A RAND NOTE

FAST CONCURRENT SIMULATION USING THE TIME WARP MECHANISM, PART I: LOCAL CONTROL

David Jefferson, Henry Sowizral

December 1982

N-1906-AF

Prepared for The United States Air Force

Rand
SANTA MONICA, CA. 90406

The research reported here was sponsored by the Directorate of Operational Requirements, Deputy Chief of Staff/Research, Development, and Acquisition, Hq USAF, under Contract F49620-82-C-0018. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

Published by The Rand Corporation

A RAND NOTE

FAST CONCURRENT SIMULATION USING THE TIME WARP MECHANISM, PART I: LOCAL CONTROL

David Jefferson, Henry Sowizral

December 1982

N-1906-AF

Prepared for The United States Air Force

Rand
SANTA MONICA, CA. 90406

PREFACE

This Note describes the results of Rand research on distributed simulation conducted under the project "Computer Technology for Real-Time Battle Simulation," funded by Project AIR FORCE. The goal of this project is to improve the technology of computer simulation, particularly the understandability, modifiability, and speed of simulations. This Note examines a technique for accelerating a simulation's execution.

Computerized simulation is a critical tool in many military and aerospace applications, including weather forecasting, airfoil design, pilot training, strategic planning and other general scientific and engineering applications. It is a notoriously expensive tool, consuming huge amounts of computer time on powerful computers. The speed of almost all simulations can be dramatically increased by exploiting their inherent concurrency. Many processors, running in parallel, can cooperate in the execution of a single simulation and complete it in a fraction of the time that one processor would require. Because multiple processor systems, particularly local networks of many inexpensive processors, are becoming increasingly available, this approach should be practical in the near future.

David Jefferson, a consultant to The Rand Corporation, is an assistant professor of computer science at the University of Southern California.

SUMMARY

More than 20 years after its development, computer simulation continues to be a time-consuming process. It is common for a single simulation run to take many hours of elapsed execution time even on very powerful computers. Worse, a simulation may often require many runs to adequately sample and explore the simulated system's behavior. Despite the time and associated expense, simulation is the only tool available for attacking some complex problems in scientific, engineering, and military domains. Clearly, any generally applicable method for speeding up computer simulation would be a valuable contribution.

In this Note, the first of two, we address the problem of speeding up simulation through concurrency. Many processors running in parallel can cooperate in the execution of a single simulation to finish it in a fraction of the time required by one processor. Like all other researchers in the field of concurrent simulation we adopt an object-oriented view, where each real object in the physical system being modelled is represented by a logical object in the simulation. Dynamic interactions between real objects are represented by the exchange of time-stamped event messages in the simulation.

Several software mechanisms for concurrent, object-oriented simulation have been proposed in the literature. The best known are those invented by Peacock, Wong, and Manning (1979) at the University of Waterloo (Ontario) and by Chandy and Misra (1981) at the University of Texas at Austin. Their mechanisms all fall within what we call the Network Paradigm, and they have shown some empirical success in speeding

up small, simply structured simulations. Our analysis indicates, however, that all of their mechanisms should be disappointing on large, realistic simulations. We believe they will inevitably tend to overflow memory, or to be swamped by management overhead, or to achieve only very low concurrency.

We propose instead a radically new method for concurrent simulation called the Time Warp mechanism. Here we describe only the local control component of the Time Warp mechanism, the part concerned with the actual mechanics of discrete event simulation. In a future Note we will present the global control component, the part concerned with smooth and stable performance on a resource constrained distributed environment.

The Time Warp mechanism has a number of attractive properties.

- o It is completely transparent to the simulation programmer, in the same way that a virtual memory system is transparent to an applications programmer. The simulation programmer need not know that his program is running under the Time Warp mechanism. He need not provide any extra declarations, advice, or information. He uses the same conceptual methodology for building a concurrent simulation as he does for sequential simulation.
- o It can simulate any discrete model. Restrictions, such as a fixed communication topology, monotonically increasing message times, and nonzero minimum service times, usually necessary in the Network Paradigm, are unnecessary with the Time Warp mechanism.

o It can be profitably implemented on any MIMD architecture. It depends neither on the number of processors available nor on the interconnection architecture (shared memory, network, etc.). It does not even assume that the communication medium preserves message order. It is completely symmetric and distributed.

o It cannot deadlock, and it is guaranteed to progress forward in simulation time.

No other known concurrent simulation method combines all of these properties. In addition, we believe that the Time Warp method has greater speedup potential than any of the other concurrent simulation methods, although this remains to be demonstrated empirically. These advantages are not without cost, however. The Time Warp mechanism can be expected to use several times as much memory as other methods to achieve its speedup.

The Time Warp mechanism is an asynchronous simulation method. This means that some objects are allowed to progress ahead in simulation time while others lag behind. Its most unusual feature is the use of rollback and "antimessages" to automatically correct the temporal inconsistencies such "time spreads" generate. The Time Warp mechanism allows an object to simulate forward, in parallel with other objects, until it receives an event message that "should" have been handled in its simulated past. When this happens the object is automatically rolled back to an earlier simulation time. In addition to rolling back the primary object, however, the Time Warp mechanism must cancel all of the indirect side effects caused by any messages the primary object sent with time stamps greater than the time to which it rolled back. This is done by sending antimessages to annihilate the corresponding "ordinary" messages (thereby "unsending" the ordinary messages). The receipt of an antimessage can cause a secondary rollback. The Time Warp mechanism can be described as a distributed search over a simulation's "possible futures" to discover the true course of events.

A prototype implementation of Time Warps is running on a network of Xerox SIP 1100 (Dolphin) processors at Rand. A follow-up Note will describe this prototype system's performance.

ACKNOWLEDGMENTS

The authors would like thank Brad Allen, Jeff Barnett, Donald Cohen, David Dyer, Lee Erman, Monica Lam, Jack Mostow, John Postel, Bill Swartout and David Wile all from the Information Sciences Institute (ISI) for helping to create the algorithm for calculating GVT (to be given in Part II) and for elucidating the communication assumptions underlying it. They also thank Robert Balzer, Michael Fehling and Carl Sunshine (also of ISI) and Victor Lesser (U. Mass.) for offering suggestions leading to a better understanding of the global control mechanism. Others who have read the manuscript and given valuable comments include Orna Berry (USC), Jill Fain (CMU), Per Brinch Hansen (USC), Philip Klahr (Rand), Ross Quinlan (Rand) and Richard Swan (Megatest Corp.).

CONTENTS

FIGURES

I. INTRODUCTION

Simulations are among the most expensive of all computational tasks. One run often takes many hours of computer time to complete. Further, if the model being simulated is probabilistic, many runs may be necessary to determine the output distribution. In many scientific, engineering, and military situations, expensive computers are devoted almost exclusively to simulation, and simulation is a major bottleneck. Sometimes if a simulation takes too long it is useless, a classic example being a weather simulator that requires more than 24 hours to predict tomorrow's weather from today's data. In such cases it makes sense to try to speed up simulation by exploiting the parallelism inherent in most discrete models. If five processors running concurrently can reduce the weather simulation's elapsed run time from 24 hours to 10 hours (for example), then the increase in hardware cost for four extra processors may be justified.

This Note is Part I of a two-part report on recent results of research at Rand under the auspices of the Project on Computer Technology for Real-Time Battle Simulation (Klahr, McArthur, Narain and Best, 1982; McArthur and Klahr, 1982), commonly known as the ROSS Project (Rule Oriented Simulation System). The goals of the ROSS Project are to develop techniques to improve the understandability and performance of very large, object-based simulations. Part of the research effort has been concentrated on a new method, the Time Warp method, for doing concurrent or distributed simulation. Part I will first present the fundamental issues in concurrent discrete event simulation. A critical analysis of the methods for concurrent simulation proposed in the literature will follow. We then give a detailed description of the local control part of the Time Warp mechanism, the part concerned with the actual mechanics of discrete event simulation. The discussion examines the concepts of messages, antimessages, annihilation, and rollback.

Part II will present the global control component of the mechanism, the part concerned with smooth and stable performance in a concurrent or distributed environment. The central concepts are global virtual time and global control state. The global control mechanism deals with such issues as normal termination, error termination, storage management, name management, input/output, and message flow control, all in a simple, unified, and distributed manner. We currently have a running prototype implementation of the Time Warp system written in Interlisp on a network of five Xerox SIP 1100 (Dolphin) processors at Rand. A follow-up report will detail the system's performance on a number of different simulations as soon as the data are available.

The Time Warp method has several important advantages over other concurrent simulation methods described in the literature.

- o It is completely transparent. The simulation programmer does not need to know that his program is running under the Time Warp mechanism. He does not have to interact with it or provide any extra declarations, advice, or information to it. He uses the same conceptual methodology for building a concurrent object-based simulation as he uses for sequential object-based simulation.

- The Time Warp system can simulate any discrete model. Restrictions that are necessary for other concurrent methods, such as a fixed communication topology, monotonically increasing message times, or nonzero minimum service times, are not necessary for the Time Warp method.
- The Time Warp system can be profitably implemented on any multiple-processor architecture. The mechanism does not depend on the number of processors available, nor on the system architecture (shared memory, network, etc.). It does not even depend on the assumption that message order is preserved by the communication medium. It is completely symmetric and distributed.
- The Time Warp mechanism cannot deadlock, and it always progresses forward in simulation time.

No other known concurrent simulation method combines all of these properties. In addition, we expect that the Time Warp system will usually provide greater speedup than any of the other concurrent simulation methods. These advantages are not without cost, however. The Time Warp mechanism can be expected to use several times as much memory as other methods to achieve maximum speedup.

II. SURVEY OF THE PROBLEM

Discrete event simulation differs semantically from other computational paradigms primarily because of the notion of simulation time, which plays a special logical role in the global coordination of the computation. A simulation on one processor is normally organized as a collection of procedures, coroutines, or pseudo-parallel processes invoked according to a special scheduling discipline, lowest simulation time first. Here we assume the reader is familiar with the broad outlines of discrete event simulation. General discussion of simulation methodology and simulation languages can be found in Fishman (1978) and Franta (1977).

Many methods have been proposed in the literature for implementing concurrent or distributed simulation, some of which we will discuss later. They can be broadly classified into two groups, the synchronous and the asynchronous methods.

In a synchronous method all objects in the simulation progress forward in simulation time together, in synchrony, with no object ahead in time of any other. The usual event queue implementations for sequential simulation are all synchronous methods, and several generalizations of them for distributed simulation have been published. In contrast, an asynchronous method permits some objects to simulate ahead in time while others lag behind. Of course, an asynchronous method must include some mechanism for ensuring that when an object that is "behind" schedules an event for execution by an object that is "ahead" it does not cause any events to be executed in the wrong order.

But within that constraint, an asynchronous method tries to maximize the number of events being executed in parallel. We emphasize that the difference between synchronous and asynchronous methods is in the implementation. It is not a semantic difference, and it is therefore in principle invisible to the simulation programmer.

In Sowizral and Jefferson (1982) we show that synchronous methods are fundamentally unable to provide any speedup at all for many simulations, in particular the very common kind of simulation with low "virtual concurrency." We conclude that asynchronous methods are essential, and therefore we will discuss only asynchronous methods here.

Below we give an overview of the Network Paradigm, which underlies most previous work on concurrent simulation. Following that we present an introduction to the Time Warp method. In Sec. III we describe in much greater detail the methods based on the Network Paradigm along with our assessment of their strengths and weaknesses. Then in Sec. IV we give a detailed description of the Time Warp mechanism.

OVERVIEW OF THE NETWORK PARADIGM AND CONSERVATIVE MECHANISMS

For the last several years two main groups have been active in the field of asynchronous distributed simulation: Chandy and Misra at the University of Texas at Austin, and Peacock, Wong, and Manning at the University of Waterloo. In most of their work (Chandy and Misra, 1979, 1981; Peacock, Wong, and Manning, 1979a and b; Peacock, Manning, and Wong, 1980), they adopt what we will call the Network Paradigm of distributed simulation.

