Proceedings of the 1997 Winter Simulation Conference
ed. S. Andradóttir, K. J. Healy, D. H. Withers, and B. L. Nelson

CLONING: A NOVEL METHOD FOR INTERACTIVE PARALLEL SIMULATION

Maria Hybinette
Richard Fujimoto

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

ABSTRACT

A new scheme for interactively testing what-if and alternative scenarios in parallel simulations is presented. Potential branches or choices can be specified interactively and interjected into the on-going simulation as *decision points.* Once a decision point is defined, different "futures" are computed by cloning the simulation, and executing the clones concurrently. A construct called *virtual logical processes* is introduced to avoid repeating common computations in different cloned simulations.

1 INTRODUCTION

Interactive techniques can be used to effectively steer, analyze and monitor parallel simulation applications. Capabilities that may be provided by an interactive system includes pausing, steering, monitoring, rollback and reverse execution. Pausing the simulation is important when a participant wishes to interject immediate or future branches into the simulation as *decision points.* Steering is the insertion of decision points without pausing. Steering and pausing can also provide for modification of simulation parameters and variables. Monitoring is a process whereby specified variables in the simulation are sampled. The variables may be sampled continuously or triggered when a predefined condition occurs, such as the expiration of an execution path. Rollback provides for the re-evaluation of old decision points and variables. Reverse execution enables the system to re-construct the events leading to an outcome. In this paper we introduce a new interactive mechanism for the exploration of what-if and alternative scenarios.

As an example of how interactive simulation can be used, consider air traffic control. Here, the controller could benefit from an interactive simulation to determine which scheduling policy offers minimal delay. If an unexpected event occurs within the simulation, such as a runway closure, the controller interacting with the simulation may want to evaluate the effect of activating different scheduling policies. This can be done by dynamically cloning (replicating) the simulation with different scheduling policies. The effect of each policy within each clone is monitored to determine which policy offers the most effective solution. Inefficiencies in the air traffic flow can be discovered by monitoring the simulation then allowing backward execution to evaluate cause and effect relationships.

Existing simulators must re-execute a simulation from a check-pointed state or from the beginning to explore alternatives in what-if-scenarios. These techniques re-evaluate scenarios in a batch mode, one alternative after the other. We propose to dynamically create and evaluate different alternatives ahead of time and in parallel with the on-going simulation by using a mechanism that provides for interactive *cloning.*

The mechanism advocated here clones a new version of a simulation in progress to evaluate an alternative. Cloning is initiated on the arrival of a request to explore a possible execution path. The alternate version of the simulation proceeds in parallel with the original simulation. At least one of the versions will eventually commit and the other will expire (unless both results are requested.) The decision of which alternative to expire may be interactive or automatic depending on a given condition included in the original request. A benefit of exploring alternatives in parallel is that an increasing number of alternatives can be inspected before resolution.

Our approach is intended for parallel discrete event simulators, where the simulation consists of a collection of logical processes (LPs) potentially executing on different processors (Fujimoto 1990). This type of simulator is driven by LPs exchanging time-stamped event messages. The simulation maintains consistency by enforcing all events to be processed in time-stamp order. Two consistency protocols prevail:

conservative protocols and optimistic protocols. The conservative protocol enforces consistency by avoiding the possibility of ever receiving an event in the past. The optimistic protocol, in contrast, provides for receiving an event from the past by "rolling back" previously processed events with earlier time-stamps than the one received.

The cloning mechanism is applicable to both conservative and optimistic simulation protocols. Parallel discrete event simulators provide speed up for time-consuming applications such as strategic battle-management, telecommunication networks, air-traffic scheduling, various engineering applications and other large scale simulations. Strategic and tactical battlefield simulation in particular would benefit from our mechanism, since they often weigh options and must be responsive to unanticipated scenarios.

The paper is organized as follows: Background work is discussed in Section 2. This is followed by a discussion of the virtual logical process paradigm. Section 4 presents the cloning mechanism. Next, Section 5 discusses implementation concerns and implications. We conclude with a summary and a discussion of future directions.

2 BACKGROUND

Interactive parallel discrete event simulation is a relatively new field. It is becoming active and the body of literature is growing. Initial methods for interactive parallel discrete event simulation using an optimistic simulator have been discussed in (Steinman 1991) and (Franks et al. 1997).

In (Steinman 1991) a special event type called an *external* event is used to enable interaction with the simulator. These events can steer the simulation in some desired direction and sample or query the progress of the simulator. The approach of (Franks et al. 1997) allows a user to test what-if scenarios provided they are interjected before a deadline. Alternatives are examined one after the other and the simulation must undo the effect of the previous alternative before considering another.

The cloning approach introduced here improves over these related methods by allowing multiple evaluations to be carried out simultaneously. Cloning generalizes to both conservative and optimistic simulation models. Both approaches can adopt the mechanism on top of existing systems.

Cloning or replication is well known in the distributed computing community. Traditionally it is used to ensure fault-tolerance by providing identical processes, identical transactions, duplicate data, or redundant services (Schneider 1990). Cloning can also improve throughput by placing process replicas in the proximity of where a service is needed (Goldberg 1992; Schneider 1990).

