MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Laboratory for Computer Science

Computation Structures Group Memo 182

Simulation on a Distributed System

by

Randal E. Bryant

This paper will appear in the Proceedings of the First International Conference on Distributed Systems.

July 1979

Simulation on a Distributed System[a]

Randal E. Bryant

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts

Abstract -- The simulation of a discrete event system naturally decomposes into a set of concurrent processes with each process simulating the behavior of one system element. A distributed system could best exploit this concurrency if the processes interact only by message-passing, and the control of the simulation is decentralized. In this paper a simulation method is developed in which the processes simulate the interactions of the system elements by sending *stimulus messages* to one another and coordinate their activities with decentralized control. Into the process programs. The control does not place any real-time speed constraints on the computation or communication and does not require any further communication links between processes. Some of the concepts developed here could be applied to other types of distributed systems as well.

Introduction

Traditionally, discrete event systems have been simulated on a single, sequential computer even though these systems often consist of a number of concurrent, interacting elements. Much effort has been expended in developing techniques of interleaving the simulations of the individual elements to achieve apparent concurrency on a single machine. Simulation languages such as Simula[4,12] allow the programmer to specify the system in terms of a number of concurrent, synchronized processes, leaving the task of correctly interleaving the process executions to the compiler and the run-time system. This approach must slow down as the number of events to be simulated grows, because a single processor must simulate the behavior of every element as well as control the sequencing of operations.

(a) This research was conducted under a graduate fellowship from the National Science Foundation. Additional funding was supplied by the National Science Foundation under grant DCR75-04060, and by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract no. N00014-75-C-0661.

The advent of distributed systems consisting of a number of autonomous, communicating processors provides an opportunity to increase the speed of simulation by exploiting the inherent concurrency of discrete event simulations. Rather than being constrained to a minimum space -- maximum time implementation on a single processor, the number of processors in the simulation facility can be traded-off against the time required for the simulation. The natural modularity and concurrency of simulation provides an ideal application for distributed systems.

This paper describes one possible approach to defining and implementing discrete event simulations on a distributed computer system. In this approach the simulation is carried out by a set of autonomous, message-passing processes. The proper sequencing of event simulations is maintained by two decentralized, speed-independent control methods. The first, called *time incrementation*, requires only small additions to the process programs and guarantees that each process will simulate its events in the proper order. The second, called *time acceleration*, requires a static analysis of the system's interconnection structure and more additions to the process programs. Time acceleration speeds up the passage of simulation time in idle sections of the simulation and, when all events have been simulated, causes the simulation time to be accelerated to infinity thus allowing the simulation to terminate. In the following presentation only the basic concepts are described with few assumptions about the types of systems to be simulated. Many refinements could be added to improve the efficiency for particular applications.

Several groups have developed methods for performing simulations on distributed systems. The author presented the basic model and control methods described here in the context of simulating concurrent, modular computer systems.[1] Chandy and Misra independently developed a similar simulation model and the time incrementation method of control.[2,3] Peacock, Wong, and Manning present an overview of the subject along with several centralized and decentralized control methods.[11] Others have developed methods based on a more centralized control.[10]

In addition, a method for terminating general distributed systems has been developed[5] based on Hoare's Communicating Sequential Process model.[7] This technique bears many similarities to the method presented here for terminating the simulation by accelerating the simulation time to infinity. Their method, however, requires a freezing of system activity while the termination condition is being tested, whereas the method presented here allows normal system activity during a test.

Computational Model

As an abstract view of the operation of a distributed system we shall assume the computer system supports a set of sequential processes which communicate only by sending messages to one another. Processes cannot exert direct control over one another, nor can they directly access information outside of their own local states. Process programs will be presented in PASCAL,[8] extended with the constructs send and receive.

When a process executes the procedure send(d,m), it will cause the message m to eventually be placed in the input buffer of process d. The sending process can proceed without waiting for the message to be received. When a process executes the function receive, it suspends operation until a message is present in the input buffer. Then the first message in the buffer is removed and returned as the function value. If a process sends two messages to another process, they will be received in the order sent. Messages sent from two different processes, however, can in general arrive in any order.