In the Network Paradigm, objects in the simulation are represented as deterministic sequential processes acting as nodes in a network. Directed arcs in the network represent communication channels between the objects. Each object has associated with it a variable called local simulation time that records the simulation time of the last event executed by that object and thus measures how far it has progressed in the simulation. Interactions between objects in the network (called events, customers, transactions, etc.) consist of time stamped event messages moving along the network's arcs. Each event message is tagged with a time stamp that specifies the simulation time for processing the event.

An object in the Network Paradigm may have several input arcs, and it maintains separate message queues for each one. An important assumption in the Network Paradigm is that the communication medium preserves the order of messages and that the time stamps of the messages sent along any particular arc must form a nondecreasing sequence. Hence, each node can easily merge its several input message streams. Note that this constitutes a restriction on the usual semantics of simulation. In the terminology of the Network Paradigm, the usual semantics would allow messages sent along an arc to be in any order, so long as the time stamp on each message is greater than or equal to the sender's local simulation time at the moment of sending.

Nodes in the Network Paradigm are processes that continuously execute the following program:

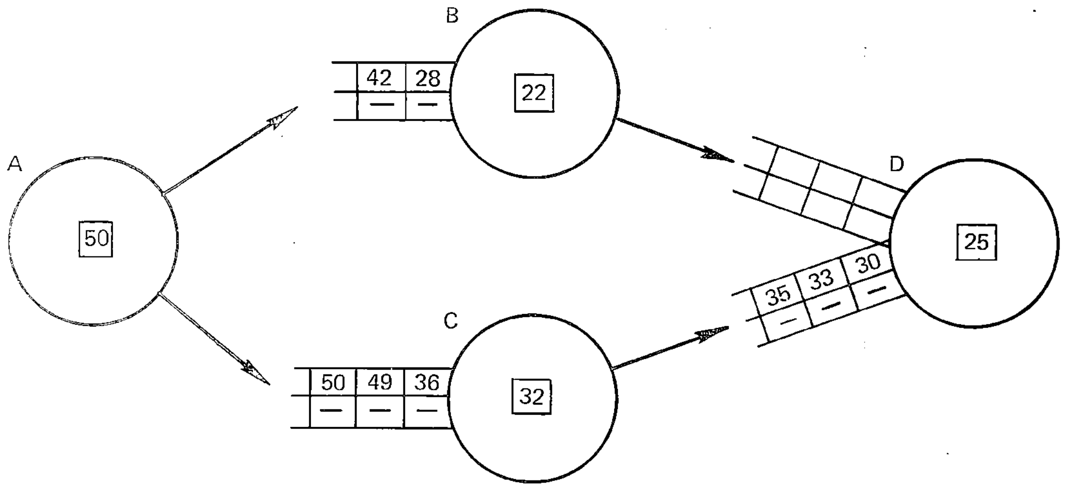
1. Receive (or wait for) the "next message across all input arcs."

2. Update the local simulation time for this node to be the value in the message's time stamp.
3. Perform the actions appropriate to the message. This will typically involve changing the node's state and sending one or more time stamped messages along the output arcs of the node.
4. Go to step 1.

Ideally, if each of the n nodes in the simulation were assigned to a different processor then all could execute concurrently, thereby achieving an optimal n-fold speedup over the usual single processor case. But severe problems prevent such ideal behavior from happening in real simulations. First there is a limit to the amount of concurrency available for exploitation in any particular simulation. Aside from that, there are problems arising from the nature of concurrent or distributed computation.

The major problems stem from Step 1 and the definition of the phrase "next message." The intended meaning is "the message with the lowest time stamp that is now enqueued <u>or</u> <u>will</u> <u>ever</u> <u>arrive</u> along any of the node's input arcs." If every input arc has at least one unprocessed message enqueued, then the "next message" is simply the message with the lowest time stamp across all of the input queues. Because the queues are generally ordered by increasing time stamp, this requires examining only the first message in each queue. The requirement that messages arriving along each arc must form a nondecreasing sequence (and the assumption that message orders are preserved during transit) guarantees that no message can arrive later with an earlier time stamp than the one selected by this procedure.

However, it is frequently the case, and in many types of simulations it is usually the case, that a node will have one or more of its input queues empty when it tries to receive the "next message." When this happens the node must wait, because if it accepts any message from the nonempty input queues, it has no guarantee that later (in real time) it will not receive, along one of the empty input arcs, a message with a time stamp earlier than that of the message it did accept. Such an eventuality would constitute a logical error in the simulation because it would cause events to be simulated in the wrong time order. We illustrate the Network Paradigm in Fig. 1 where we see that object D



In this simulation snapshot we show, inside each object, the local time for that object. In the Network Paradigm, objects A, B and C are all eligible to execute concurrently, but object D must block because one of its input queues is empty. If D were now to process the message from C with time stamp 30, it would risk receiving a later message from B with a time stamp less than 30. Such an eventuality would cause some message(s) to be processed out of order.

Fig. 1--Conservative mechanism within the Network Paradigm must block, at least momentarily, because one of its input arcs has an empty message queue.

For concurrent simulation in the Network Paradigm (static communication graph, separate queues for each input arc, messages with increasing simulation times along each arc) we can define the following terms. An object is <u>safe</u> at a particular moment if it has at least one message queued on each input arc. An object with at least one empty input arc is <u>unsafe</u>. We can define a <u>conservative</u> <u>mechanism</u> for concurrent simulation as one in which at any given moment, all safe objects are considered eligible to execute, but unsafe objects are suspended until they are safe. Clearly any conservative mechanism with fair scheduling will be "weakly input/output equivalent" to the usual sequential event-list simulation mechanism because, assuming both terminate without memory overflow, runtime error or deadlock, each object, in the course of computation, will receive the same sequence of messages (events), will progress through the same sequence of states, will send out the same messages to other objects, and will produce the same final output.

The virtues of the conservative strategy can be illustrated in Fig. 2. In a simulation where the network has no fan-in or directed cycles, each process has only one input queue and is eligible to execute whenever it has at least one unprocessed event message in that queue. Deadlock is impossible in this special case, and because no unnecessary blocking occurs the degree of concurrency is maximal.

But a conservative mechanism by itself is unsatisfactory because for most simulations (other than simple ones such as in Fig. 2) they are

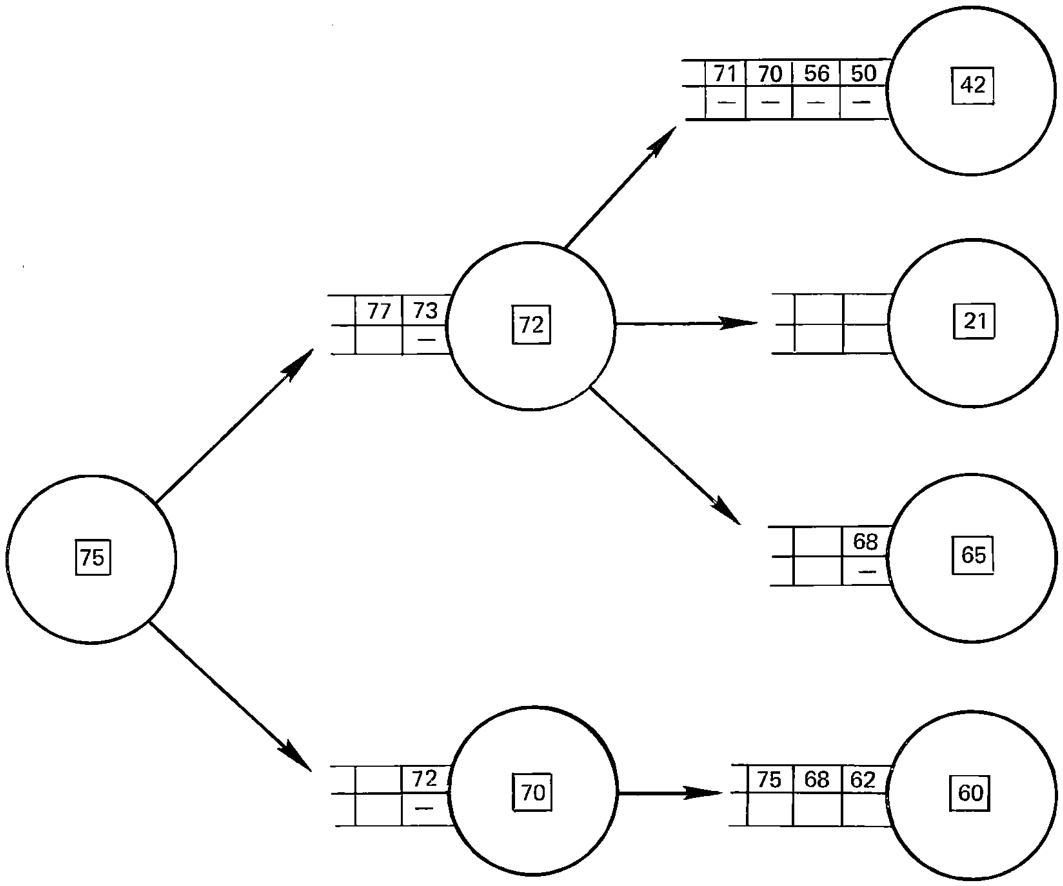
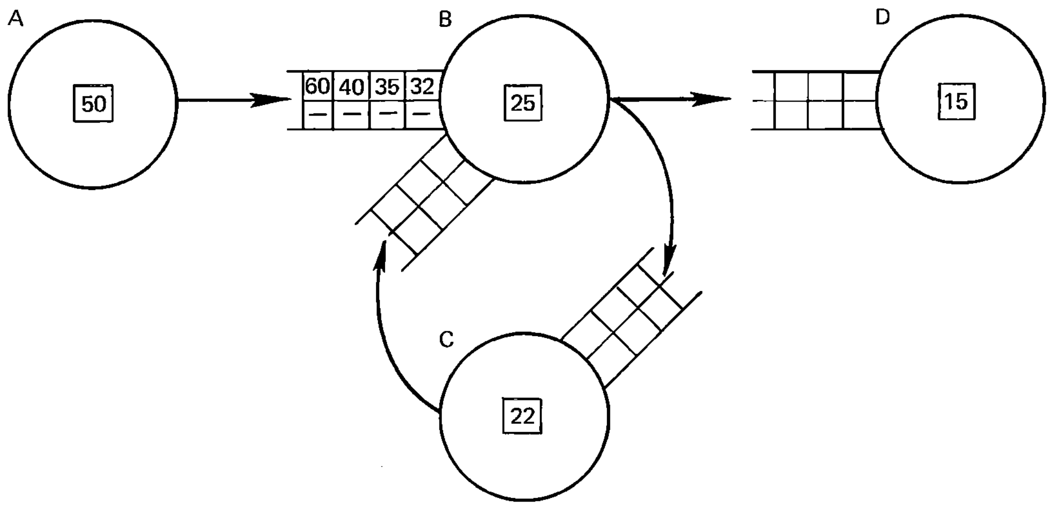


Fig. 2--Network where the conservative mechanism works well

hopelessly unstable computationally. They are generally subject to unpredictable memory overflow or deadlock, and even in the absence of these problems they allow only limited concurrency in most simulations.

Consider again the situation in Fig. 1. If the rate at which B and C together produce event messages exceeds the rate at which D consumes them then messages will back up in one or both of D's input queues, eventually overflowing memory. This, of course, is the flow control problem to which all distributed systems (even the one in Fig. 2) are subject. But in addition, there is a further problem peculiar to simulation under a conservative mechanism. If B produces event messages rarely compared with C, then most of the time D's input queue of messages from B will be empty and D will be blocked. Meanwhile messages from C can pile up and eventually overflow memory, even though D might be intrinsically fast enough to consume event messages at a rate higher than the sum of the production rates of B and C; the problem is that D spends most of its time blocked. In many simulations it is impossible to predict in advance whether this will happen, because it is influenced by variations in timing, by the details of process scheduling, and by the nondeterministic (probabilistic) nature of the simulation itself.

