Proceedings of the 2001 Winter Simulation Conference
B. A. Peters, J. S. Smith, D. J. Medeiros, and M. W. Rohrer, eds.

PARALLEL AND DISTRIBUTED SIMULATION SYSTEMS

Richard M. Fujimoto

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280, U.S.A.

ABSTRACT

Originating from basic research conducted in the 1970's and 1980's, the parallel and distributed simulation field has matured over the last few decades. Today, operational systems have been fielded for applications such as military training, analysis of communication networks, and air traffic control systems, to mention a few. This tutorial gives an overview of technologies to distribute the execution of simulation programs over multiple computer systems. Particular emphasis is placed on synchronization (also called time management) algorithms as well as data distribution techniques.

1 INTRODUCTION

Parallel and distributed simulation is concerned with issues introduced by distributing the execution of a discrete event simulation program over multiple computers. *Parallel* discrete event simulation is concerned with execution on multiprocessor computing platforms containing multiple central processing units (CPUs) that interact frequently, e.g., thousands of times per second. *Distributed* simulation is concerned with the execution of simulations on loosely coupled systems where interactions take much more time, e.g., milliseconds or more, and occur less often. It includes execution on geographically distributed computers interconnected via a wide area network such as the Internet. In both cases the execution of a single simulation model, perhaps composed of several simulation programs, is distributed over multiple computers.

There are two principal categories of simulations of concern here. The first are simulations primarily used for analysis, e.g., to evaluate alternate designs or control policies of a complex system, e.g., an air traffic network. Here the principal goal is to compute results of the simulation as quickly as possible in order to improve the effectiveness of the simulation tool. A related application is to use simulations to evaluate alternate courses of action, e.g., to evaluate different control actions imposed on an air traffic network in order to reduce delays induced by inclement weather in one portion of the air space. Use of simulations to manage ongoing processes is referred to as on-line simulation.

The second type of simulation of interest here are those used to create virtual environments into which humans and/or hardware devices are embedded. Such environments are widely used for training, entertainment (e.g., video games), and test of evaluation of devices. Virtual environments have been used extensively to train military personnel because they provide a much safer, more cost effective, and environmentally friendlier approach to training than field exercises.

Parallel and distributed simulation systems can provide substantial benefit to these applications in several ways:

- Execution times of analytic simulations can be reduced by subdividing a large simulation computation into many sub-computations that can execute concurrently. One can reduce the execution time by up to a factor equal to the number of processors that are used. This may be important simply because the simulation takes a long time to execute, e.g., simulations of communication networks containing tens of thousands of nodes may require days or weeks for a single run.
- Very fast executions are needed for on-line simulations because there is often very little time available to make important decisions. In many cases, simulation results must be produced in seconds in order for simulation results to be useful. Again, parallel simulation provides a means to reduce execution time.
- Simulations used for virtual environments must execute in real time, i.e., the simulator must be able to simulate a second of activity in a second of wallclock time so that the virtual environment appears realistic in that it evolves as rapidly as the actual system. Distributing the execution of the simulation across multiple processors can help to achieve this property. Ideally, *scalable* execution can be obtained whereby the distributed simulation continues to run in real time as the system be-

ing simulated and the number of processors are increased in proportion.

- Distributed simulation techniques can be used to create virtual environments that are geographically distributed, enabling one to allow humans and/or devices to interact as if they were co-located. Such *distributed virtual environments* have obvious benefits in terms of convenience and reduced travel costs.
- Distributed simulation can simplify integrating simulators that execute on machines from different manufacturers. For example, flight simulators for different types of aircraft may have been developed on different architectures. Rather than porting these programs to a single computer, it may be more cost effective to "hook together" the existing simulators, each executing on a different computer, to create a new virtual environment.
- Another potential benefit of utilizing multiple processors is increased tolerance to failures. If one processor fails, it may be possible for other processors to continue the simulation provided critical elements do not reside on the failed processors.

Work in parallel and distributed simulation systems has taken place in three, largely separate research communities. The first is the high performance computing community which was concerned primarily with speeding up the execution of simulation programs by distributing their execution over multiple CPUs. Early work in synchronization algorithms dates back to the late 1970's with seminal work by Chandy and Misra (1978), and Bryant (1977) (among others) who are credited with first formulating the synchronization problem and developing the first algorithms to solve it. These algorithms are among a class of algorithms that are today referred to as conservative synchronization algorithms. A few years later, seminal work by Jefferson developed the Time Warp algorithm (Jefferson 1985). Time Warp is important because it defined fundamental constructs widely used in a class of algorithms termed optimistic synchronization. Conservative and optimistic synchronization techniques form the core of a large body of work concerning parallel discrete event simulation techniques.

The second community involved in the development of distributed simulation technology is the defense community. While the high performance computing community was largely concerned with reducing execution time, the defense community was concerned with integrating separate training simulations in order to facilitate interoperability and software reuse. The SIMNET (SIMulator NETworking) project (1983 to 1990) demonstrated the viability of using distributed simulations to create virtual worlds for training soldiers in military engagements (Miller and Thorpe 1995). This lead to the development of a set of standards for interconnecting simulators known as the Distributed Interactive Simulation (DIS) standards (IEEE Std 1278.1-1995 1995). The 1990's also saw the development of the Aggregate Level Simulation Protocol (ALSP) that applied the SIMNET concept of interoperability and model reuse to wargame simulations. ALSP and DIS have since been replaced by the High Level Architecture whose scope spans the broad range of defense simulations, including simulations for training, analysis, and test and evaluation of hardware components.

A third track of research and development efforts arose from the Internet and computer gaming community. Work in this area can be traced back to a role-playing game called dungeons and dragons and a textual fantasy computer game called Adventure developed in the 1970's. These soon gave way to MultiUser Dungeon (MUD) games in the 1980's. Important additions such as sophisticated computer graphics helped created the video game industry that is flourishing today.