The computational model abstracts away the times at which statements are actually executed. The relative order of statement executions is constrained only by the sequential nature of the processes; the fact that a message cannot be received before it is sent; and the first-in, first-out property of the communication links from one process to another.

Other computational models based on message-passing processes have been proposed,[6,7,9] although they differ in their timing assumptions and control structures.

The computational model permits a wide variety of physical realizations, ranging from a single processor that supports message-passing processes to a distributed system with each process mapped onto a separate processor. Furthermore, the speed-independence of the model simplifies the incorporation of fault detection and recovery mechanisms in the physical realization.

Simulation Model

Description

The means by which the computer system models the behavior of a physical system is called the *simulation model*. In the following presentation the physical system is modeled with a set of processes, where each process simulates both the functional and time behavior of a system element. The time behavior is simulated *explicitly* by computing and recording the times at which events would occur, rather than *implicitly* by running the simulation at a speed proportional to the speed of the physical system. This computed time measure will be called *simulation time*.

Processes simulate the interactions of the physical system elements by sending *stimulus messages* to one another, where a stimulus has three characterizing features:

- the identity of the receiving process,
- the nature of the stimulus, and
- the simulation time at which the stimulus would be received by the corresponding physical element.

The basic activity simulated by a process is called an *event* and consists of three steps:

1. A stimulus message is received;
2. a new state of the process is computed based on the old state and the nature and simulation time of the stimulus; and
3. some number (possibly zero) of stimulus messages are sent to other processes and possibly to the sending process itself.

The simulation time of an event equals the simulation time of the input stimulus and takes place in zero simulation time. Events must be simulated in chronological order, i.e. in order of their simulation times. If two stimulus messages for a process have the same simulation time, a choice function arbitrates the order in which the events are simulated. The simulation time of a stimulus message must be a real number greater than or equal to the simulation time of its creating event.

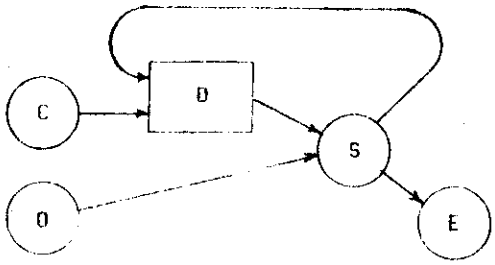
We shall further restrict the simulation model to closed, statically-structured systems. In a closed system processes can interact only with other processes. Input to the system is provided by *source* processes which generate sequences of stimulus messages while receiving no input messages. In a statically-structured system, all processes conceptually exist for all time. Practically speaking, however, a process can be terminated once the system reaches a condition where the process will receive no more stimulus messages, and all events for which it has received stimulus messages have been simulated. Nonetheless, the number and structure of processes must be fixed in advance.

The simulation model resembles the model provided by the Simula programming language, except that interaction between processes occurs by message-passing rather than through shared accesses to global variables and calls to a centralized scheduler. This change in orientation will allow the simulation to exploit the power of distributed systems.

Example

The system shown in Figure 1 will serve as an example for the remainder of the presentation. The system contains two source processes: customer source C and object source O. C generates two customers a and b with simulation times 5 and 8. O generates three objects o1, o2, and o3 with simulation times 0, 20, and 1000. Customers are sent via a delay process D to a server process S. The delay process always delays customer a by 10 time units and by b by 5. The objects are sent directly to the server. The server services a customer with an object and sends the customer first back to the delay process for one more cycle and then to an exit process E. The input and output stimulus messages for each event simulation are shown in Figure 2.