If the network contains a directed cycle (and nearly all large, interesting simulations do) then the conservative mechanism is vulnerable to deadlock. A momentary pause in the message stream along one or more arcs of the cycle may cause a local deadlock by "deadly embrace." In Fig. 3 we see an example where objects B, C, and D are permanently unsafe. (Here, and throughout most of the rest of this Note, we use the term "deadlock" to mean "unterminated process(es) permanently ineligible to execute." We do not require that there be any recognizable cyclic deadly embrace.) In fact, in this example, with the pure conservative mechanism, it is impossible for B to process even the first event message from A. Notice that this is a deadlock caused by the conservative mechanism; it is not a logical deadlock in the simulation model. The same simulation executed under the sequential



Here objects B, C, and D are permanently blocked, not having any possibility of a message arriving at their empty input queues. Only A can execute, piling up messages in B's input queue. The simulation will end in deadlock or memory overflow, depending on whether A terminates or blocks before filling up memory at B.

Fig. 3--Local deadlock

event list method would complete successfully.

These two problems, memory overflow and deadlock, taken together contribute to extreme computational instability in large simulations. On the one hand, if event messages arrive at a node faster than the node can process them, its message queues will certainly overflow memory (if the simulation runs long enough). On the other hand, if messages arrive from several sources and the total rate of arrival is lower than the node's theoretical capacity to process the messages, then most of the time one or more of the input message queues will be empty and so again either memory overflow (in the other queues) or deadlock is probable. The likelihood must be extremely remote that a large, complex simulation will succeed, at every node and around every network cycle, in walking the narrow path between these two dangers long enough to terminate normally.

Even if there were a way to avoid these problems there is still a feeling that the conservative mechanism does not extract enough of the available concurrency--that it is too conservative. If we examine Fig. 1 once again, we see that D is blocked until B sends another event message because it might send one whose time stamp is less than 30. If in fact B does do that, then D's waiting was necessary. But if after all that waiting B eventually sends a message with a large time stamp, say 50, then D's waiting was in vain. It could have safely processed its messages from C, resulting in increased concurrency as well as less risk of deadlock or memory overflow. We believe that in real simulations under a conservative mechanism blocking "in vain" is what would happen a large fraction of the time.

We have shown that for several reasons pure conservative mechanisms are unsuitable for concurrent simulation. But the Network Paradigm has nevertheless been the point of departure for much of the research in the field, and we shall go into greater depth in Sec. III where we survey several other proposed mechanisms. Before we do that, however, it is best to give an overview of the Time Warp mechanism to give the reader an appreciation of its contrast with the Network Paradigm.

OVERVIEW OF THE TIME WARP MECHANISM

The Time Warp method is an asynchronous method that speeds up simulations by automatically exploiting the concurrency that results from the programmer's decomposition of a model into interacting objects. Although it is directly applicable only to simulation, we believe that it is a particularly elegant example of several principles of distributed computation that can shed light on the design of systems for many other applications.

The Time Warp method applies to a larger class of simulations than does the Network Paradigm. There is no notion of a fixed communication graph connecting the simulation objects; rather, any object may interact with any other object at any time. Because no communication network is presumed, there is no need for an object to have multiple input queues. Each object has only one input message queue, and all incoming messages are funneled into it. There is no restriction corresponding to the requirement in the network model that messages be sent in nondecreasing order along each arc. For example we permit an object *A* at simulation time 100 to send a message with time stamp 200 to object *B*, and then later at simulation time 120 to send a message with time stamp 150 to the same object *B*. In all cases, of course, the time stamp on a message must be greater than or equal to the simulation time of the sending object at the moment of sending. This ensures that objects cannot schedule events in the past.

Throughout the rest of this Note we use the term *virtual time* in connection with the Time Warp mechanism to be synonymous with what is usually called *simulation time*. Using the new term emphasizes the more elastic nature of time introduced with the Time Warp mechanism and also is consistent with other computer science uses of the word "virtual."

Like the Network Paradigm mechanisms, the Time Warp mechanism is _asynchronous_, meaning that there is no global variable that represents the simulation clock; instead, each individual object contains its own variable called LVT (_Local_ _Virtual_ _Time_). An object's LVT acts as the simulation clock for that object. Because the Time Warp mechanism is asynchronous, in any particular snapshot of a simulation some objects will have LVT values greater than others.

The key feature of the Time Warp mechanism is that an object always receives and acts upon the messages in its input queue one by one in order of their time stamps _until_ _it_ _exhausts_ _the_ _queue_. In contrast with the Network Paradigm, a Time Warp object never waits until it can "safely" process the next message. It always charges ahead, blocking only when its input queue is exhausted, and then only until another message arrives.

This policy of always charging ahead risks the possibility that a message will arrive at an object whose LVT is _less_ _than_ the time stamp of the incoming message. In other words, it is possible for a message to arrive "in the past." Let us call such a message a _straggler_. It might appear that the possibility of straggler messages could cause event messages to be processed out of order--a serious problem as each message may cause both a state change in the receiving object and the sending of additional messages. If even one message were to be processed out of time stamp order, the results of the entire simulation might be invalid.

Because it is almost certain that there will be some stragglers in the course of a simulation, the Time Warp system provides a mechanism to ensure the simulation's integrity. Whenever a straggler arrives at an object's input queue, the Time Warp mechanism automatically restores that object to a state from a virtual time earlier than the time stamp of the straggler, cancels any side effects that it may have caused in other objects (possibly by rolling them back as well), and then starts simulating all affected objects forward again. We call this complex of actions _rolling back_ the object. A rolled back object may reprocess some messages that it processed before, but this time the straggler, whose late arrival caused the rollback in the first place, will be processed in its correct sequential position.

The mechanism used to roll back part of a simulation is the heart of the Time Warp mechanism. Although the idea of rolling back to an earlier state may seem hopelessly expensive and clumsy at first glance, the mechanism is in fact quite economical and even elegant. It rolls back only the objects that must be rolled back, it rolls them back only as far as necessary, and it rolls them back at the first moment the information mandating it becomes available. We expect that rolling back can be done quickly enough and rarely enough so that in large simulations the increased concurrency achieved will more than pay for the overhead of the rollback mechanism.

It is helpful to view the Time Warp mechanism as a game of chance. The object of the game is to minimize "losses"--real delay involved in wasted computation and rolling back. Every time an object processes a message _M_ with time stamp _t_ it makes a bet that no message with a time stamp earlier than time _t_ will arrive later. If it wins that bet no time is lost. If it loses the bet, the time lost is the delay involved in restoring the object to an earlier state and running it forward to the point when message _M_ can be processed. The success of the Time Warp system is based on the assumption that most simulation programs are "well-behaved" and that stragglers will arrive rarely enough in the long run to make the gamble worthwhile.

We might compare this to the game played by paging systems. Every time a running program makes a memory reference it takes a gamble that the page referenced will be resident in memory. When that gamble is won there is no time lost and the memory reference proceeds without delay. When it is lost, the cost is the delay needed for disk accesses and page table manipulation before the memory reference can proceed. The success of paging systems rests on the empirical fact that programs are reasonably well-behaved and that with good replacement strategies the gamble is won often enough to be worthwhile.

In Sec. III we describe and critique some of the distributed simulation methods using the Network Paradigm that have appeared in the literature. Readers interested primarily in the Time Warp mechanism may wish to skip directly to Sec. IV.

III. METHODS BASED ON THE NETWORK PARADIGM

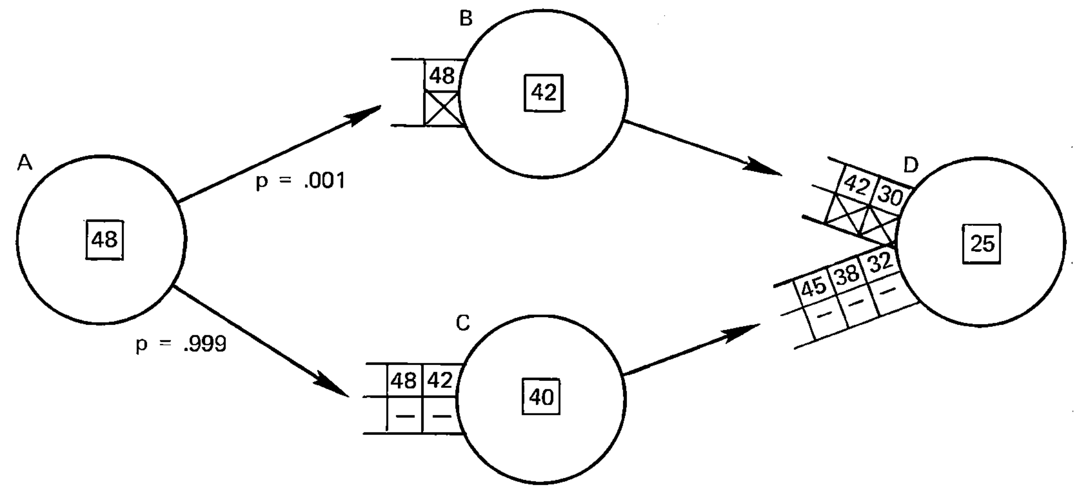
THE LINK TIME ALGORITHM

Peacock, Wong, and Manning (1979) describe their Link Time Algorithm as a mechanism in which the sending node for each arc (link) maintains a link time for that arc. A link time is a lower bound on the time stamp of the next event message to be sent along that arc and is periodically communicated to the receiving node, which can use the information to determine whether it is safe to execute even if the input queue for that arc is empty of event messages. Although the authors do not explain the details of how this is done, what they describe seems to be equivalent to the following mechanism (described by Chandy and Misra, 1979). Each node of the simulation automatically sends extra time stamped "null" messages along some or all of its output arcs whenever the local simulation time at that node changes. The null messages have no semantic content in the simulation and can be treated as "events" requiring no action. Like real transactions they play a part in the safe/unsafe decision, and processing them does cause the local simulation time of the receiving object to advance. They can be interpreted as "advice" from the sender that the time stamp on the null message is a lower bound for the time stamps of all future messages it will send.

Null messages have the positive effect of making the receiving node safe a greater fraction of the time than they otherwise would be, resulting in less unnecessary waiting and increased concurrency. In

Fig. 4 we see that process D has two null messages from process B in its queue. These messages ensure that D will be safe at least until it finishes processing the second message from C. Without these null messages D would be unsafe until the next real transaction arrives from B.

The Link Time technique can sometimes result in a greater degree of concurrency than the unembellished conservative mechanism. But we



Null messages are indicated by crossed diagonals. Object A sends 99.9 percent of its "real" event messages to C, but whenever it does so it sends a corresponding null message to B to advise it that no more "real" messages with earlier time stamps will arrive. B in turn passes these on to D. We see that D is now safe, and will continue to be safe at least until it finishes processing the message with time stamp 42. If by that time B has sent on the null message with time stamp 48, then D will still be safe and can process the real message with time stamp 45 without pausing.

Fig. 4--The Link Time algorithm using null messages

expect it to get bogged down frequently in excess message processing when the number of null messages begins to swamp the number of real messages. This will tend to happen in simulations where most of the "real" traffic travels on only a few of the arcs. It also tends to happen when the average number of output arcs per node is greater than one and there are lengthy (or infinite) paths in the communication graph, because then the total number of messages (event messages plus null messages) increases exponentially with time. Because distributed simulations can be expected to be communication-bound, rather than computation-bound, it seems desirable to avoid situations where the number of overhead messages can greatly exceed the number of real messages.