This paper is organized as follows. The next section is concerned with the execution of analytic simulations on parallel computers, with the principal goal of reducing execution time. Synchronization is a key problem that must be addressed. Section 3 is concerned with an approach to parallel simulation known as time decomposition. Section 4 is concerned with distributed virtual environments and issues such as data distribution that arise in that domain. This paper is an updated version of a previous tutorial presented at this conference in (Fujimoto 1999b). A much more detailed treatment of this subject is presented in (Fujimoto 2000).

2 TIME MANAGEMENT

Time management is concerned with ensuring that the execution of the parallel/distributed simulation is properly synchronized. This is particularly important in analytic simulations. Time management not only ensures that events are processed in a correct order, but also helps to ensure that repeated executions of a simulation with the same inputs produce exactly the same results. Currently, time management techniques such as those described here are typically not used in training simulations, where incorrect event orderings and non-repeatable simulation executions can usually be tolerated.

Time management algorithms usually assume the simulation consists of a collection of *logical processes* (LPs) that communicate by exchanging timestamped messages or events. The goal of the synchronization mechanism is to ensure that each LP processes events in timestamp order; this requirement is referred to as the *local causality constraint*. It can be shown that if each LP adheres to the local causality constraint, execution of the simulation program on a parallel computer will produce exactly the same results as an execution on a sequential computer. An important side effect of this property is that

it is straightforward to ensure that the execution of the simulation is repeatable.

Each LP can be viewed as a sequential discrete event simulation. This means each LP maintains some local state and a list of time stamped events that have been scheduled for this LP (including local events within the LP that it has scheduled for itself), but have not yet been processed. This pending event list must also include events sent to this LP from other LPs. The main processing loop of the LP repeatedly removes the smallest time stamped event and processes it. Thus, the computation performed by an LP can be viewed as a sequence of event computations. Processing an event means zero or more state variables within the LP may be modified, and the LP may schedule additional events for itself or other LPs. Each LP maintains a simulation time clock that indicates the time stamp of the most recent event processed by the LP. Any event scheduled by an LP must have a time stamp at least as large as the LP's simulation time clock when the event was scheduled.

Time management algorithms can be classified as being either *conservative* or *optimistic*. Each of these are described next.

2.1 Conservative Synchronization

The first synchronization algorithms were based on conservative approaches. This means the synchronization algorithm takes precautions to avoid violating the local causality constraint. For example, suppose an LP is at simulation time 10, and it is ready to process its next event with time stamp 15. But how does the LP know it won't later receive an event from another LP with time stamp (say) 12? The synchronization algorithm must ensure no event with time stamp less than 15 can be later received before it can allow the time stamp 15 event to be processed.

Thus, the principal task of any conservative protocol is to determine when it is "safe" to process an event, i.e., when can one guarantee no event containing a smaller time stamp will be later received by this LP. An LP cannot process an event until it has been guaranteed to be safe.

2.1.1 First Generation Algorithms

The algorithms described in (Bryant 1977, Chandy and Misra 1978) were perhaps the first synchronization algorithms to be developed. They assume the topology indicating which LPs send messages to which others is fixed and known prior to execution. It is assumed each LP sends messages with non-decreasing time stamps, and the communication network ensures that messages are received in the same order that they were sent. This guarantees that messages arriving on each incoming link of an LP arrive in timestamp order. This implies that the timestamp of the last message received on a link is a lower bound on the timestamp of any subsequent message that will later be received on that link.

Messages arriving on each incoming link are stored in first-in-first-out order, which is also timestamp order because of the above restriction. Local events scheduled within the LP can be handled by having a queue within each LP that holds messages sent by an LP to itself. Each link has a clock that is equal to the timestamp of the message at the front of that link's queue if the queue contains a message, or the timestamp of the last received message if the queue is empty. The process repeatedly selects the link with the smallest clock and, if there is a message in that link's queue, processes it. If the selected queue is empty, the process blocks. The LP never blocks on the queue containing messages it schedules for itself, however. This protocol guarantees that each process will only process events in non-decreasing timestamp order.

Although this approach ensures the local causality constraint is never violated, it is prone to deadlock. A cycle of empty links with small link clock values (e.g., smaller than any unprocessed message in the simulator) can occur, resulting in each process waiting for the next process in the cycle. If there are relatively few unprocessed event messages compared to the number of links in the network, or if the unprocessed events become clustered in one portion of the network, deadlock may occur very frequently.

Null messages are used to avoid deadlock. A null message with timestamp T_{null} sent from LP_A to LP_B is a promise by LP_A that it will not later send a message to LP_B carrying a timestamp smaller than T_{null}. Null messages do not correspond to any activity in the simulated system; they are defined purely for avoiding deadlock situations. Processes send null messages on each outgoing link after processing each event. A null message provides the receiver with additional information that may be used to determine that other events are safe to process.

Null messages are processed by each LP just like ordinary non-null messages, except no activity is simulated by the processing of a null message. In particular, processing a null message advances the simulation clock of the LP to the time stamp of the null message. However, no state variables are modified and no non-null messages are sent as the result of processing a null message.

How does a process determine the timestamps of the null messages it sends? The clock value of each incoming link provides a lower bound on the timestamp of the next event that will be removed from that link's buffer. When coupled with knowledge of the simulation performed by the process, this bound can be used to determine a lower bound on the timestamp of the next *outgoing* message on each output link. For example, if a queue server has a minimum service time of T, then the timestamp of any future departure event must be at least T units of simulated time larger than any arrival event that will be received in the future.

