Proceedings of the 1989 Winter Simulation Conference
E.A. MacNair, K.J. Musselman, P. Heidelberger (eds.)

DISTRIBUTED SIMULATION USING HIERARCHICAL ROLLBACK

Richard L. Gimarc
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

ABSTRACT

As the complexity of computer and communication systems increases, it becomes increasingly difficult to construct and evaluate models of these systems which can be used to study their performance under varying conditions. A modeling technique which is once again gaining in popularity due to its generality and ability to represent systems in varying degrees of detail is discrete event simulation.

Constructing a simulation model of a real or proposed system is well understood and many tools are available. The major problem which arises is model evaluation. Due to the complexity and level of detail in a simulation model, it may require an excessive amount of time and computer resources to perform the evaluation. There is an area of current research, though, which is addressing these performance problems: distributed simulation.

Distributed simulation has the potential to reduce the often lengthy run–time required for complex discrete event simulation by using multiple, cooperating processors instead of a single processor for model execution. Also, the computer resource requirements can more easily be satisfied by spreading the demand across a set of processors rather than depending on a single processor to furnish the required resources.

In this paper a new technique for distributed simulation called *hierarchical rollback* is presented. Hierarchical rollback employs multiple processors for the evaluation of a single simulation model. Synchronization and coordination of the processors is based on a unique checkpoint/rollback mechanism. It will be shown how hierarchical rollback can be used to execute a hierarchical simulation model on a distributed set of processors.

1. INTRODUCTION

Distributed simulation (Misra 1986) has the potential to reduce the often lengthy run–time required for evaluating a complex discrete event simulation by using multiple, cooperating processors instead of a single processor. With n processors working in concert, distributed simulation has the potential to reduce run–time to 1/n–th of that of sequential simulation. Parallelism inherent in the simulation model may be exploited by the simultaneous execution of submodels on separate processors. Concurrent submodel execution has the potential to dramatically reduce the elapsed run–time.

Characteristics common to the distributed simulation techniques presented in the literature are

1. The simulation model is partitioned into disjoint submodels, and each submodel is assigned to a separate processor.
2. Each submodel has its own event list and simulation clock.
3. There is no shared memory.
4. Processors communicate exclusively using messages.
5. The set of processors (submodels) execute concurrently until simulation terminates.

The one characteristic which differs markedly between the various distributed simulation techniques is how they manage submodel interaction. If each submodel is logically isolated and never interacts with other submodels, then local submodel execution can proceed independently of other submodels. However, it is often the case that there is some degree of submodel interaction. For example, suppose submodel S(1) on processor P(1) generates an event which must be executed at time t by submodel S(2) on processor P(2). If S(2)'s local simulation time is greater than t when it receives the event, then S(2) will not be able to process the event directly since that would invalidate the simulation (by not processing events in chronological order). Thus, there must be some mechanism in place to ensure synchronization for interacting submodels.

Managing submodel interaction is the primary problem addressed by the different distributed simulation algorithms. There are two basic approaches that have appeared in the literature: conservative and optimistic.

The *conservative* approach (Peacock 1978) (Chandy 1979) (Chandy 1981) controls the local progress made by each submodel such that when an external event is received by a submodel, it is guaranteed that the event time is not in the receiving submodel's local past. Each submodel throttles its local progress based on the progress made by the submodels which could send it simulation events. This ensures that submodels will receive events to be executed in present or future time which is easily accomplished by posting the event on their event list.

In the *optimistic* approach (Jefferson 1985), submodel execution proceeds asynchronously on the (optimistic) assumption that it will not receive an event from another submodel. If submodel interaction does occur, a checkpoint/rollback mechanism is used to enable the received event to be executed at the proper time. Rollback requires the saving of previous submodel states (using checkpoints) so that local simulation can be backed up in time for proper event scheduling. The rollback mechanism is used instead of throttling local submodel progress to guarantee that events are executed in the proper order.