The most serious problem with the Link Time algorithm, however, is that it does not address the problems of memory overflow or deadlock. Peacock et al. do not discuss the memory overflow issue, but they do describe a deadlock prevention mechanism suggested for use with the Link Time algorithm. Their strategy is to require a nonzero minimum service time at each node in the network. This means that at each node there is some positive number $\underline{e}$ such that the node cannot both receive an event message with time stamp $\underline{t}$ and send one with a time stamp between $\underline{t}$ and $\underline{t}+\underline{e}$. The proof that this is sufficient to avoid deadlock is given in Peacock, Wong, and Manning (1979), where the authors also credit the result independently to Chandy and Holmes.

This requirement does avoid deadlock, but it constitutes a serious artificial constraint on the types of models that can be simulated. Many of the most frequently used service time distributions (exponential, Erlang, lognormal, etc.) do not have a nonzero lower bound. Queueing models using such distributions for service times either cannot be simulated or can be simulated only with distortion.

Furthermore, although deadlock is technically avoided, as a practical matter a "near-deadlock" problem is created that is almost as paralyzing. When the minimum service time is made small to minimize distortion in the model, the time and message traffic wasted in avoiding near-deadlock situations can be huge. Suppose we choose a minimum service time of 0.001, where the mean service time is 1. Then the situation shown in Fig. 5 might easily arise in a part of some larger simulation. In the upper section object _B_ would like to receive the "real" message with time stamp 34 from _A_ but it is blocked for lack of input from _C_. It has, however, sent null messages to _C_ and _D_ reflecting the minimum service time of 0.001. In the middle section those null messages have been received, and _C_ has sent a null message back to _B_. _B_ is not blocked now, but it cannot receive the "real" message with time stamp 34 from _A_ yet. It again sends null messages, again with time stamps 0.001 greater than its virtual time, as we see in the lower section. Clearly, null messages will flow around the _B_-_C_-_B_ cycle counting from 32.000 to 34.000 by increments of 0.001 until finally _B_ is eligible to receive the "real" message from _A_. Approximately 2000 "unnecessary" null messages will be sent and received _sequentially_, _with no concurrency at all_. Increasing the minimum service time will reduce the number of null messages but will also increase the distortion in the simulation. Both alternatives are undesirable.

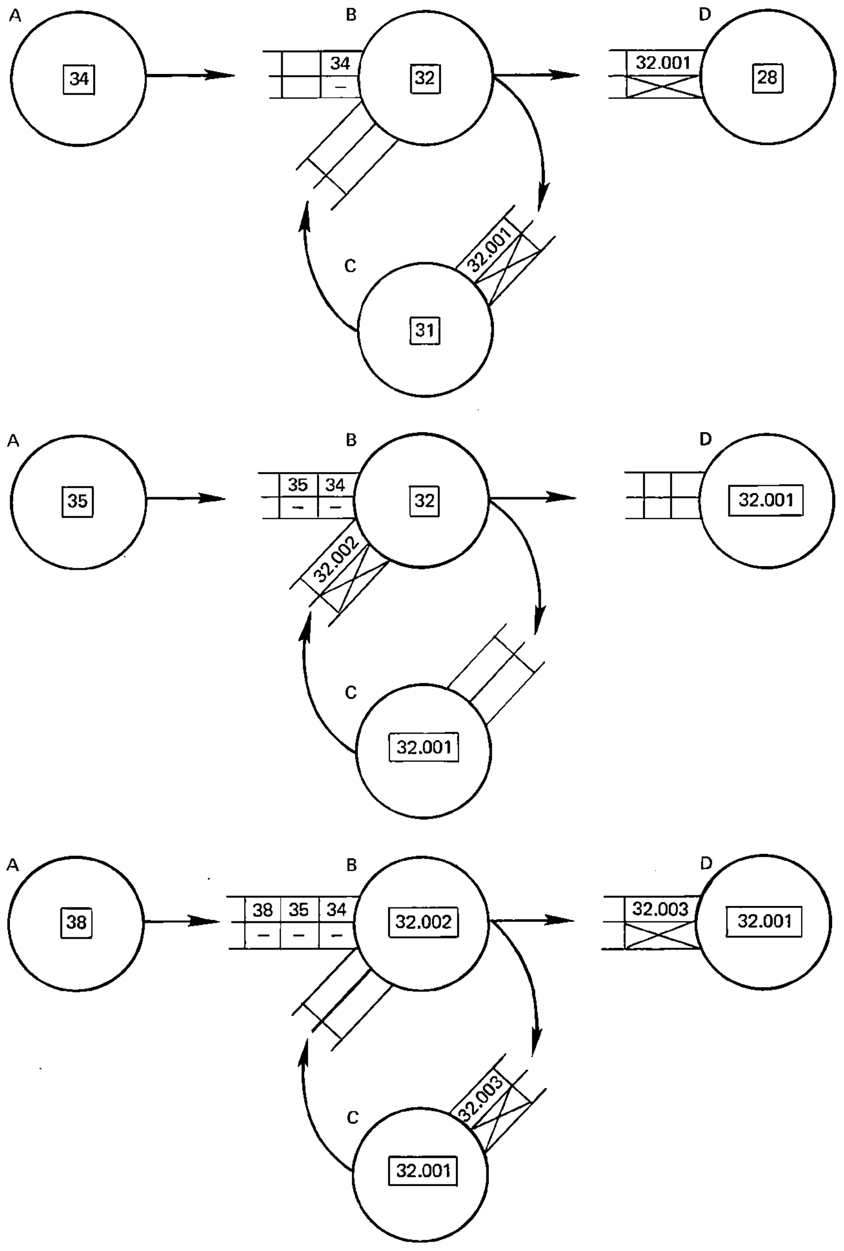


Fig. 5--Minimum service times used to avoid deadlock

THE BLOCKING TABLE ALGORITHM

One way to avoid deadlock and possibly increase the concurrency over that available from the Link Time algorithm (at least theoretically) is to use the Blocking Table Algorithm (Peacock, Wong, and Manning, 1979), which continually and incrementally computes an approximation to the transitive closure of that part of the communication graph formed by the empty arcs (arcs terminating in an empty queue). In this way each node knows not only which immediate predecessors are blocking it (because the queue associated with that arc is empty) but which of their predecessors are blocking them, etc. From each of the blockers the node requests a lower bound on the simulation time at which they will send out their next event message. This information at each node is then called its "blocking table." (See Peacock, Wong, and Manning, 1979, for details.) The blocking table theoretically allows a less localized definition of "safe" and "unsafe" nodes, so that the decision as to whether a node is eligible to execute is based on information global to the simulation, rather than on information from its immediate predecessors only (as in the Link Time Algorithm). But distributed computation of the transitive closure, even incrementally as in the Blocking Table algorithm, appears to be prohibitively expensive for large simulations because even a single event can both delete and insert new empty arcs into the subgraph of empty arcs. The time spent doing exact transitive closure would overwhelm the time spent actually simulating, which is why Peacock et al. substitute an algorithm such as the Blocking Table algorithm, which computes a faster approximation to the transitive closure.

Our judgment of the Blocking Table algorithm is that the number of messages exchanged while computing the blocking table for each blocked node would far exceed the number of messages exchanged during actual event simulation in most cases. The system would spend more time deciding who goes next in the simulation than it would actually simulating. The problem would seem to get worse in larger simulations, although without a more detailed description of the algorithm from the authors it is not possible to say how much worse. In the absence of empirical information to the contrary, the viability of the Blocking Table approach to deadlock avoidance has to be considered unproved.

THE CHANDY-MISRA METHOD

Chandy and Misra (1981) consider a mechanism, also within the Network Paradigm, that deals directly with both the deadlock and the memory overflow problems. First, they require that each message queue have a fixed finite length. In addition to blocking whenever one of its input arcs is empty, a node must also block whenever it sends a message along an output arc where the message queue is full. (The authors mostly discuss the case where queue lengths are fixed at zero, and thus every message transmission amounts to explicit synchronization between sender and receiver.) This technique clearly solves the memory overflow problem and in fact guarantees that the storage used by a concurrent simulation is within a constant factor of that used by the usual sequential event list mechanism, an important advantage.

However, this same mechanism exacerbates the deadlock problem. Because a node can block while either sending or receiving an event message, a deadly embrace can occur around any undirected cycle in the simulation network (a cyclic path defined without regard to the direction of the arcs). Thus, in a large simulation run under this mechanism there are innumerably more opportunities for deadlock than there are under an ordinary conservative mechanism. Furthermore, each deadlock rapidly becomes global. In a connected network when a local deadlock occurs around a cycle, all other nodes soon block either because they have at least one empty input arc or they fill up an input queue sending to a node that is blocked.

Chandy and Misra suggest that concurrent simulations always run in tandem with a distributed deadlock detection mechanism. The simulation would proceed in a cycle of alternating episodes of (a) concurrent simulation until detection of deadlock (using a deadlock detection algorithm based on a termination detection algorithm of Dijkstra's), followed by (b) breaking of the deadlock (using a special sequential process).

Chandy and Misra state that they do not expect the deadlock breaking process to be a sequential bottleneck, presumably because it is fast and rarely invoked. We are not sure. Because the use of fixed-length queues leaves the simulation so vulnerable to deadlock we believe that the simulation-until-deadlock phase will be highly unstable in any large, realistic simulation and that the simulation would hardly get up to speed before some local deadlock around a small undirected cycle would shut it down. It also appears to us that the mechanism used for breaking and detecting deadlock and restarting some of the processes must either perform a global analysis of the system that detects the maximum number of processes to restart or some faster analysis that detects fewer processes but leaves the system in a near deadlock state. Only empirical evidence will tell whether the overhead for deadlock detection and breaking will exceed the time saved by concurrency.

IV. THE TIME WARP MECHANISM IN DETAIL

It seems fair to say that the approaches to concurrent discrete event simulation that have been investigated so far, while showing some genuine empirical success in very small, cycle-free simulations (Peacock, Wong, and Manning, 1979; Chandy and Misra, 1979) leave much room for improvement in the dimensions of generality, deadlock avoidance, stability, and potential speedup in realistic simulations. The remainder of this Note is devoted to describing the Time Warp mechanism, a radical departure from these methods and one that we believe avoids the problems of the Network Paradigm in a clean, understandable, and elegant manner.

We introduce the Time Warp system by describing messages and objects and the special queueing discipline connecting them. We then illustrate the main simulation mechanism by following a typical object through a complete simulation cycle.

MESSAGES, QUEUES AND OBJECTS

A message in the Time Warp system has six components, as shown in Fig. 6.

1. The sender: the name of the object sending the message.
2. The receiver: the name of the receiving object.
3. The send time: the virtual time of the sending object when the message was sent, used to order messages in the output queue of the sending object.

