose $SP.1$ receives a message from $SP.0$ that the first event $u.0$ initiates is at time 10 given that $u.2$ initiates no event before time 8. Suppose

that initially $SP.1$ conditionally-knows $u.1$ initiates its first event at time 20 given $u.0$ initiates no event before time 5. Upon receipt of the message from $SP.0$, $SP.1$ conditionally-knows $u.1$ initiates its first event at time 20 given $u.2$ initiates no event before time 5 (because if $u.2$ initiates no event before time 5 then—from the message from $SP.0$—$u.0$ initiates no event before time 5 in which case—from $SP.1$'s initial conditional knowledge—$u.1$ initates its first message at time 20). $SP.1$ sends this conditional knowledge to $SP.2$.

Now suppose that initially $SP.2$ conditionally-knows that $u.2$ initiates its first event at time 12 given $u.1$ initiates no event before time 9. Upon receiving the message from $SP.1$ that $u.1$ initiates its first event at time 20 given $u.2$ initiates no event before time 5, $SP.2$ gains the knowledge that $u.2$ initiates its first event at time 12.

This little example illustrates that there are many ways of processing conditional knowledge.

6 CONCLUSION

The goal of this paper is to suggest that we may have been unnecessarily restrictive in the way we think about distributed simulation. We attempted to develop a framework in terms of knowledge to free ourselves from unnecessarily restrictive modes of thought. As a consequence, different methods of distributed simulation suggested themselves. Of course, the methods could have been thought of without the aid of the concept of knowledge. We hope, however, that the concept is useful in designing simulators.

In our opinion the success of distributed simulation depends, in large part, on our ability to exploit properties of the system that is being simulated. For example using only conditional knowledge (as in sequential simulation), or using only unconditional knowledge (as in some methods of distributed simulation), does not exploit properties of the system to the fullest. We think that there is a better chance of achieving efficiencies on parallel processors by employing both forms of knowledge. Indeed, there may be other types of knowledge that can be exploited for specific applications.

A problem that has been receiving a lot of attention lately is that of designing "general purpose" distributed discrete-event simulators in which the processes in the system to be simulated are treated as "black boxes". In our opinion, the efficiency of such simulators depends on the interfaces to the black boxes. If a black box merely specifies what the corresponding process does during the next incremental interval of time, then the simulation reduces to a time-driven simulation. If a black box only specifies unconditional knowledge then the resulting simulator may be inefficient, as suggested in our battlefield example. If a black box only specifies conditional knowledge, then the resulting simulator may be inefficient, as suggested by our job-shop example.

The problem is to trade-off the "general purpose" nature of the simulator on the one hand with efficiency. If we insist that a general-purpose simulator is one that has very little information about the system being simulated, then we may have to pay the price in terms of efficiency. On the other hand, we cannot afford to propose a new method for simulation for each new application. The concepts of knowledge and conditional knowledge may help in defining an interface that does not expose too much of what is in the black boxes, and also results in simulators that are efficient.

This paper is extremely informal. A great deal of work remains to be done to define the concepts proposed here formally, to explore new concepts that will help further in freeing us from narrow modes of thought about simulation, in specifying the interfaces to the black boxes of general- purpose simulators, and in implementing the ideas on parallel machines.

Acknowledgement: We are grateful to Rivi Sherman for suggesting a correction in our definition of knowledge.

REFERENCES

Chandy, K. M. and J. Misra. 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 J. Misra. 1981. "Asynchronous Distributed Simulation ia a Sequence of Parallel Computations." *CACM 24*, no. 4 (Apr.): 198-205.

Chandy, K. M. and J. Misra. 1986. "How Processes Learn." *Distributed Computing*, no. 1 (1986): 40-52.

Halpern, J. Y. and Y. Moses. 1984. "Knowledge and Common Knowledge in a Distributed Environment." *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, (Vancouver, Canada, Aug. 1984).

Jefferson, D. R. 1985. "Virtual Time." *ACM Trans. Prog. Lang. Syst. 7*, 3(July): 404-425.

Misra, J. 1986. "Distributed Discrete-Event Simulation." *Computing Surveys 18*, no. 1 (March 1986): 39-64.