In this paper a new technique for distributed simulation called hierarchical rollback is presented. Hierarchical rollback is a hybrid scheme containing features common to both the conservative and optimistic approaches. Its goal is to meld these two approaches in such a way to maximize their strengths and minimize their deficiencies.

The next section describes the concept of a hierarchical simulation. Following that, hierarchical rollback is described.

2. HIERARCHICAL SIMULATION MODEL

A hierarchical simulation model is a special type of simulation model which exhibits a hierarchical structure of objects being modeled. A system called the Hierarchical Simulation System (HSS) has been developed which supports the construction and evaluation of hierarchical simulation models of office systems. Although the focus of HSS is on office systems, the concepts it employs can be applied to other application areas.

2.1 The Hierarchical Simulation System

The Hierarchical Simulation System (Gimarc 1985) is an interactive, object-oriented simulation modeling system which provides a complete environment for building and executing hierarchical simulation models of office systems. HSS embodies a modeling methodology which provides a framework which structures the perception and representation of office systems. This methodology enables an office planner to design and construct a simulation model which directly reflects the hierarchical structure of the office system being modeled. The hierarchical structure is then simulated by HSS.

There are two steps in the HSS modeling methodology:

Step 1: Decompose the office system into a hierarchical collection of information processing subsystems.
Step 2: Describe the information paths in the office system.

Information Processing Subsystems. An office system can be viewed as a hierarchical collection of information processing subsystems. At the highest level, the office system can be decomposed into first-level subsystems. This decomposition may, for example, be driven by business areas, physical location, or department boundaries. After the first-level subsystems have been identified, they may themselves be decomposed into lower-level subsystems. The result of this decomposition is a hierarchical collection of information processing subsystems. The decomposition of an office system into a hierarchy of information processing subsystems may be obtained in a manner analogous to the programming methodology of stepwise refinement described in (Wirth 1971).

The HSS modeling methodology enables the office planner to obtain a hierarchical description of an office system by a sequence of refinement steps. Initially the office system being modeled is viewed as a single entity. The first step in the refinement process is to identify the major subsystems of the office system. These major subsystems are then decomposed repeatedly until the components of the office system (workstations, LANs, file servers, etc.) appear as leaves on the refinement tree. The resulting tree structure is referred to as the HSS *model tree*.

The HSS model tree represents the hierarchical structure of the office system being modeled. Each subtree represents a logical subsystem. Unlike the program tree built during program design using stepwise refinement, the HSS model tree is seen directly in the HSS simulation model. All nodes in the model tree have a corresponding representation in the simulation model.

Information Paths. An office system is composed of both information processing equipment and the information that is processed by that equipment. Information in an office generally flows along fixed paths. That is, information originates at an information source, travels through various stages of information processing and finally reaches its destination. The source and destination can usually be identified as specific components in the office system. An information path is the sequence of office system components through which information flows. There may be several information paths within an office system, each characterized by the type of information that flows along the path. And further, information paths need not be disjoint.

To complete the design of an HSS model the information paths must be identified. The office system components which appear on these paths will be the leaf nodes of the HSS model tree.

HSS Simulation Model. An HSS simulation model may be constructed in a mechanical fashion directly from the design obtained by applying the HSS modeling methodology. The simulation model is constructed using the same two-step procedure applied during model design. Namely,

Step 1: Build the model tree.
Step 2: Build the information paths.

HSS provides a complete on-line environment for constructing a hierarchical simulation model. The user interface supports the top-down modeling methodology and provides a set of parameterized objects representing office system components.

3. HIERARCHICAL ROLLBACK: SYSTEM MODEL

The model tree derived from building a hierarchical simulation model defines the basic structure for the processors used in hierarchical rollback. The model tree is partitioned into subtrees, and each subtree is assigned to a processor. The *processor tree* will be used for distributed simulation with hierarchical rollback.