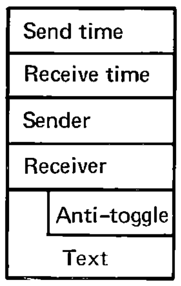
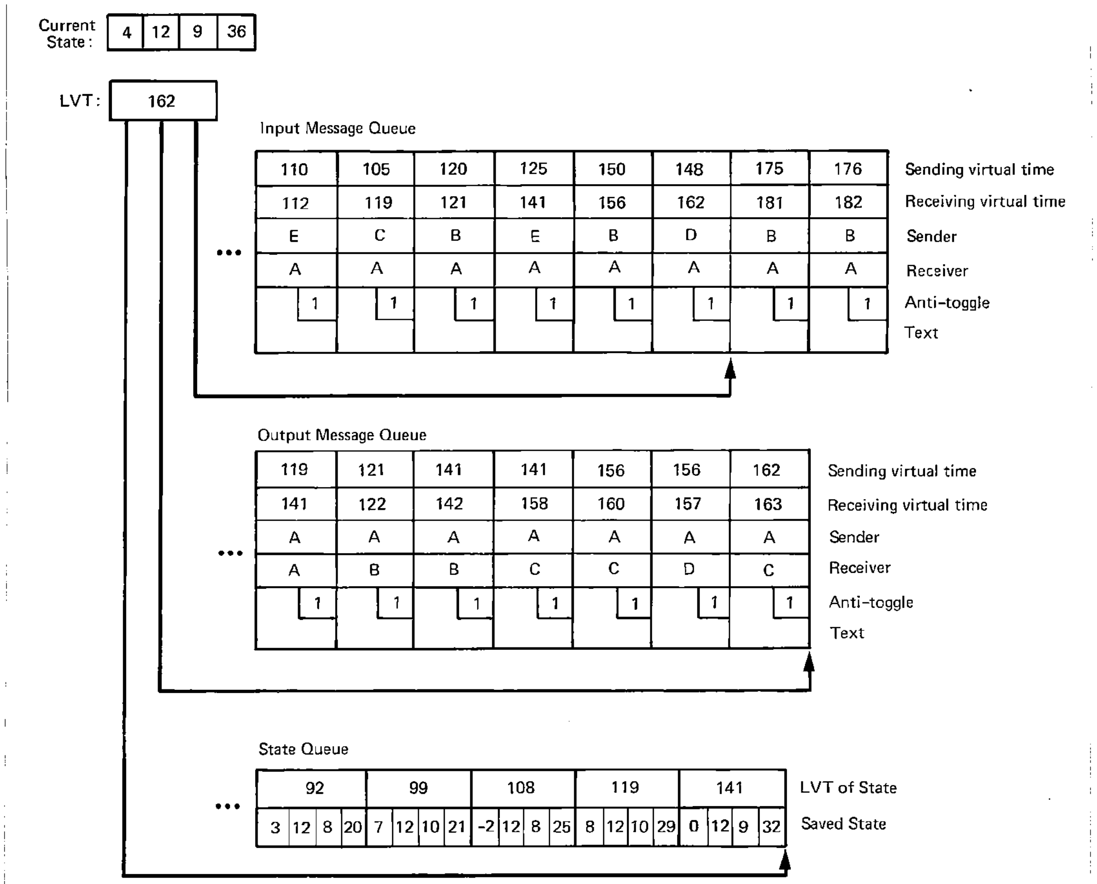


Fig. 6--The structure of a message in the Time Warp system

4. The receive time, also called the time stamp: the virtual time at which the message is scheduled to be received, used to order messages in the input queue of the receiving object.
5. The antitoggle: a bit indicating whether the message is an ordinary message (indicated by 1) or an antimessage (indicated by -1). Two messages that are alike in all fields but have complementary antitoggles are called antimessages of one another. Antimessages are created only during a rollback, and their existence is completely invisible to the simulation programmer.
6. The text: model-dependent event information, indicating presumably the kind of event, along with any parameter values needed for the simulation of the event.

In Fig. 7 we show in detail the data structure used to represent an object in the Time Warp system, including the scaffolding necessary to support the rollback mechanism. An object consists of five components.

1. The _LVT Register_ holds the LVT for this object. This is the virtual time of the last event processed by the object and, hence, acts as its local simulation clock.
2. The _Current State_ holds the variables representing the current (at virtual time LVT) state of the object, including its random seeds, if any.
3. The _Input Message Queue_ holds time stamped input messages. Some of the messages may be in the _local past_, meaning that they have time stamps earlier than the object's LVT and have already been processed; others are in the _local future_, and have not yet been processed (or else they have been processed one or more times but with rollbacks in each case).
4. The _Output Message Queue_ holds messages sent by the object to other objects. Output message queues are similar to input message queues except that the messages are ordered by the virtual time of _sending_, rather than by the time they are to be received. Some messages in the output queue were sent at times earlier than the object's LVT in the past, and others (if the object is in the Coasting Forward phase of a rollback) may have been sent in the local future.
5. The _State Queue_ holds copies of some of the object's past states, ordered by the LVT of those states. These states are saved in case the object must roll back. Not all past states



Current State:			
4	12	9	36

LVT: 162

Input Message Queue

110	105	120	125	150	148	175	176	Sending virtual time
112	119	121	141	156	162	181	182	Receiving virtual time
E	C	B	E	B	D	B	B	Sender
A	A	A	A	A	A	A	A	Receiver
1	1	1	1	1	1	1	1	Anti-toggle
								Text

Output Message Queue

119	121	141	141	156	156	162	Sending virtual time
141	122	142	158	160	157	163	Receiving virtual time
A	A	A	A	A	A	A	Sender
A	B	B	C	C	D	C	Receiver
1	1	1	1	1	1	1	Anti-toggle
							Text

State Queue

92	99	108	119	141	LVT of State
3 12 8 20	7 12 10 21	-2 12 8 25	8 12 10 29	0 12 9 32	Saved State

The figure shows a snapshot of Object A immediately after it has processed the input message with time stamp 162.

Fig. 7--The Structure of a Time Warp Object

need be saved; there must be at least one past state, but if more states are saved the rollback operation is generally faster.

For simplicity we will assume no message <u>coincidences</u> occur, no cases where two messages with equal time stamps (other than a message and its antimessage) are sent to the same object. All of the mechanisms we describe can be generalized easily to handle coincidences, but only at the expense of clarity in this description.

Whenever a message is inserted into a queue, whether it is an ordinary message or an antimessage, it is inserted at a position determined either by its sending time or its receiving time. Output queues are ordered by sending time and input queues are ordered by receiving time. However, when a message is "inserted" into a queue that already contains its antimessage, then both the message and the antimessage are destroyed, and the queue ends up with one fewer message rather than one more message. Thus, messages and their antimessages <u>annihilate</u> one another whenever they come into "contact," in a manner reminiscent of the behavior of particles and antiparticles in physics. This property is a key part of the rollback mechanism and is suggested by the following algebraic axioms, which partially specify the queueing discipline:

```
-M <> M
-(-M) = M
Insert(-M,Insert(M,Q)) = Q
```

where Q is a message queue and -M is the antimessage of M. It does not matter which, M or -M, is the ordinary message and which is the antimessage, so long as they are complementary. A message and its antimessage annihilate during the enqueueing operation, no matter what other messages were enqueued or annihilated in the meantime. (This latter fact is not captured by the above axioms.)

To show the Time Warp mechanism in detail, we examine the execution of a typical simulation. We assume, without loss of generality, that every object executes the basic loop shown in Fig. 8. All computation

```
declare State-Variables;
begin
    loop until termination
        Receive(Message);         { Receive contents of the next
                                    event message into the
                                    variable Message.            }
        Update(LVT);              { Increase LVT to the value in
                                    the time stamp of the message }
        Update-State;             { Calculate new values for
                                    State_Variables using Message
                                    and LVT                      }
        Send-Messages             { Send event messages to other
                                    objects using Message and LVT }
    endloop
end;
```

Fig. 8--An object's sequential control structure

in this loop is considered to be strictly sequential and deterministic, for reasons to be described later. At each iteration an object typically reads the next message from its input queue, updates its LVT, recomputes its state, and sends event messages to other objects. The Time Warp system exerts no effort to ensure that messages arrive at an object's input queue in monotonically increasing time order. In particular a process does not block simply because it cannot guarantee that a straggler will not arrive. Instead, the simulation proceeds on the assumption that there will be none, and that only if one does arrive should exceptional measures be taken.

When an object processes an input message it does not consume it; when it sends an output message, it retains a copy. Each object must save copies of all its input and output messages and some of its past states as well, in case it has to roll back. Fortunately it is not necessary to save states and messages all the way back to the beginning of the simulation. They need to be saved only as far back as a virtual time called GVT (Global Virtual Time), and GVT increases as the simulation proceeds so that the amount of storage needed to save past states and messages stays more or less constant. In Part II of this study we will show how to perform Time Warp simulation in a fixed amount of memory on a shared-memory multiprocessor, or a fixed amount per processor on a network. Because GVT and its influence on the global coordination of the simulation are the subject of Part II, we cannot discuss it here in detail. However, there is always enough back information present in the Time Warp system to perform any of the rollbacks that are called for.

THE ROLLBACK MECHANISM

Examining the situation in Fig. 7 we see that the object has an LVT of 162 and is ready to process the input message with time stamp 181. At this point if a message should arrive with a time stamp greater than 162 (in the future) then that message would simply be enqueued in its appropriate position (possibly causing annihilation if it is the antimessage of a message already in the queue) and the simulation would continue uninterrupted.

But what if a straggler (time stamp less than or equal to 162) should arrive? Then the simulation must roll back to the time in the time stamp of the message. To roll an object back to a time _t_, we must cancel all the side effects that occurred as a result of processing messages with time stamps greater than or equal to _t_, including indirect side effects that may have been induced in other objects because of message interactions. Rolling back one object may thus trigger other objects to roll back as well. Finally we restore the object to a state from _strictly_ _earlier_ than time _t_, and start the simulation forward again in a manner to be described.

For example, suppose object _F_ sends a message with time stamp 135 to object _A_ in Fig. 7. Because _A_'s LVT is 162, it "should" already have processed that message. The simulation must roll back to time 135 for it to proceed correctly. "Rolling back the simulation" does not mean rolling back every object. The only candidates for rolling back are _A_ and those other objects directly or indirectly affected by the messages that were sent by _A_ when its LVT was greater than or equal to 135.

The procedure for rolling back an object, shown in Figs. 9 through 12, consists of several steps. First the system inserts the straggler (with time stamp 135) into its proper position in the input queue of the receiver. Of course, if the arriving message is the antimessage of another message already in the input queue, this "insertion" actually results in the deletion of a message. Rollback must occur any time a straggler arrives, regardless of whether it causes an annihilation.

We will assume that each object's state is saved (by the Time Warp mechanism) "every once in a while"; exactly when is not particularly important. We believe that the simple strategy of saving an object's state every _k_ events at that object, for some small _k>=1_, is appropriate. The smaller _k_ is, the more efficient (in time) the rollback mechanism is, but the more it costs in memory to hold the saved states.

The rollback procedure is divided into three phases: _Restoration_, _Cancellation_ and _Coast Forward_. First, the Restoration phase restores the object's state variables, including random seeds and LVT, to the values they had when the last state save was done at a virtual time strictly earlier than the time stamp of the straggler. Because only a fraction of an object's past states are saved, the Restoration phase generally "overshoots" a little and restores a state one or more events earlier than the one that would allow immediate processing of the straggler.

Second, the Cancellation phase cancels all side effects in other objects that resulted from messages sent at or after that given in the time stamp of the straggler. For each such message this is done (as we will illustrate in a moment) _simply by sending its antimessage_.

Finally, because the Restoration phase generally overshoots a little, the Coast Forward phase is necessary to correct for the difference. It simulates the object forward again from the restoration point to the point where it can receive the straggler. As we will show, simulation in the Coasting Forward phase is an abbreviated version of normal forward simulation, and so we refer to it as "coasting." The

Coast Forward phase would not be necessary if the State Queue held a trace of all past states so that overshooting would not occur.