Figure 1. Structure of Simulation Example



C: Customer Source S: Server
O: Object Source E: Exit
D: Delay

Figure 2. Events in Simulation Example

Input Stimulus Messages	Output Stimulus Messages
process C:	
-	<D, a, 5.>
	<D, b, 8.>
process O:	
-	<S, o1, 0.>
	<S, o2, 20.>
	<S, o3, 1000.>
process D:	
<D, a, 5.>	<S, a, 15.>
<D, b, 8.>	<S, b, 13.>
<D, b, 13.>	<S, b, 18.>
<D, a, 20.>	<S, a, 30.>
process S:	
<S, o1, 0.>	-
<S, b, 13.>	<D, b, 13.>
<S, a, 15.>	-
<S, b, 18.>	-
<S, o2, 20.>	<D, a, 20.>
<S, a, 30.>	-
<S, o3, 1000.>	<E, b, 1000.>
process E:	
<E, b, 1000.>	-

Stimulus messages are listed as <receiver, nature, time>

Implementation

Basic Activities

Each process must both simulate the behavior of a physical system element and perform the necessary control operations to preserve a proper sequencing of event simulations.

We will assume each process program contains a subprogram

```
function simulate(s: stimulus) : stimlist
```

which implements the basic event simulations of the process. This function takes a stimulus message as the input argument, computes the new state (which is stored as global variables of the process program,) and returns a linked list of output stimulus messages. A skeletal program for a process is shown in Figure 3. Phrases enclosed in brackets « » represent informally specified sections of the program to be expanded as the development proceeds.

Figure 3. Process Program Showing Basic Activities

```
program simprocess;
begin
                .
                .
                .
  while not « termination condition » do
  begin
    while eventlist <> nil
        and « event can be safely simulated » do
    begin
      s := « next stimulus message on eventlist »;
      outlist := simulate(s);
      while outlist <> nil do
      begin
        mo := « next message on outlist »;
        if mo.process = ownprocess
          then schedule(eventlist, mo)
          else send(mo.process, mo)
        end
      end;
    end; {basic simulation}
                .
                .
                .
    mi := receive;
    if mi.type = 'stim'
        then schedule(eventlist, mi)
        else « perform control operations »;
                .
                .
                .
  end {main loop}
end. {simprocess}
```

In the program the input stimulus messages are stored in a linked list eventlist ordered by simulation time. This event list should not be confused with the global event list of a sequential simulator. It contains only the events for a single process. In the main loop of the program, the input stimulus messages for all events which can be safely simulated are removed and simulated. After an event simulation the newly generated stimulus messages are sent to their destination processes either by executing the command send or, if the destination is the process itself, by inserting it directly into the event list with the procedure schedule. This procedure inserts messages by simulation time and arbitrates the order of stimulus messages with the same simulation time. Once no more events can be simulated, a new input message is accepted. If the new message contains a stimulus, it is inserted into the event list. Otherwise the message contains control information and requires control operations. In either case the new message may enable more event simulations; hence the main loop is repeated until a termination condition is reached.

Control Operations

The skeletal program shows only the basic simulation activities and the "local" control where the incoming stimulus messages are kept in a chronological list. We must now fill out the program with "interprocess" control to allow the process to determine which events can be safely simulated and when the termination condition is satisfied. Without this control a process may first simulate one event and then receive the stimulus message for an event with an earlier simulation time, because stimulus messages need not arrive at a process ordered by simulation time. The combination of local and interprocess control will ensure that each process simulates its events in chronological order.

The simplicity of the computational model limits the variety of techniques which can be employed. In particular, the control operations must operate independently of the actual times at which events are simulated. Instead, we must exploit the sequencing constraints of the computational model and the timing characteristics of the simulation model to achieve a proper sequencing of event simulations. Furthermore, the control operations will be decentralized and incorporated directly into the process programs to prevent a tight synchronization of the processes.

The ideal control method would be simple and general, would require limited amounts of communication and limited connectivity between processes (i.e. which processes may send messages to which), yet would only prevent an event from being simulated when absolutely necessary. These goals, however, conflict with one another. One can at best achieve a balance between them.

Time Incrementation

The first control method, called *time incrementation*, provides great simplicity and generality, at the cost of potentially high amounts of communication (although with limited connectivity requirements) and potentially poor efficiency. Nonetheless, it does fulfill the basic requirements of the control operations.