For processor P(i), the processor tree is defined by the following data structures:

P(i).parent	Name of the processor that is P(i)'s parent. If P(i) is the root node in the processor tree, then P(i).parent is NULL.
P(i).child(j)	Name of P(i)'s j-th child processor. If P(i) does not have a j-th child, then P(i).child(j) is NULL.

There is no shared memory in the processor tree. The only resource that is shared between processors are communication channels which link each processor to its parent and child processors.

3.1 Processor Communication

Processors communicate with each other exclusively via messages. Processor P(i) has communication channels attached to its parent and child processors, if they exist, in the processor tree. Thus, P(i) can only communicate with its immediate parent and child processors. To send a message to a processor outside of this neighbor set, P(i) must route the message through its parent or child processor, whichever is closer to the desired destination (measured in terms of the number of communication links traversed).

Assumptions. The following assumptions are made on the processor communication mechanism:

1. The communication channel is error-free.
2. Messages cannot be lost on the channel.
3. Messages are received in the order sent (the channel does not change message order).
4. Processor input buffers have infinite capacity (thus messages cannot be lost due to buffer overflow).

Input/Output Buffers. Each processor maintains separate input and output buffers. The input buffer holds messages received from other processors and the output buffer holds messages that must be transmitted to other processors. Messages are kept in ascending message time stamp order. For messages with identical time stamps, ordering is first-in-first-out.

The input and output buffers both have pointers which are used to indicate the next message to be processed. This mechanism allows the buffers to serve as historical repositories of messages that have already been processed. The input buffer pointer is incremented after a message from the buffer is processed. Likewise, the output buffer pointer is incremented after a message is transmitted.

Message Types. Every message m contains the following data fields:

m.sender	Name of the processor that will send message m.
m.receiver	Name of the processor that is to receive message m.
m.type	Message type: DATA, ACK, or TIME.
m.time	Message time stamp.
m.text	Arbitrary text, depends on message type.
m.send_id	Message sequence number.

DATA messages are used to transmit simulation events between processors. A description of the event is placed in m.text. Event time is placed in m.time and m.send_id is used to uniquely identify the message.

An ACK message is used to acknowledge the receipt of a DATA message. M.send_id is set to the sequence number of the DATA message being acknowledged. Since time is meaningless for an ACK message, m.time is set to zero, and likewise, m.text is set to NULL.

Processors exchange simulation time information using TIME messages. The appropriate time value is placed in m.time and fields m.text and m.send_id are set to NULL.

Communication Protocol. The communication protocol governing message transmission is very simple. DATA messages are assigned a message sequence number (m.send_id) which uniquely identifies the message. When processor P(i) transmits a DATA message to P(j), it expects to receive a corresponding ACK message with matching sequence number from P(j). The ACK indicates that the receiving processor has successfully received the message and scheduled the event it contains for processing. P(i) can have at most one outstanding unacknowledged DATA message sent to each receiving processor.

There are no protocol restrictions on TIME messages. They are never acknowledged and message receipt is assumed to be successful.

4. HIERARCHICAL ROLLBACK: COMPONENTS

Hierarchical rollback (HR) is a distributed simulation technique which uses a checkpoint/rollback mechanism to synchronize submodel interaction. The three key components of HR are time management, checkpoint processing, and rollback processing.

Time management is used to control the processing of events and submodel interaction. The algorithms used are fully distributed so that there is no single processor which is designated as the master time keeper.

Each processor has a checkpoint process which periodically saves copies of the processor's local state space in checkpoint records. These checkpoint records are utilized when a processor has to synchronize itself due to submodel interaction.

Rollback processing is the synchronization mechanism used by HR. With the aid of saved checkpoint records, a processor is able to step back its local simulation time in order to synchronize itself during submodel interaction.

These three components are discussed in detail in the following sections.

4.1 Time Management

Simulation time is maintained at three different levels: local, subtree, and global. Each processor (ie, submodel) has its own local simulation clock which corresponds to its local time. Processor P(i)'s local time is maintained in the following data structure:

P(i).local_time	Current value of P(i)'s local simulation clock.