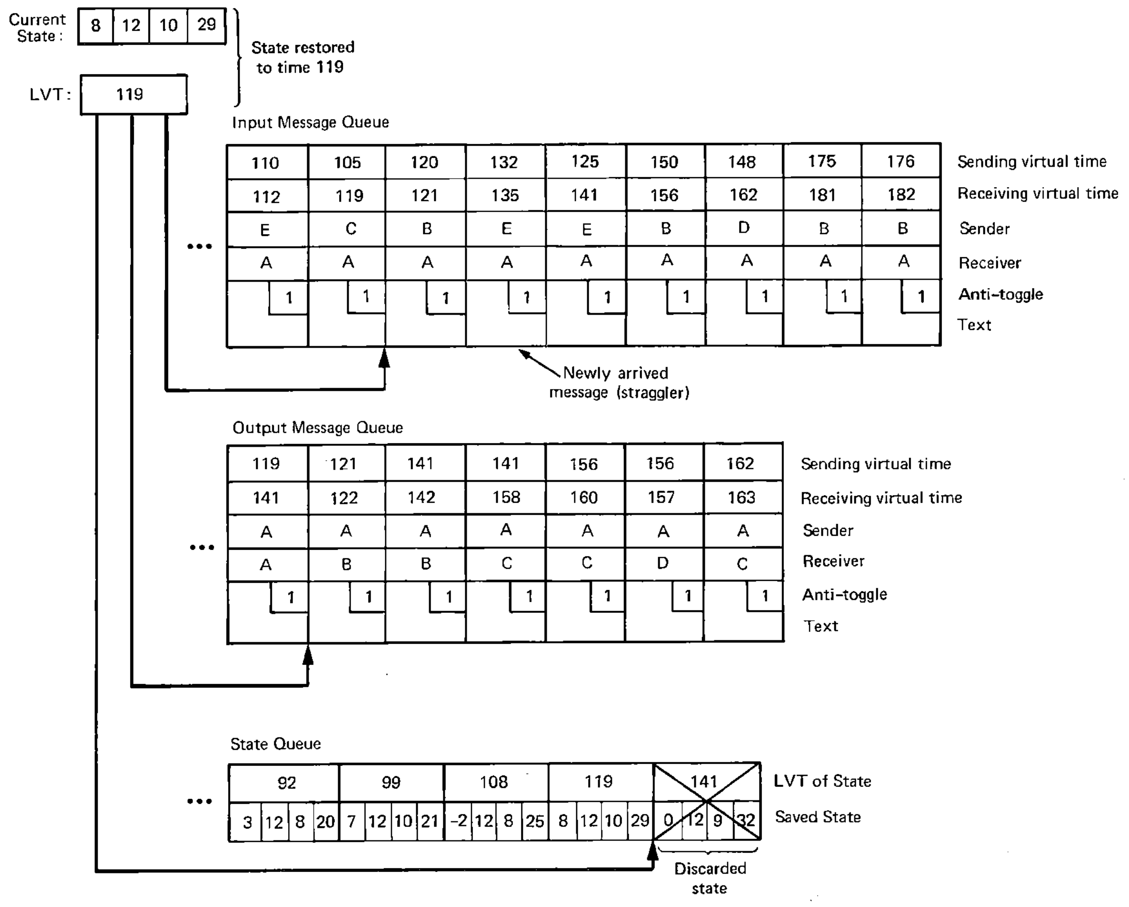
The Restoration Phase

Let us examine in detail the operation of the rollback mechanism on Object A. In the Restoration phase (Fig. 9) the system searches A's State Queue for the latest saved state from a virtual time strictly earlier than 135 (in this case the state from time 119) and restores the object's state variables to the values they had then. This represents a slight overshoot; if we had saved the state that existed after the processing of the message with time stamp 121, the Coast Forward phase would be unnecessary.

Those states in the State Queue having LVTs later than 135 represent a projected future for object A computed on the assumption that stragglers would not arrive. Because that assumption proved wrong, those projected future states are now discarded.

The Cancellation Phase

In the Cancellation Phase we must undo all of the direct and indirect side effects on other objects caused by messages that were sent by A after virtual time 135. It is not obvious how to do this because some of those messages might still be enqueued in the future part of the receiver's Input Queue waiting to be processed, while others may already have been processed and have caused side effects (and also have caused the sending of more messages to a third set of objects). It is even possible that some of the messages have been processed and reprocessed several times as a result of other unrelated rollbacks.



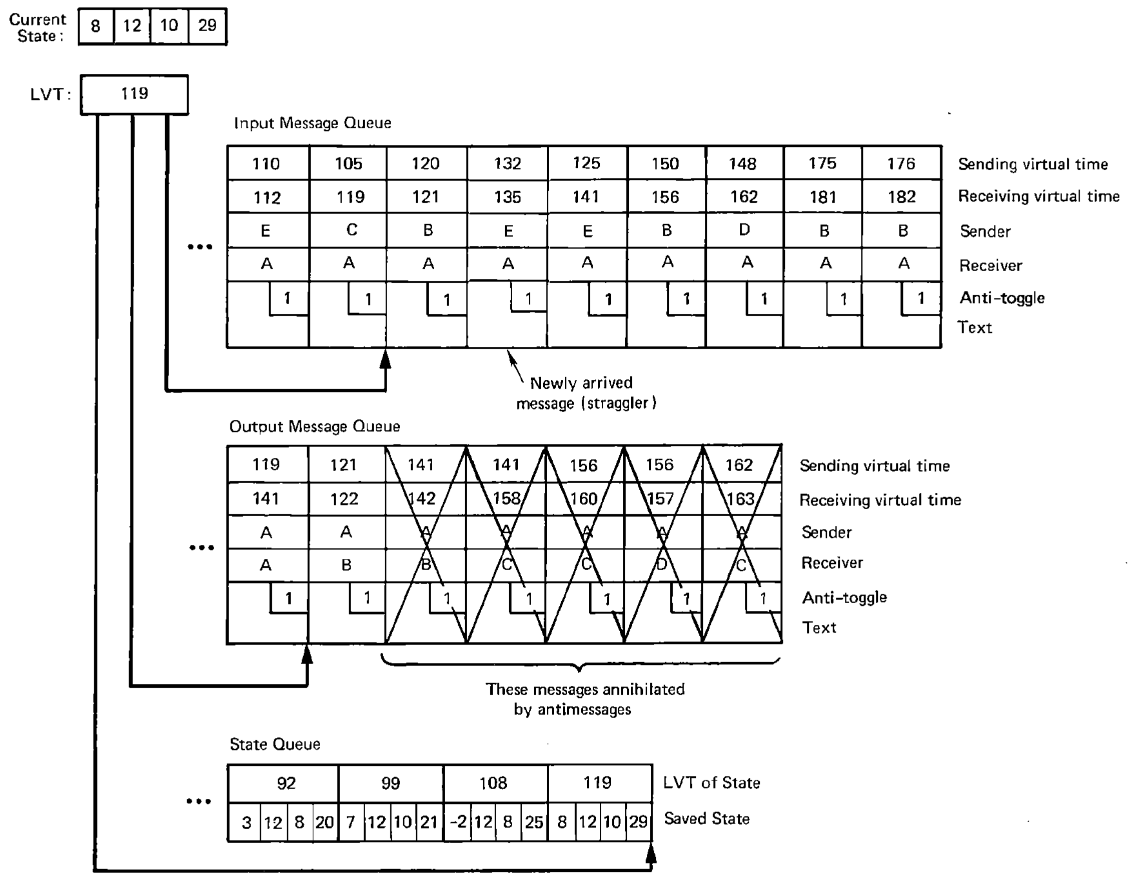
When an input message with a time stamp of 135 arrives, the message is first inserted into its proper position in the input queue. Because the message arrived in Object A's local past, the Restoration Phase of the Time Warp mechanism must restore A to a previous state, the last one that was saved at a time earlier than 135. In this case it restores Object A to the state saved at time 119 and discards those states saved after time 119.

Fig. 9--The Restoration Phase of the rollback procedure

Nevertheless, the mechanism that accomplishes the intended effect is extremely simple: to cancel exactly the side effects of a single message we simply send its antimessage. Thus, in the example in Fig. 10 the Time Warp system will send antimessages for all of the five messages that have to be canceled.

The following is an example of how this works for the first of the five messages, the one sent at simulation time 141 to object _B_ for receipt at time 142. The Time Warp system first constructs the antimessage for this message, identical in all respects except that its antitoggle is -1 instead of 1. Sending an antimessage is exactly like sending an ordinary message. One copy is inserted into the output queue of the sender, and another copy is transmitted and inserted into the input queue of the receiver. When the antimessage is enqueued in _A_'s output queue it "meets" the original message, and the two are annihilated. The net effect is to remove the original message from the output queue so that there is no longer any record at the sender that the message was ever sent. When the other copy of the antimessage is enqueued in _B_'s input queue, one of the following will happen:

1. The antimessage may arrive in _B_'s future. If so, it simply annihilates the original message in the input queue and the net effect is as though the original message had never been sent.
2. It may arrive in _B_'s past. If so, it is still enqueued in _B_'s input queue, and still annihilates itself and the original message. However, because it arrived in _B_'s past, _B_ must roll back to virtual time 142. This may, of course, cause the sending of another round of antimessages.


Current State:
LVT:
Input Message Queue
Newly arrived message (straggler)
Output Message Queue
These messages annihilated by antimessages
State Queue
...

Current State:

8	12	10	29

LVT: 119

Input Message Queue

110	105	120	132	125	150	148	175	176	Sending virtual time
112	119	121	135	141	156	162	181	182	Receiving virtual time
E	C	B	E	E	B	D	B	B	Sender
A	A	A	A	A	A	A	A	A	Receiver
1	1	1	1	1	1	1	1	1	Anti-toggle
									Text

Output Message Queue

119	121	~~141~~	~~141~~	~~156~~	~~156~~	~~162~~	Sending virtual time
141	122	~~142~~	~~158~~	~~160~~	~~157~~	~~163~~	Receiving virtual time
A	A	~~A~~	~~A~~	~~A~~	~~A~~	~~A~~	Sender
A	B	~~B~~	~~C~~	~~C~~	~~D~~	~~C~~	Receiver
1	1	~~1~~	~~1~~	~~1~~	~~1~~	~~1~~	Anti-toggle
							Text

State Queue

92	99	108	119	LVT of State
3 12 8 20	7 12 10 21	-2 12 8 25	8 12 10 29	Saved State

All messages sent at or after time 135 must be canceled. The Time Warp mechanism will construct their antimessages and send them exactly as it would send ordinary messages; it will enqueue the antimessages in A's output queue, transmit copies to the receivers, and enqueue them in the receivers' input queues.

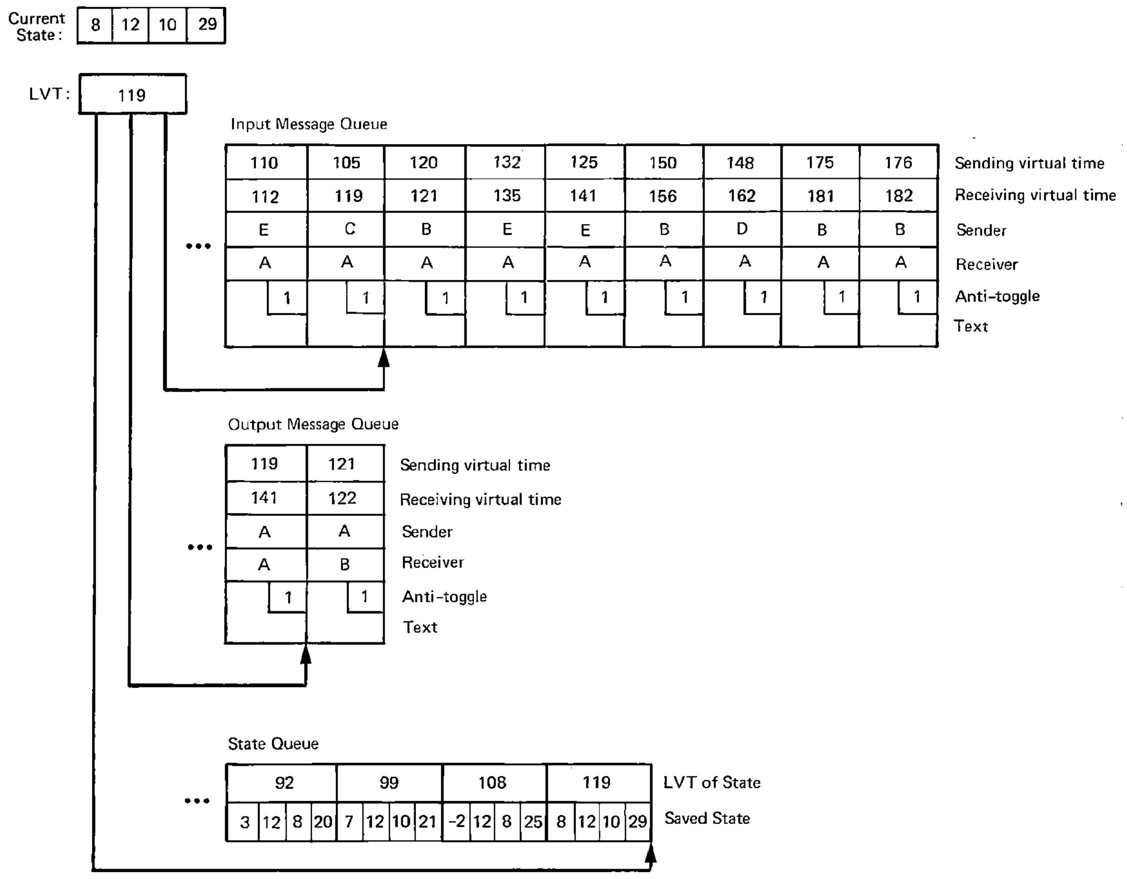
Fig. 10--Before the Cancellation Phase

From the second case we see that the rollback mechanism does a tree-walk in the tree of objects rooted at _A_, with branches representing message transmissions. Notice that the "tree" is possibly a cyclic graph, and that in the course of rolling back object _A_ the mechanism may in fact cycle back to _A_. This causes no harm and can be carried out without danger of infinite loop, deadlock, or any similar hazard.