With this method, every process maintains an asynchronous "clock" indicating the earliest possible simulation time of any unreceived stimulus messages for the process. Any events with simulation times less than the clock can be simulated. Every process sends information about its clock to those processes to which it may potentially send stimulus messages. As a result, the process clocks keep advancing until all events are simulated. The details of the method are described below.

Each process P_i can potentially receive stimulus messages from some subset inprocess_i of processes (not including itself) and send stimulus messages to some subset outprocess_i of processes (not including itself.) We will say there is a *link* from P_i to P_j if P_i is in inprocess_j (or equivalently if P_j is in outprocess_i.) Before the process can simulate an event with simulation time t, it must determine that no further stimulus messages with simulation time less than or equal to t will be received from a process in inprocess_i.

Each process P_i maintains a time counter intime_{ij} for each process P_j in inprocess_i. This time counter indicates the earliest possible simulation time of any stimulus messages from P_j which have not yet been received. The process uses these time counters to maintain its clock:

$$\text{clock}_i = \min_j(\text{intime}_{ij}).$$

Processes communicate their estimates of the time counters in the form of *increment* messages.[a] When process P_i receives an increment message from P_j, it updates intime_{ij} with the new estimate and recomputes clock_i. If clock_i has advanced, any events with simulation times less than clock_i are simulated and any newly generated stimulus messages are sent. Then an increment message containing the value $\text{clock}_i + \text{delay}_i$ is sent to every process in outprocess_i, where delay_i is the minimum "delay" of the process, i.e. the smallest possible difference between the simulation time of a stimulus sent by P_i and the simulation time of the event which creates this stimulus. We rely on the first-in, first-out property of communication

(a) Note that unlike other applications of time incrementation,[1,2,3,11] the simulation times of the stimulus messages cannot be used for updating the time counters, because a process in our simplified model may not send stimulus messages in chronological order. This occurs with process D of the example system.

from one process to another to prevent an increment message from overtaking an earlier stimulus message. During the simulation, processes keep sending increasing time estimates to one another and keep advancing their clocks until all events are simulated.

A process program including time incrementation operations is shown in Figure 4. For the sake of brevity, some sections of the program are given informally. This program utilizes a procedure sendall to send a message to all processes in a set and a function submin to compute the minimum of some subset of values in an array.

The additional factor $delay_j$ is added to increase the asynchrony of process clocks for processes connected in a cycle. A "cycle" can be defined as follows. Let the *interconnection graph* G be defined as the directed graph with a node for each process in the system and an edge from node i to node j if there is a link from P_i to P_j. That is, P_i may potentially send a message containing a stimulus to P_j. Then we say that two processes are contained in a cycle if the corresponding nodes in G are contained in a cycle. If two processes are contained in a cycle, their clocks cannot differ by more than the sum of the delay times of the processes in the cycle. Hence, to maximize the asynchrony of the process clocks one should make the strongest estimates of the delay times possible. Furthermore, to avoid deadlocks, the sum of the delay times around every cycle must be greater than zero. That is, if a set of processes $P_1, \ldots, P_i$ form a cycle:

$$delay_1 + \ldots + delay_i > 0.$$

Otherwise, the clocks would become frozen at a single value with none of them allowed to advance. Hence, the delay times play a key role in advancing the clocks.

The simulation of a source process requires a different program, because these processes receive no messages but in one event create a set of stimulus messages for other processes. Following the sending of the stimulus messages, the process sends increment messages with simulation time "infinity" (a special value greater than any representable real number) to all processes to which it has links, indicating that no more stimulus messages will be sent. The program is shown in Figure 5. For efficiency reasons, the source process could send increment messages with intermediate time values within the stream of stimulus messages, enabling the receiving processes to start their simulations before the final infinity messages are received.