As simulation events are executed within a processor, local simulation time increases. In the event of a rollback, a processor may have to step back its local time.

Subtree time is also maintained at the processor level and reflects the progress made by the subtree rooted at that processor. For example, if processor P(i)'s subtree time is t, then all processors within the subtree rooted at P(i) have successfully completed their local simulation up to time t.

The following two data structures are used to compute subtree time for processor P(i):

P(i).child_time(j)	Last known subtree time of P(i)'s j−th child processor.
P(i).subtree_time	P(i)'s current subtree time.

Prior to examining its input channels for messages, P(i) computes a temporary subtree time as follows:

temp_time = MIN(P(i).local_time,P(i).child_time(*))

If this new subtree time is not less than P(i).subtree time, then P(i) declares that it has a new subtree time by updating P(i).subtree_time with temp_time and sending a TIME message to P(i).parent with the new subtree time. When P(i)'s parent, say P(k), receives the TIME message, it will update P(k).child_time(i) with the new time.

A processor's current subtree time also controls the transmission of DATA messages up the processor tree. Processor P(i) may transmit a DATA message to P(i).parent if the message time is not greater than P(i).subtree_time.

Subtree time forms a monotonically nondecreasing sequence. By propagating subtree time up the processor tree, the root node will know how far the entire simulation has progressed. Thus, subtree time for the root node in the processor tree corresponds to global simulation time which will be discussed shortly.

Subtree time updates are performed by all processors in the processor tree. Eventually, the root node in the processor tree will receive TIME messages from all of its children and update its subtree time. But, in addition to calling it a new subtree time, it will also set a new global time.

When global time is updated, the root processor will send TIME messages with the new global time to all of its child processors. It child processors, upon receiving a global time update message from its parent, will send TIME messages to all of its child processors with the new global time. The propagation of global time updates continues until all leaf nodes in the processor tree have updated their global time.

TIME messages are used to update subtree and global time. TIME messages flowing up the processor tree signal subtree updates while TIME messages flowing down the tree signal global time updates. Thus, there is a continuous flow of TIME messages up and down the processor tree. And since subtree time forms a monotonically nondecreasing sequence, global time forms a monotonically nondecreasing sequence.

4.2 Checkpoint Processing

Each processor has a checkpoint process which is responsible for generating and saving copies of the processor's local state space in a set of checkpoint records.

The checkpoint process is controlled by the following two parameters:

1. Checkpoint interval.
2. Maximum number of checkpoint records that may be saved.

The checkpoint interval is a nonzero value that determines the frequency of checkpoints. Following processor initialization, the first checkpoint record is written at local time zero. The checkpoint processor is then scheduled to generate a new checkpoint record whenever local time has advanced the specified checkpoint interval. Thus, if the checkpoint interval is ckpt_interval, then checkpoint records will be generated at times n*ckpt_interval, where n=0,1,2,...

The second checkpoint process parameter specifies the maximum number of checkpoint records that may be saved. (Note that there must be at least one saved checkpoint record.) If the checkpoint process is requested to generate a checkpoint record and the maximum number of saved checkpoints currently exist, the processor will be unable to advance its local time until at least one checkpoint record is released.

Each checkpoint record contains a time stamp in addition to a copy of the processor's local state space. If the time stamp is t, then the checkpoint record contains a copy of the processor's local state space after all time t processing has completed.

4.3 Rollback

During the course of execution, it may be necessary for a processor to perform a rollback in order to synchronize itself with its neighbor processors. Specifically, a processor will perform a rollback when it receives a DATA message sent by another processor with time stamp not in its local future (message time is less than or equal to the processor's current local time).

Rollback processing is driven by a parameter called *rollback time*. When a tardy DATA message is received, rollback time is defined as the time stamp on the DATA message. Rollback time identifies the precise point in time where a processor must go in order to synchronize itself.