Whenever a process finishes processing a null or non-null message, it sends a new null message on each outgoing link. The receiver of the null message can then compute new bounds on its outgoing links, send this information on to its neighbors, and so on. It can be shown that this algorithm avoids deadlock (Chandy and Misra 1978).

The null message algorithm introduced a key property utilized by virtually all conservative synchronization algorithms: *lookahead*. If an LP is at simulation time T, and it can guarantee that any message it will send in the future will have a time stamp of at least T+L regardless of what messages it may later receive, the LP is said to have a lookahead of L. As we just saw, lookahead is used to generate the time stamps of null messages. One constraint of the null message algorithm is it requires that no cycle among LPs exist containing zero lookahead, i.e., it is impossible for a sequence of messages to traverse the cycle, with each message scheduling a new message with the same time stamp.

2.1.2 Second Generation Algorithms

The main drawback with the null message algorithm is it may generate an excessive number of null messages. Consider a simulation containing two LPs. Suppose both are blocked, each has reached simulation time 100, and each has a lookahead equal to 1. Suppose the next unprocessed event in the simulation has a time stamp of 200. The null message algorithm will result in null messages exchanged between the LPs with time stamp 101, 102, 103, and so on. This will continue until the LPs advance to simulation time 200, when the event with time stamp 200 can now be processed. A hundred null messages must be sent and processed between the two LPs before the non-null message can be processed. This is clearly very inefficient. The problem becomes even more severe if there are many LPs.

The principal problem is the algorithm uses only the current simulation time of each LP and lookahead to predict the minimum time stamp of messages it could generate in the future. To solve this problem, we observe that the key piece of information that is required is the time stamp of the next unprocessed event within each LP. If the LPs could collectively recognize that this event has time stamp 200, all of the LPs could immediately advance from simulation time 100 to time 200. Thus, the time of the next event across the entire simulation provides critical information that avoids the "time creeping" problem in the null message algorithm. This idea is exploited in more advanced synchronization algorithms.

Another problem with the null message algorithm concerns the case where each LP can send messages to many other LPs. In the worst case, the LP topology is fully connected meaning each LP could send a message to any other. In this case, each LP must broadcast a null message to every other LP after processing each event. This also results in an excessive number of null messages.

One early approach to solving these problems is an alternate algorithm that allows the computation to deadlock, but then detects and breaks it (Chandy and Misra 1981). The deadlock can be broken by observing that the message(s) containing the smallest timestamp is (are) always safe to process. Alternatively, one may use a distributed computation to compute lower bound information (not unlike the distributed computation using null messages described above) to enlarge the set of safe messages.

Many other approaches have been developed. Some protocols use a synchronous execution where the computation cycles between (i) determining which events are "safe'" to process, and (ii) processing those events. It is clear that the key step is determining the events that are safe to process each cycle. Each LP must determine a lower bound on the time stamp (LBTS) of messages it might later receive from other LPs. This can be determined from a snapshot of the distributed computation as the minimum among:

- the simulation time of the next event within each LP if the LP is blocked, or the current time of the LP if it is not blocked, plus the LP's lookahead and
- the time stamp of any transient messages, i.e., any message that has been sent but has not yet been received at its destination.

A barrier synchronization can be used to obtain the snapshot. Transient messages can be "flushed" out of the system in order to account for their time stamps. If first-in-first-out communication channels are used, null messages can be sent through the channels to flush the channels, though as noted earlier, this may result in many null messages. Alternatively, each LP can maintain a counter of the number of messages it has sent, and the number if has received. When the sum of the send and receive counters across all of the LPs are the same, and each LP has reached the barrier point, it is guaranteed that there are no more transient messages in the system. In practice, summing the counters can be combined with the computation for computing the global minimum value.

To determine which events are safe, the *distance between LPs* is sometimes used. This "distance" is the minimum amount of simulation time that must elapse for an event in one LP to directly or indirectly affect another LP, and can be used by an LP to determine bounds on the timestamp of future events it might receive from other LPs. This assumes it is known which LPs send messages to which other LPs. Full elaboration this technique is beyond the scope of the present discussion, however, these techniques and others are described in (Fujimoto 2000).

Another thread of research in synchronization algorithms concerns relaxing ordering constraints in order to improve performance. Some approaches amount to simply

ignoring out of order event processing (Sokol and Stucky 1990, Rao, et al. 1998). Use of time intervals, rather than precise time stamps, to encode uncertainty of temporal information in order to improve the performance of time management algorithms have also been proposed (Fujimoto 1999a) (Beraldi and Nigro 2000). Use of causal order rather than time stamp order for distributed simulation applications has also been studied (Lee, et al. 2001).

2.2 Optimistic Synchronization

In contrast to conservative approaches that avoid violations of the local causality constraint, optimistic methods allow violations to occur, but are able to detect and recover from them. Optimistic approaches offer two important advantages over conservative techniques. First, they can exploit greater degrees of parallelism. If two events *might* affect each other, but the computations are such that they actually don't, optimistic mechanisms can process the events concurrently, while conservative methods must sequentialize execution. Second, conservative mechanism generally rely on application specific information (e.g., distance between objects) in order to determine which events are safe to process. While optimistic mechanisms can execute more efficiently if they exploit such information, they are less reliant on such information for correct execution. This allows the synchronization mechanism to be more transparent to the application program than conservative approaches, simplifying software development. On the other hand, optimistic methods may require more overhead computations than conservative approaches, leading to certain performance degradations.

The Time Warp mechanism (Jefferson 1985) is the most well known optimistic method. When an LP receives an event with timestamp smaller than one or more events it has already processed, it rolls back and reprocesses those events in timestamp order. Rolling back an event involves restoring the state of the LP to that which existed prior to processing the event (checkpoints are taken for this purpose), and "unsending" messages sent by the rolled back events. An elegant mechanism called anti-messages is provided to "unsend" messages.