The Coasting Forward Phase

In Fig. 11 we see object _A_ after all five messages have been canceled. We are now ready to Coast Forward. The output message sent at time 121 was not canceled; and, in general, output messages between the restoration point (119) and the time of the straggler (135) are never canceled. The reason is that the simulation will take exactly the same path from time 119 to time 135 as it did before the restoration. After time 135 it can change course as a result of the new message. Message 121 is certain to be re-sent, so it is wasteful to cancel it because the cancellation may cause other objects to roll back needlessly.

During Coasting Forward the object executes the simulation cycle as usual, except that every time it "sends" a message, the sending is inhibited because the sender's output queue and the receiver's input queue already have copies anyway. Thus, during the Coasting Forward phase an object only goes through the motions of message transmission. The significant actions occurring during Coasting Forward are changes to the object's LVT and state.


Current State:
8 | 12 | 10 | 29
LVT: 119
Input Message Queue
110 | 105 | 120 | 132 | 125 | 150 | 148 | 175 | 176 Sending virtual time
112 | 119 | 121 | 135 | 141 | 156 | 162 | 181 | 182 Receiving virtual time
E | C | B | E | E | B | D | B | B Sender
A | A | A | A | A | A | A | A | A Receiver
1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 Anti-toggle
Text
Output Message Queue
119 | 121 Sending virtual time
141 | 122 Receiving virtual time
A | A Sender
A | B Receiver
1 | 1 Anti-toggle
Text
State Queue
92 | 99 | 108 | 119 LVT of State
3 12 8 20 | 7 12 10 21 | -2 12 8 25 | 8 12 10 29 Saved State

Here we see that all output messages sent at or after time 135 have been annihilated by antimessages. Not depicted in this figure is the fact that copies of those antimessages are on their way to the receivers of the original messages. There they will also cause annihilation and, possibly, more rollbacks.

Fig. 11--After the Cancellation Phase

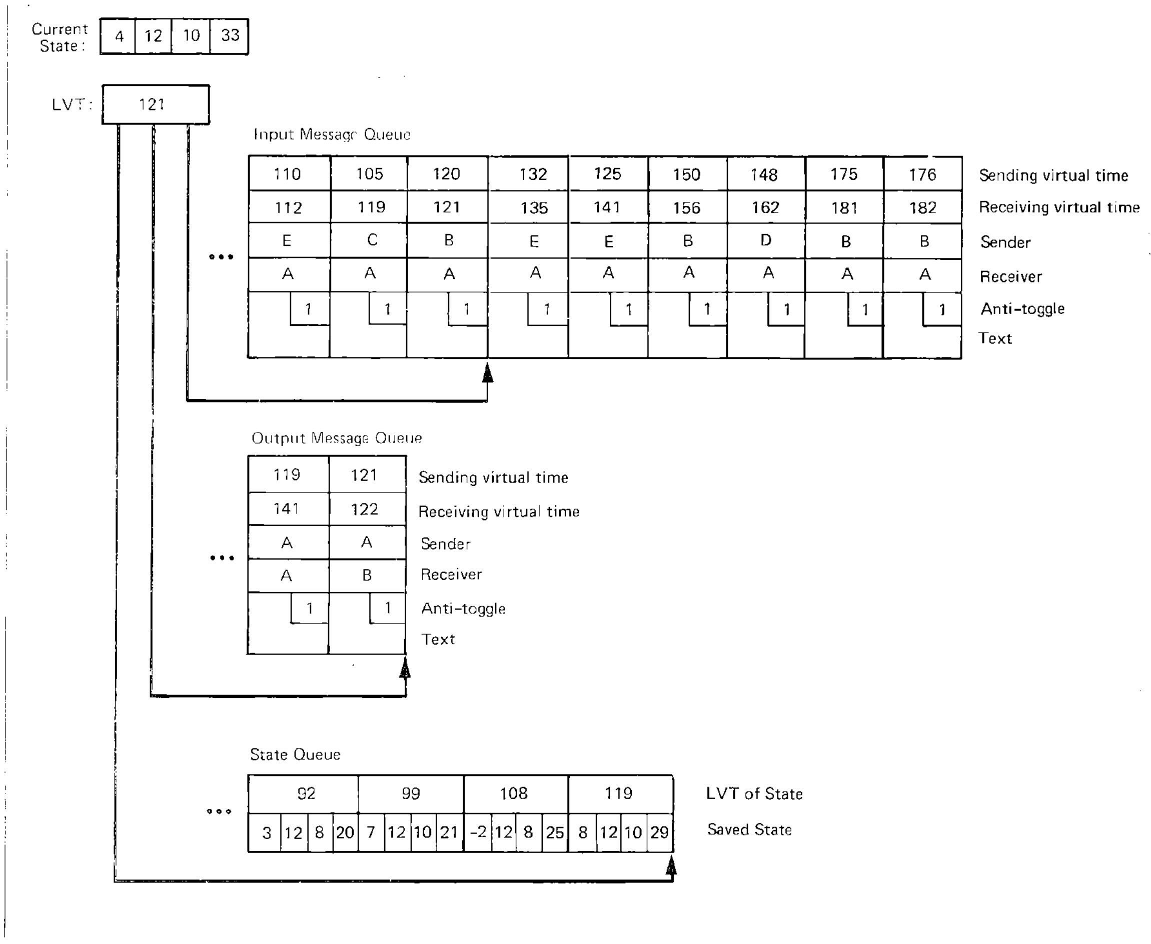
The facts that message 121 is "certain to be re-sent" and that the simulation will take "exactly the same path" during the Coasting Forward phase that it did originally are consequences of our assumption that the program body in each object is deterministic. If we were to allow the body of an object to be nondeterministic, it would be necessary to eliminate the Coasting Forward phase of the roll back mechanism entirely, which can be done only if every object's state is saved after each event at that object. We believe that this alternative would usually require too much memory. An object can still model probabilistic behavior without violating the prohibition on nondeterminism. All that is necessary is to make the random number seeds part of the object's state so that they can be saved and restored along with the rest of the state.

It is very important that the messages sent by _A_ between the restoration time (119 in the example) and the time of the straggler (135) are not canceled and then re-sent, but are definitely not canceled and not re-sent. First, this is a great saving of time because the costs associated with sending messages are almost always the dominant costs in object-oriented simulation. In addition, this part of the mechanism is not merely an optimization. It is necessary to prevent the possibility of rollbacks cascading arbitrarily far back in virtual time. Without this feature, for example, the antimessage with time stamp 142 sent to _B_ during _A_'s rollback might cause _B_ to roll back _to a time earlier than 135_. This might happen because the latest state saved for _B_ had an LVT of 110. _B_, then, in the course of rolling back to a time earlier than 110, might have to send an antimessage to _A_ with a time stamp of, say, 115, causing A to roll back even further. A's second rollback might cause antimessages to be sent to B, causing yet another rollback, and there is apparently nothing to prevent the possibility of all objects cascading back to time 0.

However, the inhibition of both the cancellation and re-sending of output messages sent between 119 and 135 guarantees that the messages sent by A during and immediately after its rollback (until the next one) must have time stamps greater than or equal to the 135. As a result, a rollback to time 135 can cause secondary rollbacks to times no earlier than 135. More precisely, those objects restore their last saved state from before 135 and then coast forward without sending any messages until they get to time 135. This prevents the backward cascade.

In Fig. 12 we see the last stage in the rollback process. The object is now in the correct state to receive the straggler and continue simulating forward.

One final note: After the Restoration and Cancellation phases of a rollback, new messages may arrive and new rollbacks may begin before the Coasting Forward phase of the first rollback has finished. It is possible to have any number of rollbacks "in progress" at the same time. With care in the implementation for handling these possibilities, Coasting Forward need not be an atomic action and does not have to be considered part of the rollback mechanism. It is sometimes better to view Coasting Forward as an alternative mode of forward simulation. An object stays in Coasting Forward Mode (with message sending inhibited) until it completes all rollbacks it has in progress; when no rollbacks are in progress, it is in Normal Mode (sending messages).



Current State:			
4	12	10	33

Input Message Queue

110	105	120	132	125	150	148	175	176	Sending virtual time
112	119	121	135	141	156	162	181	182	Receiving virtual time
E	C	B	E	E	B	D	B	B	Sender
A	A	A	A	A	A	A	A	A	Receiver
1	1	1	1	1	1	1	1	1	Anti-toggle
									Text

Output Message Queue

119	121	Sending virtual time
141	122	Receiving virtual time
A	A	Sender
A	B	Receiver
1	1	Anti-toggle
		Text

State Queue

92				99				108				119				LVT of State
3	12	8	20	7	12	10	21	-2	12	8	25	8	12	10	29	Saved State

The input message with time stamp 121 is re-received and reprocessed. As a result, Object A updates its state and "sends" a message to Object B--an exact copy of the one still in the output queue that A sent the last time it received message 121 before the rollback. But because A is in the Coasting Forward phase, the actual sending of the message (including its enqueueing in A's output queue) is inhibited. This is why Coasting Forward is faster than ordinary forward simulation. In this example the rollback procedure is now complete, and A can start normal forward simulation by receiving the original straggler with time stamp 135.

Fig. 12--The Coasting Forward Phase

V. DISCUSSION

WEAK CORRECTNESS OF THE MECHANISM

Space does not permit us to develop a mathematical formalism for the semantics of simulation; hence we cannot make a formal argument for the correctness of our mechanism. However, we can outline an informal argument that when the Time Warp mechanism is used to execute a simulation program it duplicates (with one possible exception) the input-output behavior produced by the same program when executed under the sequential event-list mechanism familiar from such languages as Simscript (Consolidated Analysis Centers, 1976) or Simula-67 (Dahl, Myhrhaug, and Nygaard, 1970; Birtwistle et al., 1973). The exception concerns the behavior of an object when it receives two or more messages with exactly the same time stamp. Some languages take the position that such coincidental messages are executed in FIFO order; others allow a priority scheme to determine the order. There are also good arguments for leaving the order undefined. Under the Time Warp mechanism, all messages with the same time stamp are delivered to the object together, as a set. It is then up to the object's program to decide how to handle the situation. We believe that this is the cleanest and most general solution to the problem, allowing the object to choose which of the several conventions is appropriate for the application. Aside from this issue, the Time Warp mechanism produces exactly the same output as the event list mechanism.