Rollback time partitions a processor's execution history into three disjoint time intervals. Suppose processor P(i) receives a DATA message m and m.time is less than P(i)'s current local time, P(i).local_time. P(i) must perform a rollback in order to process m at the proper time. P(i)'s execution history is partitioned into the following three time intervals:

I1=[0,m.time), I2=[m.time], I3=(m.time,P(i).local_time]

Interval I1 contains all processing performed by P(i) prior to the rollback time. Rollback will not affect any of the processing already performed by P(i) during I1.

I2 contains a single point in time, namely the rollback time. Since message m must be executed during I2, P(i)'s processing will change during its reexecution of I2 because of the additional processing required by message m.

I3 contains all processing performed by P(i) after the rollback time. Since I2 processing will change due to the introduction of message m, so will I3. Therefore, all of the processing already performed by P(i) during I3 is invalid and must be canceled.

Therefore, to perform a rollback to accept a tardy DATA message m, processor P(i) must

1. Restore its local state space to what is was at some time during I1.
2. Add message m to its input buffer so that it will be executed during I2.
3. Delete all remnants of I3 processing.

The following sections describe how P(i) performs each of these tasks.

Interval I1 processing. P(i) has saved copies of its local state space in checkpoint records. Recall that the checkpoint record written at time t contains a copy of P(i)'s local state space after all time t processing was performed. Thus, P(i) must find and restore a checkpoint record with time stamp less than m.time. P(i) is guaranteed to find such a checkpoint record since one was written immediately after initialization at time zero.

Suppose P(i) selects a checkpoint record with time stamp ckpt_time. P(i) will restore the selected checkpoint record which effectively moves P(i)'s local clock back to time ckpt_time. In addition to restoring the checkpoint record, P(i) will delete all saved checkpoint records with larger time stamp since they will be rewritten when P(i) resumes forward execution.

Examine how this checkpoint record restore partitions interval I1 into the following two intervals:

I1'=[0,ckpt_time], I1"=(ckpt_time,m.time)

When P(i) resumes forward execution its local clock will be set to ckpt_time. Since the restored checkpoint record was written after all ckpt_time events were processed, the first event reexecuted by P(i) will have event time greater than ckpt_time. That is, P(i) will resume forward execution in I1".

Since P(i) has already executed during I1", all output messages it previously generated and input messages it previously received and executed will be in its output and input buffers, respectively. The input messages will be processed as they were during the previous execution of I1". And, as P(i) regenerates output messages, it will notice that these messages are already in the output buffer and the regenerated output messages will be discarded.

Interval I2 Processing. P(i)'s reexecution during I2 is identical to its reexecution during I1" with the exception of the new message m. I2 processing will proceed exactly as it did during the previous execution until message m is selected from P(i)'s input buffer for processing. Since m is the last message in the input buffer with time stamp m.time, P(i)'s execution history will diverge from its previous execution with the processing of m.

Interval I3 Processing. In order to cancel the effect of the previous execution of I3, P(i) must examine its and output buffer to determine whether other processors will also have to rollback.

During P(i)'s previous execution during I3, it may have generated output messages to be sent to other processors. Furthermore, the output messages it generated may have already been transmitted. This implies that P(i)'s rollback may affect the local processing performed by other processors.

P(i) begins by examining its output buffer for DATA messages with time stamp in I3. Suppose P(i) locates such a message, call it m*.

CASE 1: m* has not been transmitted. Since m* has not yet been transmitted, P(i) simply deletes the message from its output buffer.

CASE 2: m* has already been transmitted. The processor receiving m* must be notified that the message needs to be canceled.

CLAIM: P(i) sent m* to one of its child processors. DATA messages are sent up the processor tree only if the DATA message time is not greater than the subtree time of the sending processor. Thus, since P(i) just received a message with time stamp m.time, it could not have previously received a message with time stamp m*.time, where m*.time is greater than m.time. Therefore, m* was sent by P(i) to a child processor.