During the simulation of the example system, processes C and D would send stimulus messages followed by increment messages with time infinity to D and S, respectively. Suppose that delay for process D is set to 5 and for all others to 0. Then a sequence of increment messages would be sent around the cycle containing D and S, causing their clocks to be incremented by 5 each time. After 3 such increment messages S would simulate its first servicing of customer b. After 2 more it would simulate the first servicing of a, and after 198 more it would simulate the second servicing of b. Meanwhile process E would receive a sequence of increment messages from S and then a stimulus message containing customer b. E could simulate the exit event for b after receiving one more increment message. At this point all events have been simulated, but increment messages would still be sent around the cycle indefinitely, because the clocks in D and S would never reach infinity. This would also prevent E from ever reaching a termination condition.

Several characteristics of the time incrementation method are demonstrated by this example. When a process is contained in a cycle its clock is advanced in increments

Figure 4. Process Program with Time Incrementation

```
program simprocess;
begin
  clock := 0.0; oldclock := -1.0;
  for « all processes j  in inprocess » do
          intime[j] := 0.0;

  while  clock < infinity  do
  begin
    while eventlist <> nil and
      « time of first event on eventlist » < clock do
    begin
      « remove and simulate first event »;
      « send output stimulus messages »
    end;

{if clock was incremented send increment messages}
    if clock > oldclock
        then begin
          with mo do
          begin type := incr;
                source := ownprocess;
                time := clock + delay end;
          sendall(outprocess, mo)
          oldclock := clock;
        end;

    mi := receive;  {accept a new input message}
    case mi.type of
        stim: schedule(eventlist, mi);
        incr:
          begin
            intime[mi.source] := mi.time;
            clock :=
              max(clock, submin(inprocess, intime))
          end
    end {case}
  end {main loop}
end. {simprocess}
```

Figure 5. Program for a Source Process

```
program sourceprocess
begin
  outlist := simulate;
  « send output stimulus messages »;
  with mo do
    begin type := incr;  time := infinity end;
  sendall(outprocess, mo)
end
```

no greater than the sum of the delay times around the cycle. Hence, many increment messages may be sent around a cycle before an event can be simulated. Furthermore, the process can never reach the termination condition even after all event simulations have been completed. With time incrementation, processes send such limited control information to one another that they must make very weak estimates of their next event times.

Despite its limitations, the time incrementation control method fulfills its basic requirements. It has been shown that methods similar to the one described here guarantee that all events will eventually be simulated (as long as the sums of the delay times around every cycle are greater than zero) and that the events will be simulated in the proper order.[1,2]

Time Acceleration

We can overcome the shortcomings of time incrementation by incorporating an additional form of control operations into the processes. This control periodically determines the earliest possible simulation time of the next event for a group of processes connected in cycles and causes the clocks of all these processes to be "accelerated" ahead. When possible it will accelerate the simulation time to infinity and cause the processes to terminate. As with time incrementation, time acceleration is decentralized, speed-independent, and does not require new communication links between processes.

Time acceleration requires a static analysis of the interconnection structure of the processes. The interconnection graph G for the system defines a partition of the processes into a set of equivalence classes $C_1, \ldots, C_m$ corresponding to the strongly-connected components of G. Each class is either *singleton* or *cyclic*. A singleton class contains only one process which is not contained in any cyclic path. A cyclic class contains at least two processes (because G contains no self-loops) such that for any P_i, P_j in the class, there is a path from P_i to P_j and from P_j to P_i. Hence, the activities of process P_i can potentially affect P_j and *vice-versa*. In the example, processes C, D, and E each form singleton classes, while processes D and S form a cyclic class.

The difficulties of the time incrementation method lie in the cyclic classes. With time acceleration additional control operations are added to the process programs for each cyclic class. These operations remain internal to the class. Between processes in different classes, only stimulus and increment messages are sent. Hence, we need only describe the operations for a single cyclic class.