The full demonstration of this fact can be made only after a discussion of the concept of Global Virtual Time in Part II, but we can present the thrust of the argument. After all of the false projected futures and all of the rolling back, the net effect is that each object processes the same events (time stamped messages) as it would according to the sequential event mechanism, and in the same order (increasing virtual time). To be sure, some messages having no counterpart in the sequential mechanism can be generated and sent during the time that an object is charging ahead not knowing that a straggler is on its way. But all such "false" messages eventually get canceled, leaving only those that have counterparts in the sequential mechanism. Also, events may get processed globally in a different real-time order from the increasing virtual time discipline imposed by the sequential mechanism; but locally, at each object, they are processed according to that discipline, and that is all that matters for input/output equivalence.

This argument relies on the fact that the Time Warp mechanism cannot deadlock and that under fair scheduling the simulation does make progress. We address those issues now.

TIME WARP IS DEADLOCK FREE

Earlier we adopted the meaning of "deadlock" to be "unterminated processes permanently ineligible to execute." This is perhaps a narrow definition in the context of a concurrent system where message cancellation and rollback are possible. One could define deadlock as the inability of the computation to progress forward with its mission. Under this definition a computation that moves forward a little and then rolls back over and over again, without at least progressing farther each time, is deadlocked in the broader sense.

If we adopt the narrow definition of deadlock for the moment we can prove very simply that under fair scheduling the Time Warp mechanism _per se_ is deadlock free, although particular simulation runs may deadlock. A Time Warp object is _always_ eligible to execute (and under fair scheduling will eventually execute) unless it has (at least momentarily) exhausted its input queue. Consider a subset of the objects in a simulation and ask how all of the objects in it might be _permanently_ blocked. At any moment they can all be blocked simultaneously if and only if they have processed all of the messages in their input queues. They will then be _permanently_ blocked if and only if no more messages arrive from any object outside of the subset.

Such collective permanent blocking in a set of objects is in some sense a local deadlock. But it does not arise from the Time Warp mechanism; it simply reflects quiescence in that portion of the simulated model. Because this argument applies to any subset of the simulation, it obviously applies to the whole simulation as well. The whole simulation "deadlocks" only when all objects have exhausted their inputs. Such a situation is properly interpreted as quiescence or termination in the simulated model, not deadlock in the Time Warp mechanism.

In Part II we will prove that any Time Warp simulation is deadlock free in the broader sense that it will always make progress in virtual time unless the user's simulation model quiesces or violates one of the following two semantic rules.

1. Each object must compute for only a finite amount of time (and send only a finite number of messages) for each event.
2. There must be no infinite sequence of objects $\underline{A}_1$, $\underline{A}_2$, $\underline{A}_3$ ... and no virtual time $\underline{t}$ such that each object in the sequence sends a message with time stamp $\underline{t}$ to the next object in the sequence.

Violation of these conditions is analogous to having an infinite loop or infinite recursion in the user's program.

Although the full proof of progress depends on precise definition of the concept of Global Virtual Time in an environment where arbitrary message delays are possible, we can sketch the argument here for the case of instantaneous message transmission. In the context of instantaneous message transmission, let us define the farthest behind object of a simulation snapshot to be that object whose LVT is minimal over all objects with at least one unprocessed message in their input queues. (Assume that there is a unique farthest behind object.) We can then state that the farthest behind object cannot roll back. The reason is that it could only roll back if a message were to arrive in its past, but no object can send such a message. All objects are either blocked (with no messages in their input queues) or are at least as far ahead as the farthest behind object and can only send messages (instantaneously) with time stamps greater than or equal to their LVTs. If we define the GVT of a snapshot to be the LVT of the farthest behind object, we find that GVT can never decrease during the simulation. Assuming (1) that the user has followed the restrictions described above, (2) that there is always at least one object with an unprocessed message (so there is no quiescence) and (3) that scheduling is fair, GVT must actually increase. The value of GVT is thus a natural measure of progress in the simulation.

EFFICIENCY

Despite the comforting facts that the Time Warp system does not deadlock and that it does progress forward, one might still worry that it might progress forward too slowly--ten steps forward and nine steps back. Consider that the rollback of one object might start a chain reaction of rollbacks among other objects. In principle, it is possible for a single message to cause indirectly a rollback of every object in the simulation other than the message's sender. Although this is a matter for empirical testing, there are several general arguments to suggest that the actual incidence of rollback will be much less than one might fear, especially in large simulations.

First, when one object rolls back and sends out antimessages, the number of antimessages and the number of possible secondary rollbacks is proportional to how far back (measured in events) the object is rolled. In simulations that are <u>nearly synchronous</u> (in the sense that the average number of events between GVT and LVT is small) the number of antimessages should be small. Furthermore, the very act of roll back tends to make the simulation more nearly synchronous, making other rollbacks less likely in the near future.

Second, even though the rollback of one object may cause antimessages to be sent, only a fraction of them will arrive in the "past" of the receiving object to cause secondary rollbacks. And the secondary rollbacks that do occur will not be over as many events, on the average, as the primary one was. For example, in Figs. 9 through 12 the primary rollback is to virtual time 135; the secondary rollbacks, if they happen at all, will only be to times 142, 180, 160, 157, or 163.

Third, there are implementation techniques available to further minimize the number of rollbacks. For example we can cancel messages in early-to-late order, so that if two or more antimessages are directed to the same object, usually no more than one of them will cause a rollback. In our example, Fig. 10, three antimessages will be sent to object C. The first of them might cause a rollback to virtual time 158; if so, the next two antimessages, with time stamps 160 and 163, will probably arrive in C's future and not cause further rolling back.

The result of these considerations is that although antimessages branch outward from the primary rollback site in tree-like fashion, the growth of that tree is confined and pruned by several independent effects. Our prototype implementation will allow measurement of these effects.

TRANSPARENCY OF THE TIME WARP MECHANISM

One of the most important properties of the Time Warp mechanism is that it is completely transparent to the simulation programmer. Aside from writing the program in an object-oriented, message-passing style the programmer can be almost completely unaware of the presence of the Time Warp mechanism. No special declarations or initialization are needed, as might be needed in the Network Paradigm to describe the topology of the network. No extra logic is necessary to "predict" lower bounds on the time stamp of the next message to be sent. Models need not be distorted by adopting either the minimum service time restriction or the message monotonicity restriction. It is not necessary to know how many processors are in use, or whether they are connected in a network or in a shared-memory multiprocessor configuration. Of course many issues, such as storage management and the assignment of objects to processors, are much easier for the Time Warp system implementor to handle on a shared memory architecture such as C.mmp (Wulf, Levin, and Harbison, 1981) or Cm* (Swan, Fuller, and Siewiorek, 1977) than they are on a distributed architecture such as the DCS system (Farber et al., 1973) or an Ethernet system (Metcalfe and Boggs, 1976), but the basic simulation mechanism remains the same, and the user's programming process is largely unaffected.

As noted, the Time Warp mechanism does not require that the communication medium preserve the order of messages; it only requires that the messages eventually get to their destinations reliably. This flexibility is provided partly by the rollback mechanism, which allows event messages to arrive in any order. It derives also from the definition of the annihilation mechanism, which works properly even when an antimessage arrives at the destination before the corresponding ordinary message. (It is possible that an antimessage will actually be received by an object before the corresponding ordinary message arrives. The implementor is free to choose how the object behaves in this situation. Whatever the behavior, the object is certain to roll back and undo it when the ordinary message eventually arrives.)

Independence of the order of arrival of messages a property not only of the local control part of the Time Warp mechanism, but also of the global control part as well, as we will show in Part II. This means that the Time Warp mechanism can run correctly, <u>without substantive change</u>, on store-and-forward networks such as the ArpaNet, although it remains to be seen whether much speedup can be gained on such an architecture.

Several other issues are involved in complete transparency whose full treatment must be postponed until Part II. One is the issue of errors. When an object in a simulation encounters a computational error (such as array bounds violation etc.), that error in one object must not immediately cause the entire simulation terminate abnormally. It is quite possible that the object will subsequently roll back and "uncommit" the error. In general, it is only when the farthest behind object commits an error that the the entire simulation must abort. Alternatively we can say that an error is only permanent when GVT catches up to the LVT of the object at the time of the error. The detection of this latter condition is done by the Global Control part of the Time Warp mechanism.

Another issue is input/output. If an object executes an output operation, it is important to buffer that output and not immediately commit an irreversible, externally visible action. Again this is because the object might have to roll back and it is important to be able to cancel all side effects. Output can be irrevocably committed only when GVT catches up to the LVT of the object ordering the output. By representing all output devices and files as objects within the simulation it is possible to have the necessary logic implemented with perfect transparency as part of the Global Control mechanism.

The transparency of the Time Warp mechanism does not mean, of course, that the programmer can be completely ignorant of the basic principles of concurrent programming. For example, it is important for efficiency that he design his models in such a way that none of the objects is a communication bottleneck. But aside from such common sense considerations in distributed computation there are no extra restrictions imposed by the fact that he is programming a simulation. He can concentrate on the logical design of the model, rather than on the implementation.

REFERENCES

Birtwistle, G. M., O. Dahl, B. Myhrhaug, and K. Nygaard, Simula Begin, Auerbach Publishers, Inc., Philadelphia, Pa., 1973.

Chandy, K. M. and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," IEEE Transactions on Software Engineering SE-5, (5), September 1979, 440-452.

_____, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," CACM 24 (11), April 1981, 198-206.

Consolidated Analysis Centers, Inc., Simscript II.5 Reference Handbook, 12011 San Vicente Blvd., Los Angeles, Cal. 90049, 1976.

Dahl, O., B. Myhrhaug, and K. Nygaard, Common Base Language, Norwegian Computing Center, Oslo, October 1970.

Farber, D. J., J. Feldman, F. R. Heinrich, M. D. Hopwood, K. C. Larson, D. C. Loomis, and L. A. Rowe, "The Distributed Computing System," in Proceedings 7th Annual IEEE Computer Society International Conference, IEEE Press, New York, 1973, 31-34.

Fishman, G. S., A Wiley-Interscience Publication. Volume: Principles of Discrete Event Simulation, John Wiley & Sons, New York, 1978.

Franta, W. R., The Computer Science Library, Operating and Programming Systems Series. Volume: The Process View of Simulation, Elsevier North-Holland, Inc., New York, 1977.

Klahr, P., D. McArthur, S. Narain, and E. Best, SWIRL: Simulating Warfare in the ROSS Language, The Rand Corporation, N-1885-AF, 1982.

McArthur, D. and P. Klahr, The ROSS Language Manual, The Rand Corporation, N-1854-AF, 1982.

Metcalfe, R. M. and D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," CACM 19 (7), July 1976, 395-404.

Peacock, J. K., J. W. Wong, and E. Manning, "A Distributed Approach to Queueing Network Simulation," in 1979 Winter Simulation Conference, IEEE, New York, 1979a, 399-406.

_____, "Distributed Simulation Using a Network of Processors," Computer Networks 3 (1), February 1979b, 44-56.

Peacock, J. K., E. G. Manning, and J. W. Wong, "Synchronization of Distributed Simulation Using Broadcast Algorithms," Computer Networks 4 (1), February 1980, 3-10.

Sowizral, H. A. and D. R. Jefferson, "Synchrony Versus Asynchrony for Concurrent Simulation," The Rand Corporation (forthcoming).

Swan, R., S. H. Fuller, and D. Siewiorek, "CM* --- A modular, multimicrocomputer," in Proceedings 1977 National Computer Conference, AFIPS Press, Baltimore, MD, 1977, 637-644.

Wulf, W. A., R. Levin, and S. P. Harbison, HYDRA/C.mmp: An Experimental Computer System, McGraw-Hill, New York, 1981.

RAND/N-1906-AF