P(i) will create a DATA message to send to the child processor that received m*, say P(j). The new DATA message m' will be initialized as follows:

```
m'.sender   := P(i)
m'.receiver := P(j)
m'.type     := DATA
m'.text     := NULL
m'.time     := m*.time - 1
```

Message m' is being used to force P(j) to cancel the effect of message m* that it received earlier from P(i). In order to accomplish this, the event packaged in the message is NULL and the message time is set to one time unit less that the time stamp on the message to be canceled. When P(j) receives the message, it will cancel all processing performed as a result of its earlier receipt of m* using rollback.

P(i) examines of its output buffer for messages representing child processors that must also perform a rollback. For each such message found, a NULL DATA message is created and all other messages in the output buffer with larger time stamps that were sent to that child processor are deleted. Eventually, P(i) will have deleted all remnants of interval I3 processing.

5. HIERARCHICAL ROLLBACK: THE ALGORITHM

Each processor P(i) in the processor tree executes the local program shown in Figure 1.

Following the initialization of local data areas, the processor enters a loop which continues until the simulation terminates. P(i) begins by examining its input channels for messages sent by other processor. Next, it executes its local simulation until either an output message is generated or the checkpoint process writes a checkpoint record, whichever occurs first. P(i) then sends all messages awaiting transmission that it can from its output buffer. And finally, P(i) updates its local subtree time. Procedures Receive_messages and Send_Messages will described in the following sections.

5.1 Receive_Messages

Procedure Receive_Messages will receive all messages sent to P(i) by other processors. P(i) will only receive messages awaiting reception; that is, if there is not a message to be received, P(i) will not wait for one. The processing performed when a TIME or ACK message is received as been described earlier. DATA message reception is performed by the procedure Receive_DATA_Message shown in Figure 2.

```
PROCEDURE Main
Initialize
WHILE termination criterion not met DO
    Receive_Messages
    Update_Local_State
    Send_Messages
    Update_Subtree_Time
ENDWHILE
```

Figure 1: PROCEDURE Main

```
PROCEDURE Receive_DATA_Message(DATA_msg)
CASE(P(i).state(j))
   #normal#: IF DATA_msg.time <= P(i).local_time
                THEN Local_Rollback(DATA_msg.sender,
                                    DATA_msg.time)
             Accept_DATA_Message(DATA_msg)
   #react#:  IF DATA_msg.time > P(i).last_msg_time(j)
                THEN delete DATA_msg
                ELSE IF DATA_msg.time < P(i).last_msg_time(j)
                        THEN P(i).state(j) := #normal#
                     Local_Rollback(DATA_msg.sender,
                                    DATA_msg.time)
                     Accept_DATA_Message(DATA_msg)
ENDCASE
```

Figure 2: PROCEDURE Receive_DATA_Message

Assume P(i) receives a DATA message from P(j). P(i)'s local processing is dependent on whether P(i) is waiting for P(j) to return an ACK for a previously transmitted DATA message. If P(i) is not awaiting an ACK, P(i).state(j) equal to #normal#, then P(i) compares the DATA message time stamp with its local time. If the message time is in P(i)'s local future, then P(i) calls Accept_DATA_Message to add the event contained in the message to its event list and creates an ACK message to return to P(j). If the DATA message time stamp is in P(i)'s local past or present, P(i) will perform a rollback by calling Local_Rollback. After rollback processing has completed, P(i) will call Accept_DATA_Message to complete reception of the DATA message.

If P(i).state(j) is equal to #react#, then P(i) is waiting for P(j) to return an ACK for a previously transmitted DATA message. This means that P(i) and P(j) have simultaneously exchanged DATA messages. P(i) must compare the time stamp on the DATA message it received from P(j) with the time stamp on last DATA message it sent to P(j). If the time stamp on P(j)'s data message is greater than the time stamp on P(i)'s DATA message, then P(i) knows that P(j) will have to perform a rollback when it receives P(i)'s DATA message. That rollback will require the cancelation of the DATA message just received by P(i), so P(i) may safely delete the P(j)'s DATA message.