By the assumptions of the simulation model, no event can create a stimulus message with simulation time less than the simulation time of the event. If we could determine the simulation time of the next event (in simulation time) of all processes in a class, their clocks could be accelerated to this time. Thus the clocks of the processes in a cyclic class can be accelerated to the minimum of the following:

- The simulation times for all stimulus messages on the event lists of processes in the class,
- the simulation times of all stimulus messages being sent between processes in the class, and
- $intime_{ji}$ for any process P_i in the class and process P_j not in the class.

We want determine this simulation time, however, without freezing the process activities and without a centralized observer.

Instead, some process is chosen to periodically initiate a test by sending *test* messages to all processes in the class to which it has a link. These test messages are propagated by other processes in the class along the normal communication links, thus creating a "wave" of test messages which pass through every link between processes in the class and eventually return to the initiating process. Each returning test message contains the earliest simulation time of any potential event simulations it has encountered. The minimum of all returning test message simulation times is then computed, and a wave of *set* messages is initiated containing this value. The set messages pass through all processes in the class, causing their clocks to be advanced to the new value. This technique of sending test messages and set messages via the normal communication links exploits the first-in, first-out property of these links to prevent any stimulus messages in transit from one process to another from being overlooked.

To implement the time acceleration operations for a class, one of the processes in the class is arbitrarily chosen as the *test control* process P_c. This process controls only the initiation of the test and the analysis of the results, and not the activities of other processes. Figure 6 shows the program for the control process P_c. Note that in PASCAL, * denotes set intersection, and - denotes set difference. Those sections pertaining to the simulation and time incrementation activities are abbreviated by informal descriptions.

In the program the process decides whether to initiate a test before accepting the next input message. It can safely do so any time a test is not already in progress, although other limitations on the frequency may be desirable. The process initiates the test by sending test messages containing an estimate of the maximum allowable clock time to all processes in $outprocess_c \cap class$, where class is the set of processes in the class.

Once the test has been initiated, the program continues its normal simulation and time incrementation operations. Each time a stimulus or test message is received, the time estimate is updated. A fixed number, expected, of test messages will return to the control process during a test, as will be discussed later. Once they have all been received, the final estimate is used to update

Figure 6. Program for Time Acceleration Control Process

```
program controlprocess
begin
  « initialize values »
  testing := false;
  while clock < infinity do
    begin
    « perform all enabled event simulations »;
    « send stimulus messages »;
    if not testing
      then begin  {initiate test}
        t := submin{inprocess - class, intime};
        if eventlist <> nil then t := min{t,
          « time of first event on eventlist »};
        with mo do
          begin type := test; time := t end;
        sendall{outprocess * class, mo};
        count := 0;
        testing := true
      end;

    « if clock has been incremented,
        send increment messages »;
    mi := receive;  {accept a new message}
    case mi.type of
      stim: begin
              schedule{eventlist, mi};
              if testing then t:= min{t,mi.time}
            end;
      incr: « perform incrementation operations »;
      test: begin
              count := count + 1;
              t := min{t, mi.time};
              if count = expected
                then begin {test is completed}
                  with mo do begin type := set;
                                   time := t end;
                  sendall {outprocess * class, mo};
                  clock := max{clock, t};
                  testing := false;
                  « if clock = infinity,
                       send increment messages »
                end
            end;
      set: {ignore set messages}
    end {case}
  end {main loop}
end.
```

the clock and is sent in *set* messages to processes in $outprocess_C \cap$ class. The program takes the maximum of the old clock and the message contents, because the clock may have been incremented beyond the estimated value while the test was underway. Any returning set messages are ignored. Once the clock is accelerated to infinity the process sends increment messages with simulation time infinity to all processes in $outprocess_C$ - class.

Before developing the program for the other processes in the class, some further definitions are required. Let T be a subgraph of the system interconnection graph G which forms a directed spanning tree from the processes in the class to process P_C. T defines a unique path from every process in the class to P_C. Let $returnprocess_I$ denote the (unique) process P_J such that (I,J) is an edge in T. That is, $returnprocess_I$ is the first process in the sequence of links from P_I to P_C. This spanning tree will provide a means of directing the test messages back to the control process. The program for the remaining processes in a cyclic class is shown in Figure 7.