In real-time databases (Bestavros 1994) suggests in his concurrency control algorithm, to fork reader processes of dirty data to execute the same transaction with different ordering assumptions. This avoids the hazards of blockages or restarts. An alternative schedule is adopted when a suspected inconsistency materializes; otherwise it is abandoned.

Replication has been used in the simulation community to improve the accuracy of simulation results or to find optimal parameter settings (Glasserman, Heidelberger, and Shahabuddin 1996; Vakili 1992; Glynn and Heidelberger 1991). The approach is to run multiple independent replications then average their results at the end of the runs.

In (Goldberg 1992), cloning is used to speed up the execution of distributed programs by representing read-mostly bottleneck processes as multiple replicas or clones. Virtual time synchronization keeps replicated processes consistent. A criteria in his simulation is that each process sees the same state, while the state of the cloned process in our paradigm may differ. This work differ in that we use cloning to explore different paths concurrently.

The research we report in this article is motivated by two goals: First, the desire to create a computational model for efficient interactive simulation. Second, the development of a mechanism to realize the model. In particular we are interested in a model that supports an efficient, simple, and effective way to explore alternate scenarios. This is decidedly different than the studies reported above.

3 THE VIRTUAL LOGICAL PROCESS

An essential requirement of interactive parallel discrete event simulation is to efficiently explore multiple possible intermediate outcomes. To provide speed the model should parallelize the process of determining committed states in order to quickly furnish alternatives. We propose cloning the simulation to facilitate the exploration of multiple scenarios.

A brute force approach to cloning is to replicate the entire state of the simulation and create an independent execution for each clone. This is very wasteful, however, as there may be many identical computations repeated in the different clones. A mechanism is needed to avoid replication of state and computation among the clones except when they are different.

In order to provide efficiency in cloning a simulation we introduce the notion of *virtual logical processes* (Vs). In this paradigm the simulation is viewed as a collection of virtual logical processes. Each time a simulation is cloned a new version of the Vs is created. For example, at the inception of a parallel simulation there is one version of Vs. When the simulation is cloned, a second version of the simulation is instantly created by instantiating a second collection of Vs.

Consider the example scenario in Figure 1, depicting a simulation with three LPs: A, B and C. The snapshot shows that the simulation has been cloned once, so two versions of the virtual logical processes exist. Each version is shown as a plane. The first version, the top plane, has a set consisting of three virtual logical processes. Similarly, the plane on the bottom, representing the cloned simulation, consists of a set of three virtual logical processes.

A virtual logical process has its own state and effects communication via messages between processes of the same version. The versions diverge as the state between versions diverges. To avoid copying the *entire* state upon instantiating a clone, the state is updated incrementally. This is done by assigning the state of each virtual logical process (V) to a physical logical process (P). Each of the virtual logical processes are said to *map* to the physical logical process that maintains its state. One physical process may maintain the state of several virtual logical processes.

Process mapping is analogous to virtual memory where a virtual address is translated to a physical memory address. The same address in main memory can be shared by two virtual addresses, but two addresses in main memory cannot be mapped to the same virtual address. Similarly, two virtual processes may map to a single physical process, but two physical processes cannot be mapped to the same virtual process.

To facilitate the discussion we introduce some notation and definitions. *Version* refers to an entire simulation. For example, version 1 refers to the original virtual simulation, virtual version 2 refers to the first cloned simulation of version 1. Virtual logical process A version 1 is denoted by V_A^1, where version number is the superscript and specific logical process is the subscript. V denotes a virtual logical process. P denotes a physical logical process. Similarly, P_A^2 refers to version 2 of physical logical process A. The version number of a physical logical process is the same as the *lowest* version number of the virtual logical processes that maps to it.

Before a simulation is cloned there is one set of Vs and one set of Ps. The mapping between them is one-to-one and onto. In the example, the mapping between Vs and Ps *before* cloning the simulation is simply:

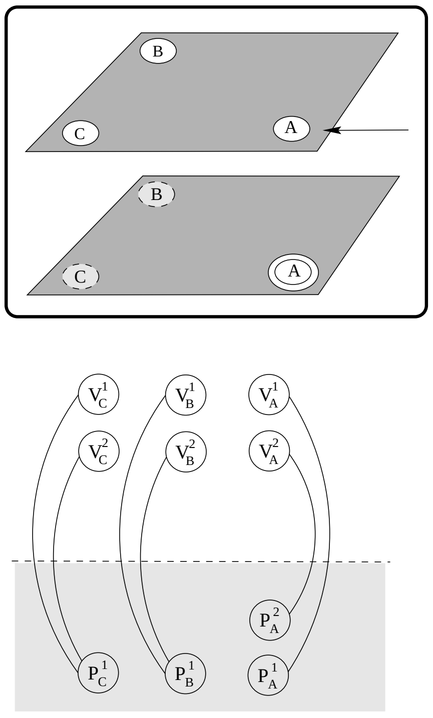


Figure 1: A Snapshot of a Simulation That Has Been Cloned; the Top Image Shows the Two Virtual Versions of the Simulation, the Bottom Image Shows the Mapping of the Virtual Processes to Physical Processes A, B and C

– V_A^1 maps to P_A^1
– V_B^1 maps to P_B^1
– V_C^1 maps to P_C^1

After cloning, there are two sets of virtual processes, each set represents a particular version of the simulation. The first set or version of virtual processes represents the original simulation, the second set represents a new version of the simulation that differs in its mapping to physical logical processes.