An anti-message is a duplicate copy of a previously sent message. Whenever an anti-message and its matching (positive) message are both stored in the same queue, the two are deleted (annihilated). To "unsend" a message, a process need only send the corresponding anti-message. If the matching positive message has already been processed, the receiver process is rolled back, possibly producing additional anti-messages. Using this recursive procedure all effects of the erroneous message will eventually be erased.

Two problems remain to be solved before the above approach can be viewed as a viable synchronization mechanism. First, certain computations, e.g., I/O operations, cannot be rolled back. Second, the computation will continually consume more and more memory resources because a history (e.g., checkpoints) must be retained, even if no rollbacks occur; some mechanism is required to reclaim the memory used for this history information. Both problems are solved by *global virtual time* (*GVT*). GVT is a lower bound on the timestamp of any future rollback. GVT is computed by observing that rollbacks are caused by messages arriving "in the past." Therefore, the smallest timestamp among unprocessed and partially processed messages gives a value for GVT. Once GVT has been computed, I/O operations occurring at simulated times older than GVT can be committed, and storage older than GVT (except one state vector for each LP) can be reclaimed.

GVT computations are essentially the same as LBTS computations used in conservative algorithms. This is because rollbacks result from receiving a message or anti-message in the LP's past. Thus, GVT amounts to computing a lower bound on the time stamp of future messages (or anti-messages) that may later be received.

A pure Time Warp system can suffer from overly optimistic execution, i.e., some LPs may advance too far ahead of others leading to excessive memory utilization and long rollbacks. Many other optimistic algorithms have been proposed to address these problems. Most attempt to limit the amount of optimism. An early technique involves using a sliding window of simulated time (Sokol and Stucky 1990). The window is defined as [GVT, GVT+W] where W is a user defined parameter. Only events with time stamp within this interval are eligible for processing. Another approach delays message sends until it is guaranteed that the send will not be later rolled back, i.e., until GVT advances to the simulation time at which the event was scheduled. This eliminates the need for anti-messages and avoids cascaded rollbacks, i.e., a rollback resulting in the generation of additional rollbacks (Dickens and Reynolds 1990). A technique called direct cancellation is sometimes used to rapidly cancel incorrect messages, thereby helping to reduce overly optimistic execution (Fujimoto 1989, Zhang and Tropper 2001).

Another problem with optimistic synchronization concerns the amount of memory that may be required to store history information. Several techniques have been developed to address this problem. For example, one can roll back computations to reclaim memory resources (Jefferson 1990, Lin and Preiss 1991). State saving can be performed infrequently rather than after each event (Lin, et al. 1993, Palaniswamy and Wilsey 1993). The memory used by some state vectors can be reclaimed even though their time stamp is larger than GVT (Preiss and Loucks 1995).

Early approaches to controlling Time Warp execution used user-defined parameters that had to be tuned to optimize performance. Later work has focused on adaptive approaches where the simulation executive automatically monitors the execution and adjusts control parameters to maximize performance. Examples of such adaptive control

mechanisms are described in (Ferscha 1995, Das and Fujimoto 1997), among others.

Practical implementation of optimistic algorithms requires that one must be able to roll back all operations, or be able to postpone them until GVT advances past the simulation time of the operation. Care must be taken to ensure operations such as memory allocation and deallocation are handled properly, e.g., one must be able to roll back these operations. Also, one must be able to roll back execution errors. This can be problematic in certain situations, e.g., if an optimistic execution causes portions of the internal state of the Time Warp executive to be overwritten (Nicol and Liu 1997).

Another approach to optimistic execution involves the use of reverse computation techniques rather than rollback (Carothers, et al. 1999). Undoing an event computation is accomplished by executing the inverse computation, e.g., to undo incrementing a state variable, the variable is instead decremented. The advantage of this technique is it avoids state saving, which may be both time consuming and require a large amount of memory. In (Carothers, et al. 1999) a reverse compiler is described to automatically generate inverse computations.

2.3 Current State-of-the-Art

Synchronization is a well-studied area of research in the parallel discrete event simulation field. There is no clear consensus concerning whether optimistic or conservative synchronization perform better; indeed, the optimal approach usually depends on the application. In general, if the application has good lookahead characteristics and programming the application to exploit this lookahead is not overly burdensome, conservative approaches are the method of choice. Indeed, much research has been devoted to improving the lookahead of simulation applications, e.g., see (Deelman, et al. 2001). Otherwise, optimistic synchronization offers greater promise. Disadvantages of optimistic synchronization include the potentially large amount of memory that may be required, and the complexity of optimistic simulation executives. Techniques to reduce memory utilization further aggravate the complexity issue.

Recently, synchronization algorithms have assumed an increased importance because of their use in the DoD High Level Architecture (HLA). Because the HLA is driven by the desire to reuse existing simulations, an important disadvantage of optimistic synchronization in this context is the effort required to add state saving and other mechanism to enable the simulation to be rolled back.

3 TIME PARALLEL SIMULATION

Time-parallel simulation methods have been developed for attacking specific simulation problems with well-defined objectives, e.g., measuring the loss rate of a finite capacity queue of an ATM multiplexer. Time-parallel algorithms divide the simulated time axis into intervals, and assign each interval to a different processor. This allows for massively parallel execution because simulations often span long periods of simulated time.

A central question that must be addressed by time-parallel simulators is ensuring the states computed at the "boundaries" of the time intervals match. Specifically, it is clear that the state computed at the end of the interval $[T_{i-1},T_i]$ must match the state at the beginning of interval $[T_i,T_{i+1}]$. Thus, this approach relies on being able to perform the simulation corresponding to the ith interval without first completing the simulations of the preceding (i-1, i-2, ... 1) intervals.