Figure 7. Program for Other Processes in a Cyclic Class

```
program otherprocess
begin
  « initialize variables »;
  testing := false;
  while clock < infinity do
  begin
    « perform all enabled event simulations »;
    « send stimulus messages »
    « if clock has been incremented,
        send increment messages »;
    mi := receive;  {accept a new message}
    case mi.type of
      stim: begin
              schedule{eventlist, mi};
              if testing then t:= min{t, mi.time}
            end;
      incr: « perform incrementation operations »;
      test: begin
              t := submin{inprocess - class, intime};
              t := min{t, mi.time};
              if eventlist <> nil then t :=  min{t,
                « time of first event on eventlist »};
              with mo do begin type := test;
                               time := t end;
              if not testing
                then begin  {first message of test}
                  testing := true;
                  sendall{outprocess * class, mo}
                end
                else send{returnprocess, mo}
            end;
      set:  if testing
              then begin  {first set message}
                clock := max{clock, mi.time};
                testing := false;
                sendall{outprocess * class, mi}
                « if clock = infinity,
                     send increment messages »
              end
    end  {case}
  end  {main loop}
end.
```

In this program, when a process P_I receives the first message for a test, it makes its own estimate of the maximum allowable clock time and sends test messages to all processes in the class to which it has links. During the test it continues with the normal simulation and time incrementation activities. When any further test messages are received they are passed on toward the control process via $returnprocess_I$. If, however, some stimulus message has been received since the start of the test, this message may have an earlier simulation time than the process' original clock estimate. Hence, the process may insert its own, more conservative estimate into the test message.

When the process receives its first set message after a test it updates its clock and passes the message to all processes in the class to which it has links. Any further set messages are ignored until after the next test.

During a test P_C initially creates $|outprocess_C \cap class|$ test messages, where $||$ denotes set size. Every other process P_I in the class sends $|outprocess_I \cap class|$ test messages upon receiving the first test message, and one test message for each test message received thereafter. Thus P_I "creates" $|outprocess_I \cap class| - 1$ new test messages. All test messages eventually return to the control process, hence P_C will receive exactly

$$expected = 1 + \sum\left(|outprocess_I \cap class| - 1 \right)$$

returning test messages during a test, where the sum is taken over all processes in the class.

The time incrementation and acceleration activities of the processes in our example cannot be predicted exactly, because they depend on the relative order of message arrivals. Suppose, however, that D is chosen as the control process for the class. D will first initiate a test with time 0 (due to a lack of sophistication in the program.) Indeed a series of tests may be performed with a resulting time of 0 until D and S receive increment messages with simulation time infinity from C and D. Then a series of incrementations and accelerations will cause the clocks in D and S to advance much more rapidly than before. Finally, once S simulates its second servicing of b, the next test will cause the clocks in D and S to advance to infinity. S will send an increment message with time infinity to E, and E will terminate.

The example simulation terminates, even though the system being simulated fails to terminate properly (i.e. with all customers passing to the exit process.) Similarly, the processes could simulate a deadlock without themselves deadlocking. Unlike physical time, simulation time is a purely synthetic quantity and can jump to infinity. The simulation has a "clairvoyance" which the physical system lacks.

It has been shown that a method similar to the one described here will cause a correct termination of a simulation.[1] That is, the simulation will not be terminated while there are still events to be simulated, but once every event has been simulated all processes will reach their termination condition. A generalization of this correctness proof to the time acceleration method would also require showing that the simulation time of a process will never be accelerated beyond the time of a possible stimulus.

Conclusion

The implementation of the simulation model demonstrates how a set of of cooperating process processes can through their actions and interactions perform the basic computations required to solve a problem and maintain a proper sequencing of these activities. They do not require intervention from a central controller, communication links between processes beyond those used to transmit normal data, nor any real-time speed of the computation or communication. Instead we exploit the characteristics of the problem to be solved and the sequencing constraints provided by the computational model to *coordinate* rather than to *synchronize* the process activities. These features allow the actual process executions to proceed with maximum asynchrony and with a minimum of interprocess communication, thus taking best advantage of the characteristics of distributed systems.