If the time stamp on P(j)'s DATA message is less than the time stamp on P(i)'s DATA message, then P(i) will have to perform a rollback since its DATA message was sent without knowing that it would receive an earlier P(j) DATA message. Furthermore, P(i) must simulate the receipt of an ACK from P(j) since P(j) will not be returning one (because of reasoning used earlier).

If the time stamp on the two DATA messages are equal, then P(i) will add P(j)'s DATA message to its input buffer and return an ACK to P(j). But before doing so, P(i) will make sure that its local state is reset so that P(j)'s DATA message will be processed at the proper time.

As seen in Receive_DATA_Message, there are two cases where P(i) calls Local_Rollback when a DATA message is received. This procedure is shown in Figure 3.

Local_Rollback performs the processing necessary for P(i) to reset its local state in order to receive a DATA message. In addition to local processing, it was noted earlier that P(i) may have to request that other processors perform a rollback in response to P(i)'s rollback. This is performed by procedure Secondary_Rollback in Figure 4.

```
PROCEDURE Local_Rollback(rollback_initiator,rollback_time)
IF rollback_initiator = P(i).parent
    THEN delete all DATA messages in P(i).input_buffer from
            P(i).parent with time stamp > rollback_time
IF rollback_time <= P(i).local_time
    THEN Restore_Checkpoint_record(rollback_time)
            Secondary_Rollback(rollback_initiator,rollback_time)
Delete all DATA messages from P(i).output_buffer with message
    time > rollback_time
```

Figure 3: PROCEDURE Local_Rollback

```
PROCEDURE Secondary_Rollback(rollback_initiator,rollback_time)
FOR all k in P(i).child(k)
    IF k <> rollback_initiator
        THEN IF P(i) sent P(i).child(k) a DATA message with time
                stamp greater than rollback_time
                THEN Build NULL DATA message to sent to P(k)
                        Purge P(k)'s messages from input buffer with
                        time stamp > NULL DATA message time stamp
```

Figure 4: PROCEDURE Secondary_Rollback

5.2 Send_Messages

The procedure used by P(i) to send messages to other processors is shown in Figure 5.

P(i) will attempt to send as many messages as possible when Send_Messages is called following the rules for processor communication. Suppose P(i) selects a DATA message to send to P(j). P(i) will only send the DATA message to P(j) if it does not currently have an outstanding unacknowledged DATA message already sent to P(j). If the DATA message may be sent, then P(i) will send the message, set P(i).state(j) to indicate that it is awaiting an ACK from P(j), and save the DATA message time stamp. If the time stamp on the transmitted data message is m.time, then P(i) must check to see whether it has already processed a P(j) DATA message with time stamp greater than m.time. If so, then P(i) must undo the processing performed as a result of that DATA message. This instance of a rollback is caused by the delayed transmission of a DATA message by a processor.

6. DISCUSSION

Hierarchical rollback is a new technique for distributed simulation which contains feature common to both the conservative and optimistic approaches. The algorithms used are fully distributed so that there is no notion of a central simulation controller. Hierarchical rollback attempts to take advantage of the strengths of the conservative and optimistic approaches while minimizing their deficiencies.

The conservative approach has the advantage of never wasting processing time by having to perform a rollback. However, processor throttling may impede progress and deadlocks may occur. The optimistic approach encourages concurrent submodel execution by not using a throttling mechanism to control the progress made by each submodel. But, when a synchronization problem does arise, rollback may waste processor time, and it is possible for a rollback to cascade through the entire network of processors.

Hierarchical rollback uses subtree time as its throttling mechanism. This throttle only limits the transmission of simulation events to higher level processors in the processor tree, it does not limit the progress made within a processor. Subtree time also limits the propagation of rollbacks to a subtree. It is impossible for the entire processor tree to be affected by a rollback initiated by a single processor.