Because of the "state-matching" problem, time-parallel simulation is really more of a methodology for developing massively parallel algorithms for specific simulation problems than a general approach for executing arbitrary discrete-event simulation models on parallel computers. Time-parallel algorithms are currently not as robust as space-parallel approaches because they rely on specific properties of the system being modeled, e.g., specification of the system's behavior as recurrence equations and/or a relatively simple state descriptor. This approach is currently limited to a handful of applications, e.g., queuing networks, Petri nets, cache memories, and multiplexers in communication networks. Space-parallel simulations offer greater flexibility and wider applicability, but concurrency is limited to the number of logical processes. In some cases, both time and space-parallelism can be used.

One approach to solving the state matching problem is to have each processor guess the initial state of its simulation, and then simulate the system based on this guessed initial state (Lin and Lazowska 1991). In general, the initial state will not match the final state of the previous interval. After the interval simulators have completed, a "fix-up" computation is performed to account for the fact that the wrong initial state was used. This might be performed, for instance, by simply repeating the simulation, using the final state computed in the previous interval as the new initial state. This "fix-up" process is repeated until the initial state of each interval matches the final state of the previous interval. In the worst case, N such iterations are required when there are N simulators. However, if the final state of each interval simulator is seldom dependent on the initial state, far fewer iterations will be needed.

In (Heidelberger and Stone 1990) the above approach is proposed to simulate cache memories using a least-recently-used replacement policy. This approach is effective for this application because the final state of the cache is not heavily dependent on the cache's initial state. A variation on this approach devised in the context of simulating statistical multiplexers for asynchronous transfer mode (ATM) switches precomputes certain points in time where one can guarantee that a buffer overflow (full

queue) or underflow (empty queue) will occur (Fujimoto, et al. 1995). Because the state of the system, namely, the number of occupied buffers in the queue, is known at these points, independent simulations can be begun at these points in simulated time, thereby eliminating the need for a fix-up computation.

Another approach to time-parallel simulation is described in (Greenberg, et al. 1991). Here, a queuing network simulation is expressed as a set of recurrence equations that are then solved using well-known parallel prefix algorithms. The parallel prefix computation enables the state of the system at various points in simulated time to be computed concurrently. Another approach also based on recurrence equations is described in (Baccelli and Canales 1993) for simulating timed Petri nets.

4 DISTRIBUTED VIRTUAL ENVIRONMENTS

While the foundation for parallel discrete event simulation lies in early research concerning synchronization algorithms, early work in DVEs came from the SIMNET project that demonstrated the viability of interconnecting autonomous simulators in a distributed environment for military training exercises (Miller and Thorpe 1995). SIMNET was used as the basis for the initial DIS protocols and standards, and many of the fundamental principles defined in SIMNET remain in DIS and the HLA today. SIMNET realized over 250 networked simulators at 11 sites in 1990.

From a model execution standpoint, a DIS exercise can be viewed as a collection of autonomous virtual (manned training simulators), live (physical equipment), and constructive (wargaming simulators and other analytic tools) simulators, each generating its own representation of the battlefield from its own perspective. Each simulator sends messages, called *protocol data units* (*PDUs*), whenever its state changes in a way that might affect another simulator. Typical PDUs include movement to a new location, firing at another simulated entity, changes in its appearance to other simulators (such as rotating the turret of a tank), etc.

In order to achieve interoperability among separately developed simulators, a set of standards have been developed (IEEE Std 1278.1-1995 1995). The standards specify the format and contents of PDUs exchanged between simulators as well as when PDUs should be sent.

DIS is based on the following underlying design principles (DIS Steering Committee 1994):

- *Autonomy of simulation nodes*. Autonomy facilitates the development, integration of legacy simulators, and simulators joining or leaving the exercise while it is in progress. Each simulator advances simulation time according to a local real-time clock. Simulators are *not* required to determine which other simulators must receive PDUs; rather, PDUs are broadcast to all simulators and the receiver must determine those that are relevant to its own virtual environment.
- *Transmission of "ground truth" information*. Each node sends absolute truth about the state of the entities it represents. Degradations of this information (e.g., due to environmental effects or sensor limitations) are performed by the receiver.
- *Transmission of state change information only*. To economize on communications, simulation nodes only transmit changes in behavior. If a vehicle continues to "do the same thing" (e.g., travel in a straight line with constant velocity), the rate at which state updates are transmitted is reduced. Simulators do transmit "keep alive" messages, e.g., every five seconds, so new simulators entering the exercise can include them in their virtual environment.
- *Dead Reckoning Algorithms*. All simulators use common algorithms to extrapolate the current state (position) of other entities between state updates. More will be said about this later.
- *Simulation time constraints*. Because humans cannot distinguish differences in time less than 100 milliseconds, a communication latency of up to this amount is required. Lower latencies are needed for other, non-training, simulators, e.g., testing of weapons systems.

4.1 Dead Reckoning

DIS simulations use a technique called *dead-reckoning* to reduce interprocessor communication to distribute position information. This reduction is realized by observing that rather than sending new position coordinates of moving entities at some predetermined frequency, processors can estimate the location of other entities through a local computation.

In principal, one could duplicate a remote simulator in the local processor so that any dynamically changing state information is readily available. This local computation, when applied to computing position information of moving entities, is referred to as the *dead-reckoning model (DRM).*

In practice, the DRM is only an approximation of the true simulator. An approximation is used because (1) the DRM does not receive inputs received by the actual simulator, e.g., a pilot using a flight simulator decides to travel in a new direction, and (2) to economize on the amount of computation required to execute the DRM. In practice, the DRM is realized as a simplified, lower fidelity version of the true model. To limit the amount of error between the true model and the DRM, the true simulator maintain its own copy of the DRM to determine when the divergence between them has become too large, In other words, the difference between the true position and the dead-reckoned position exceeds some threshold. When this occurs, the true simulator transmits new, updated information (the true posi-

tion) to reset the DRM. To avoid jumps in the display when the DRM is reset, simulators may realize the transition to the new position as a sequence of steps (Fujimoto 2000).