The three step implementation shown here provides a methodology for the design of concurrent systems. In the first step, the basic activities required to solve the problem are defined and mapped onto a set of processes. Next, a simple but adequate set of control operations is added. Finally, mechanisms to improve the efficiency of the control are added. This methodology results in a well-structured, efficient system and facilitates the verification of the system's correctness by decomposing the proof into a series of simpler steps.

With a simple decentralized control method such as time incrementation, the processes are typically provided with only minimal information about the states of other processes. As a result, they must act very cautiously with a consequent lack of efficiency. However, this minimal solution allows the designer to concentrate on the efficiency, rather than the adequacy of more sophisticated methods.

A more sophisticated control method, such as time acceleration, typically involves determining a more global condition of the processes, thereby allowing them to act more intelligently and efficiently. The time acceleration control method shows how such a global condition can be determined without freezing the other process activities and without a centralized observer which communicates directly with the processes. The design of this type of control method involves many subtleties, because many activities are proceeding concurrently. One could easily overlook an activity as it moves from one process to another while the test of the global condition is underway.

While the control methods described here have been presented entirely in the context of simulation, they could potentially be applied to other types of distributed systems. Many of the problems encountered in distributed systems, such as deadlock, inconsistent states, and nontermination, could be avoided by replacing real-time clocks with some purely synthetic time measure analogous to simulation time. The computation can then be viewed as the simulation of some virtual machine. For example, suppose we wish to halt the processes in a distributed system once all activity has ceased except for the "background" control operations. In most current systems *ad hoc* mechanisms are employed to do this (e.g. "pulling the plug,") but as distributed systems grow larger and more complex, more reliable methods will be required. The time acceleration method could serve this

purpose by causing the synthetic time to accelerate to infinity.

[1] Bryant, R. E., Simulation of Packet Communication Architecture Computer Systems, MIT LCS Technical Report TR-188, (Nov., 1977).

[2] Chandy, K. M., and J. Misra, Specification, Synthesis, Verification and Performance Analysis of Distributed Programs; A Case Study: Distributed Simulation, Report TR-86, University of Texas at Austin, Dept. of Computer Sciences, (Nov., 1978).

[3] Chandy, K. M., and J. Misra, "A Nontrivial Example of Concurrent Processing: Distributed Simulation," Proceedings of COMPSAC '78, IEEE (1978) 822-826.

[4] Dahl, O. J., and K. Nygaard, "Simula -- An Algol-Based Simulation Language," Communications of the ACM, 9-9 (Sept., 1966) 671-678.

[5] Francez, N. "On Achieving Distributed Termination," International Symposium on the Semantics of Concurrent Computation, IRIA (July, 1979).

[6] Hewitt, C., and R. Atkinson, "Parallelism and Synchronization in Actor Systems," Principles of Programming Languages, ACM, New York, (Jan., 1977), 267-280.

[7] Hoare, C. A. R., "Communicating Sequential Processes," Communications of the ACM, 21-8 (Aug., 1978), 666-677.

[8] Jensen, K. and N. Wirth, PASCAL User Manual and Report, 2nd ed., Springer-Verlag, New York, 1974.

[9] Kahn, G., and D. MacQueen, "Coroutines and Networks of Parallel Processes," Information Processing 77, IFIP, North Holland Publishing Company, Amsterdam (1977), 993-998.

[10] Kaubisch, W. H., and C. A. R. Hoare, Discrete Event Simulation Based on Communicating Sequential Processes, Internal Memo, Dept. of Computer Science, Queen's University, Belfast, N. Ireland (1978).

[11] Peacock, J. D., J. W. Wong, and E. Manning, "Distributed Simulation Using a Network of Microcomputers," Computer Networks 3-1 (Feb, 1979) 44-56.

[12] Ziegler, B. P., Theory of Modelling and Simulation, Wiley Interscience, New York (1976).