The use of a hierarchical simulation model for hierarchical rollback encourages the top–down design of simulation models. It also provides the analyst the opportunity to lessen the probability and severity of a rollback by the judicious partitioning of the model tree into subtrees for processor assignment. The tree structure of processors also eliminates the possibility of deadlock.

The parameterization of the checkpoint process is crucial to hierarchical rollback. As the size of the checkpoint interval increases, so does the potential amount of time lost due to rollback. A large checkpoint interval will cause a processor to take large steps back in time whenever a rollback occurs. On the other hand, a small checkpoint interval will lessen the amount of processing

```
PROCEDURE Send_Messages
m := first message in P(i).output_buffer awaiting transmission
done := (m = NULL)

WHILE NOT done DO
    CASE(m.type)
        ACK:  send message m
              delete message m
        TIME: send message m
              delete message m
        DATA: CASE(P(i).state(m.receiver))
                  #normal#: send message m
                            P(i).state(m.receiver) := #react#
                            P(i).last_msg_time(m.receiver) := m.time
                            IF P(i) processed a m.receiver DATA
                               message m* with m*.time > m.time
                               THEN Local_Rollback(m*.sender,
                                                   m*.time)
                            Buffer_Purge(P(i).input_buffer,m.time)
                  #react#:  done := TRUE
              ENDCASE
    ENDCASE
    m := first message in P(i).output_buffer awaiting transmission
    done := done OR (m = NULL)
ENDWHILE
```

Figure 5: PROCEDURE Send_Messages

time lost due to rollback, but more storage is required for checkpoint storage.

The maximum number of checkpoint records that may be saved is also a controlling factor. Allowing a large number of saved checkpoint records enables a processor to progress quite far in its local simulation (similar to a large checkpoint interval). This may also result in large steps back in time when a rollback occurs. If a small number of checkpoint records are saved, this will serve as another throttling mechanism for submodel progress. It is possible that the proper combination of checkpoint parameters would cause hierarchical rollback to approximate the performance of the conservative approaches.

Experiments were performed to investigate the performance of hierarchical rollback under varying submodel interaction intensities, processor populations, and checkpoint parameters. The results of these experiments will be the subject of a forthcoming paper.

ACKNOWLEDGEMENTS

The author would like to thank his advisor, Professor K. Mani Chandy, for the advice and comments he gave during the course of this research. The research reported in the paper was supported by a grant from the IBM Corporation.

REFERENCES

Chandy, K.M. and Misra, J. (1979), Distributed Simulation: A Case Study in Design and Verification of Distributed Programs, *IEEE Transactions on Software Engineering*, Vol. SE–5, No. 5, 440–452.

Chandy, K.M. and Misra, J. (1981), Asynchronous Distributed Simulation via a Sequence of Parallel Computations, *Communications of the ACM*, Vol. 24, No. 4, 198–205.

Gimarc, R.L. and Chandy, K.M. (1985), HSS: A Tool for Evaluating the Performance of Office Systems, *Proceedings of the Computer Measurement Group International Conference.*

Jefferson, D. (1985), Virtual Time, *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, 404–425.

Misra, J. (1986), Distributed Discrete–Event Simulation, *ACM Computing Surveys*, Vol. 18, No. 1, 39–65.

Peacock, J.K., Wong, J.W., and Manning, E. (1980), Distributed Simulation Using a Network of Processors, *Computer Networks*, Vol. 3, No. 1, 44–56.

Wirth, N. (1971), Program Development by Stepwise Refinement, *Communications of the ACM*, Vol. 14, No. 4.

AUTHOR'S BIOGRAPHY

RICHARD L. GIMARC is a Ph.D. student in the Department of Computer Sciences at the University of Texas at Austin. He received his B.A and M.S. degrees in Mathematics from Texas Tech University in 1974 and 1979, respectively. His current research interests include systems modeling and performance evaluation. Mr. Gimarc is currently employed as a Product Development Manager for Boole & Babbage, Inc.

Richard L. Gimarc
7605 Eastcrest Drive
Austin, Texas 78752
(512) 454–6043