A variety of dead reckoning techniques have been proposed. Standard techniques are usually based on location, velocity, and acceleration of the moving object. Prediction based on history is described in (Singhal and Cheriton 1995). Group dead reckoning is used in (Das, et al. 1997).

4.2 Data Distribution (DD)

An important question concerns scaling exercises to include more entities and sites (locations). Significant changes to DIS are required to enable simulations of this size, particularly with respect to the amount of communications that are required.

Even with dead-reckoning, the DIS protocol described above does not scale to such large simulations. An obvious problem is the reliance on broadcasts. There are two problems: (1) realization of the communication bandwidth needed to perform broadcasts, is costly, and (2) the computation load required to process incoming PDUs is excessive and wasteful, particularly as the size of the exercise increases because a smaller percentage of the incoming PDUs will be relevant to each simulator.

Whenever a simulator performs some action that may be of interest to other simulators, e.g., moving an entity to a new location, a message is generated. Some means is required to specify which other simulators should receive a copy of this message. Specifically, the distributed simulation system must provide mechanisms for the simulators to describe both the information it is producing, and the information it is interested in receiving. Based on these specifications, the executive must then determine which simulators should receive what messages.

Data distribution has some similarities to Internet newsgroups. Specifically, newsgroup users must express what information they are interested in receiving by subscribing to specific newsgroups. The contents of the information that is being published is described by the newsgroup(s) to which it is sent, e.g., a recipe for a new cake would be published to a cooking newsgroup, not one concerning the weather. The newsgroup names are critical because they provide a common vocabulary for users to characterize both the information being published, and the information they are interested in receiving.

The set of newsgroup names defines a *name space,* i.e., a common vocabulary used to describe data and to express interests. Each user provides an *interest expression* that specifies a subset of the name space, i.e., a list of newsgroups, that indicate what information he is interested in receiving. A *description expression*, again a subset of the name space, is associated with each message that describes the contents of the message. Logically, the software managing the news groups matches the description expression of each message with the interest expression of each user. If the two overlap, i.e., have at least one element of the name space in common, the message is sent to that user.

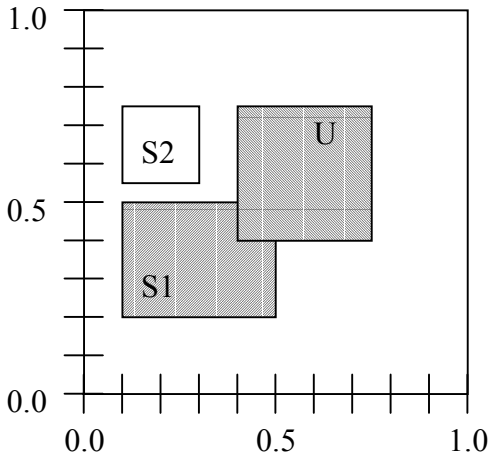

Figure 1: HLA DDM example.

The name space, interest expressions, and description expressions define the heart of the interface to the DD mechanisms. The DD software must map this interface to the primitives provided by the communication facilities such as joining, leaving, and sending messages to multicast groups. The challenging aspect of the DD interface is defining abstractions that are both convenient for the modeler to use, and provide an efficient realization using standard communication primitives. DD interfaces that are similar to basic communications primitives lend themselves to straightforward implementation, but may be difficult for modelers to use. On the other hand, higher level mechanisms such as "I am interested in receiving position updates for all tanks with a 2.0 radius circle of my current position" are more difficult to implement, leading to slow and/or inefficient mechanisms.

4.3 Data Distribution in the HLA

To illustrate these concepts, consider the data distribution mechanisms provided in the High Level Architecture. The HLA Interface Specification includes two sets of services to implement data distribution: declaration management and data distribution management. Declaration management services use a class-based approach. This means the federation defines a set of objects according to a class hierarchy, and individual federates may subscribe to receive updates to object attributes of specific classes. For example, a simulator might specify that it wishes to receive a message whenever the position attribute of any tank object (object declared from the tank class) is updated. This approach is static in the sense that interest expressions are based on classes that are statically defined. One could not, for instance, use these services to get updates for tank objects that are "close by" because the

position of other tanks relative to one's current position is not known until during the execution.

The data distribution management (DDM) services provide a means for providing this capability. The name space for the HLA DDM services is called a routing space. Routing spaces are an abstraction defined separately from objects and attributes, solely for the purpose of data distribution. A routing space is a multidimensional coordinate system. The name space for a single N-dimensional routing space is a tuple (X_1, X_2, ... X_N) with $X_{min} \leq X_i \leq X_{max}$, where X_{min} and X_{max} are federation-defined values. For example, Figure 1 shows a two-dimensional routing space with axis values ranging from 0.0 to 1.0. The relationship of the routing space to elements of the virtual environment is left to the federation designers. For example, a two dimensional routing space might be used to represent the geographical area covered by the virtual environment, however, the data distribution software is not aware of this interpretation.

Interest and description expressions in the HLA define areas called *regions*, of a routing space. Specifically, each region is a set of one or more *extents*, where each extent is a rectangular N-dimensional area defined within the N-dimensional routing space. Four extents are shown in Figure 1. Each extent is specified as a sequence of N ranges (R_1, R_2, ... R_N) where range R_i is an interval along dimension *i* of the routing space. For example, the extent labeled S1 in Figure 1 is denoted ([0.1,0.5], [0.2,0.5]), using the convention that R_1 corresponds to the horizontal axis, and R_2 corresponds to the vertical axis.

A region is the union of the set of points in the routing space covered by its extents. Interest expressions are referred to as *subscription regions*, and description expressions are referred to as *publication regions*. For example, the routing space in Figure 1 includes one update region U and two subscription regions S1 and S2. The extents defining a single region need not overlap.

Each federate can qualify a subscription to an object class by associating a subscription region with the subscription, e.g., to only get updates for vehicles within a certain portion of the routing space. Similarly, an update region may be associated with each instance of an object. If a federate's subscription region for an object class overlaps with the update region associated with the instance of the object being modified, then a message is sent to the federate.

For example, suppose the routing space in Figure 1 corresponds to the geographic area (i.e., the playbox) of a virtual environment that includes moving vehicles. Suppose the update region U is associated with an aircraft object that contains attributes indicating the aircraft's position. The region defined by U indicates the aircraft is within this portion of the playbox. Suppose S1 and S2 are the subscription regions created by two distinct federates F1 and F2, each modeling a sensor. The extents of these subscription regions are set to encompass all areas that the sensors can reach. If the aircraft moves to a new position within U, thereby updating its position attribute, a message will be sent to F1 because its subscription region S1 overlaps with U, but no message will be sent to F2 whose subscription region does not overlap with U.

Definition of subscription regions also involves certain compromises, particularly if the subscription region changes, as would be the case for a sensor mounted on a moving vehicle. Changing a subscription region can be a time consuming operation involving joining and leaving multicast groups. Defining large subscription regions will result in less frequent region modifications, but will result in the federate receiving more messages that are not relevant to it. Small regions yield more precise filtering, but more frequent changes. The region size should be set to strike a balance between these two extremes.

There has been much research in recent years focused on data distribution management techniques. Implementation of HLA DDM services involves defining a set of multicast groups, and mapping federates to these groups to define source/destination pairs. One implementation approach is to superimpose a grid over the routing space, and define a multicast group for each grid cell. Each federate must subscribe to the groups overlapping with that federate's subscription regions. A sender sends a message to each group corresponding to a grid cell overlapping the corresponding publication region.

Another approach is the region-based (or sender-based) implementation. Here, a multicast group is defined for each publication region. A sender sends a message to the group corresponding to the publication region associated with the send. Federates with subscription regions overlapping with the publication region are members of the group, and will each receive a copy of the message. A "matching" operation is required to determine group membership whenever publication or subscription regions change.

Early performance studies of the HLA and HLA-like DDM services are discussed in (Rak and Van Hook 1996, Cohen and Kemkes 1997). Performance of a hybrid grid approach using dynamic group assignments is described in (Boukerche, et al. 2000). An agent-based implementation approach is described in (Tan, et al. 2001).

5 SUMMARY

Parallel and distributed simulation technologies address issues concerning the execution of simulation programs on multiprocessor and distributed computing platforms. These technologies find applications in high performance computing contexts as well as in the creation of geographically distributed virtual environments. Originating in the 1970's, these remain active fields of research to this day.

We have given a brief introduction to this field by giving a sampling of some of the issues commonly addressed by researchers working in this area. Synchronization is a fundamental issue that has long been studied in the parallel

discrete event simulation field. A central issue in distributed virtual environments concerns efficient distribution of data, particularly for large DVEs.

REFERENCES

Baccelli, F. and M. Canales (1993). "Parallel Simulation of Stochastic Petri Nets Using Recurrence Equations." *ACM Transactions on Modeling and Computer Simulation* **3**(1): 20-41.

Beraldi, R. and L. Nigro (2000). Exploiting Temporal Uncertainty in Time Warp Simulations. *Proceedings of the 4th Workshop on Distributed Simulation and Real-Time Applications***:** 39-46.

Boukerche, A., et al. (2000). Dynamic Grid-Based Multicast Group Assignment tin Data Distribution Management. *Proceedings of the 4th Workshop on Distributed Simulation and Real-Time Applications*.**:** 47-54.

Bryant, R. E. (1977). Simulation of Packet Communication Architecture Computer Systems. *Computer Science Laboratory*. Cambridge, Massachusetts, Massachusetts Institute of Technology.

Carothers, C. D., et al. (1999). "Efficient Optimistic Parallel Simulation Using Reverse Computation." *ACM Transactions on Modeling and Computer Simulation* **9**(3).

Chandy, K. M. and J. Misra (1978). "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs." *IEEE Transactions on Software Engineering* **SE-5**(5): 440-452.

Chandy, K. M. and J. Misra (1981). "Asynchronous Distributed Simulation via a Sequence of Parallel Computations." *Communications of the ACM* **24**(4): 198-205.

Cohen, D. and A. Kemkes (1997). User-Level Measurement of DDM Scenarios. *Proceedings of the Spring Simulation Interoperability Workshop*.

Das, S. R. and R. M. Fujimoto (1997). "Adaptive Memory Management and Optimism Control in Time Warp." *ACM Transactions on Modeling and Computer Simulation* **7**(2): 239-271.

Das, T., et al. (1997). NetEffect: A Network Architecture for Large-Scale Multi-User Virtual Virtual Worlds. *Proceedings of the ACM Virtual Reality Software and Technology Conference***:** 157-163.

Deelman, E., et al. (2001). Improving Lookahead in Parallel Discrete Event Simulations of LArge-Scale Applications using Compiler Analysis. *Proceedings of the 15th Workshop on Parallel and Distributed Simulation***:** 5-13.

Dickens, P. M. and J. Reynolds, P. F. (1990). SRADS With Local Rollback. *Proceedings of the SCS Multiconference on Distributed Simulation*. **22:** 161-164.

DIS Steering Committee (1994). The DIS Vision, A Map to the Future of Distributed Simulation. Orlando, Florida, Institute for Simulation and Training.

Ferscha, A. (1995). Probabilistic Adaptive Direct Optimism Control iin Time Warp. *Proceedings of the 9th Workshop on Parallel and Distributed Simulation***:** 120-129.

Fujimoto, R. M. (1989). "Time Warp on a Shared Memory Multiprocessor." *Transactions of the Society for Computer Simulation* **6**(3): 211-239.

Fujimoto, R. M. (1999a). Exploiting Temporal Uncertainty in Parallel and Distributed Simulations. *Proceedings of the 13th Workshop on Parallel and Distributed Simulation:* 46-53.

Fujimoto, R. M. (1999b). Parallel and Distributed Simulation. *Proceedings of the Winter Simulation Conference*.

Fujimoto, R. M. (2000). *Parallel and Distributed Simulation Systems*, Wiley Interscience.

Fujimoto, R. M., et al. (1995). "Parallel Simulation of Statistical Multiplexers." *Journal of Discrete Event Dynamic Systems* **5**: 115-140.

Greenberg, A. G., et al. (1991). "Algorithms for Unboundedly Parallel Simulations." *ACM Transactions on Computer Systems* **9**(3): 201-221.

Heidelberger, P. and H. Stone (1990). Parallel Trace-Driven Cache Simulation by Time Partitioning. *Proceedings of the 1990 Winter Simulation Conference:* 734-737.

IEEE Std 1278.1-1995 (1995*). IEEE Standard for Distributed Interactive Simulation -- Application Protocols*. New York, NY, Institute of Electrical and Electronics Engineers, Inc.

Jefferson, D. (1985). "Virtual Time." *ACM Transactions on Programming Languages and Systems* **7**(3): 404-425.

Jefferson, D. R. (1990). Virtual Time II: Storage Management in distributed Simulation. *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing***:** 75-89.

Lee, B.-S., et al. (2001). A Causality Based Time Management Mechanism for Federated Simulations. *Proceedings of the 15th Workshop on Parallel and Distributed Simulation***:** 83-90.

Lin, Y.-B. and E. D. Lazowska (1991). "A Time-Division algorithm for Parallel Simulation." *ACM Transactions on Modeling and Computer Simulation* **1**(1): 73-83.

Lin, Y.-B. and B. R. Preiss (1991). "Optimal Memory Management for Time Warp Parallel Simulation." *ACM Transactions on Modeling and Computer Simulation* **1**(4).

Lin, Y.-B., et al. (1993). Selecting the Checkpoint Interval in Time Warp Simulations. *Proceedings of the 7th Workshop on Parallel and Distributed Simulation***:** 3-10.

Miller, D. C. and J. A. Thorpe (1995). "SIMNET: The Advent of Simulator Networking." *Proceedings of the IEEE* **83**(8): 1114-1123.

Nicol, D. M. and X. Liu (1997). The Dark Side of Risk. *Proceedings of the 11th Workshop on Parallel and Distributed Simulation***:** 188-195.

Palaniswamy, A. C. and P. A. Wilsey (1993). An Analytical Comparison of Periodic Checkpointing and Incremental State Saving. *Proceedings of the 7th Workshop on Parallel and Distributed Simulation***:** 127-134.

Preiss, B. R. and W. M. Loucks (1995). Memory Management Techniques for Time Warp on a Distributed Memory Machine. *Proceedings of the 9th Workshop on Parallel and Distributed Simulation***:** 30-39.

Rak, S. J. and D. J. Van Hook (1996). Evaluation of Grid-Based Relevance Filtering for Multicast Group Assignment. *Proceedins of the 14th DIS Workshop on Standards for the Interoperability of Distributed Simulations*. Orlando, FL.

Rao, D. M., et al. (1998). Unsynchronized Parallel Discrete Event Simulation. *Proceedings of the Winter Simulation Conference***:** 1563-1570.

Singhal, S. K. and D. R. Cheriton (1995). "Exploiting Position History for Efficient Remote Rendering in Networked Virtual Reality." *Presence* **4**(2): 169-193.

Sokol, L. M. and B. K. Stucky (1990). MTW: Experimental Results for a Constrained Optimistic Scheduling Paradigm. *Proceedings of the SCS Multiconference on Distributed Simulation*. **22:** 169-173.

Tan, G., et al. (2001). An Agent-based DDM for High Level Architecture. *Proceedings of the 15th Workshop on Parallel and Distributed Simulation***:** 75-82.

Zhang, J. L. and C. Tropper (2001). The Dependence List in Time Warp. *Proceedings of the 15th Workshop on Parallel and Distributed Simulation***:** 35-45.

AUTHOR BIOGRAPHY

RICHARD M. FUJIMOTO is a professor with the College of Computing at the Georgia Institute of Technology. He received the Ph.D. and M.S. degrees from the University of California (Berkeley) in 1980 and 1983 (Computer Science and Electrical Engineering) and B.S. degrees from the University of Illinois (Urbana) in 1977 and 1978 (Computer Science and Computer Engineering). He has been an active researcher in the parallel and distributed simulation community since 1985 and has published numerous papers on this subject. He has given several tutorials on parallel and distributed simulation at leading conferences. He has co-authored a book on parallel processing and recently completed a second on parallel and distributed simulation. He served as the technical lead in defining the time management services for the DoD High Level Architecture (HLA). Fujimoto is an area editor for ACM Transactions on Modeling and Computer Simulation. He also served as chair of the steering committee for the Workshop on Parallel and Distributed Simulation, (PADS) from 1990 to 1998 as well as the conference committee for the Simulation Interoperability workshop (1996-97).