The simulation may be cloned *on a set of particular physical logical processes.* In the case where the cloning set contains one physical logical process the effect is the creation of a new set of virtual logical processes and one new physical logical process. The new virtual logical processes map to the same physical processes as the original version except for the virtual logical process that now corresponds to the newly created physical logical process. The new clone then proceeds to process the message that was received as the impetus for cloning. Consequently, the state between the two versions of a physical logical process now differ from the original simulation.

In the example, the bottom image in Figure 1 shows the mapping between virtual processes and physical logical processes after the simulation is cloned on physical process A. Here the mapping of the original version of virtual processes stays the same. The mapping of the new version of virtual processes is as follows:

- V_A^2 maps to P_A^2
- V_B^2 maps to P_B^1
- V_C^2 maps to P_C^1

Notice that for virtual processes B and C, version one and version two share the same corresponding physical process; while virtual process A version one and version two maps to different physical processes.

The semantics of cloning a simulation *on* a physical logical process or processes is what enables the exploration of different scenarios. This represents a decision point where the states of the two versions start to diverge. As the simulation progresses the mapping of virtual processes to physical processes changes, as new physical processes are created. Message sends and receives are carried out in the physical layer. In this manner a physical send corresponds to a *set* of sends in the virtual process layer. In the example (see Figure 2), a send of physical process B to physical process C is echoed in the set of sends consisting of both virtual process B version 1 and version 2 to virtual process C versions 1 and 2.

4- THE CLONING MECHANISM

A cloning mechanism must consider the following issues: (1) when a physical process is created; (2) how and when the mapping between virtual logical processes and physical logical processes changes; and (3) what set of virtual logical processes receives and sends messages.

To illustrate how and when a physical logical process is created and which physical processes send and receive messages, we describe different scenarios encountered in an example military tactical simulation consisting of a platoon, a tactical wing and a theater command center. To conform with our earlier example depicted in Figure 1, the platoon is represented by logical process A, the tactical wing by logical process C, and the theater command center by logical process B. The simulation is cloned to simulate two outcomes: A missile hit and miss on the platoon. Here the state of the two versions of the simulation differ in the state of the platoon and therefore a new physical process is created to map to the second virtual version of the platoon. Each set of virtual logical processes representing the tactical wing and the theater command center maps to one physical process, P_C and P_B, respectively.

In order to determine when a physical process is created or what set of physical logical processes receive a message, the mapping before the sending of a message is considered. The approach considers the sets of virtual processes mapped to the physical send process and the physical receive process. This determines if a physical process or a message needs to be cloned. To reduce the cost of *multiple-futures*, the mechanism re-uses as many resources as possible and only replicates smaller portions that cannot be shared.

To highlight the different possible scenarios we will continue to step through the military example. Recall that the simulation is cloned on P_A^1, thus resulting in the creation of P_A^2. The cloning of P_A^2 results in V_A^2 mapping to a different process than V_A^1, this is denoted in the figure by a second circle around V_A^2 in version 2 of the planes. The damaged platoon (V_A^1) differs from the undamaged platoon (V_A^2) and thus is mapped to a different physical process.

Following the initial cloning, suppose the theater command center (B) requests a status report from the tactical wing (C). Message sends and message receives are realized on the physical layer so this is the same as physical process P_C^1 receiving a message from physical process P_B^1 as shown in Figure 2. The receiving of a message is determined by examining the sets of virtual processes mapped to the sending and receiving process. In this case only physical process P_C^1 receives a message because each member in the virtual send set (V_B^1, V_B^2) is received by virtual processes (V_C^1, V_C^2) which are mapped to the same physical process P_C^1. Thus the same physical message between the physical processes is echoed between the virtual send set and virtual receive set. This is because both virtual versions are identical in state and should be equally influenced by the same message. Logical processes apart from the consequences of the platoon remain the same and can share the same computations.

Continuing with the example, suppose the theater command center (B) requests a status report from the platoon (A) of the consequence of the missile – that is, P_B^1 sends a message to P_A^1 (see Figure 3). A message send from P_B^1 implies that each member (V_B^1 and V_B^2) in the virtual processes set that maps to P_B^1 send a message. Because virtual processes (V_A^1 and V_A^2) are mapped to different physical processes and both see the arrival of a message from their corresponding virtual sender (V_B^1 and V_B^2) two physical messages are sent, one message to each physical receiver (P_A^1 and P_A^2).

Next, the damaged platoon (V_A^1) proceeds by requesting extra man power from the theater command

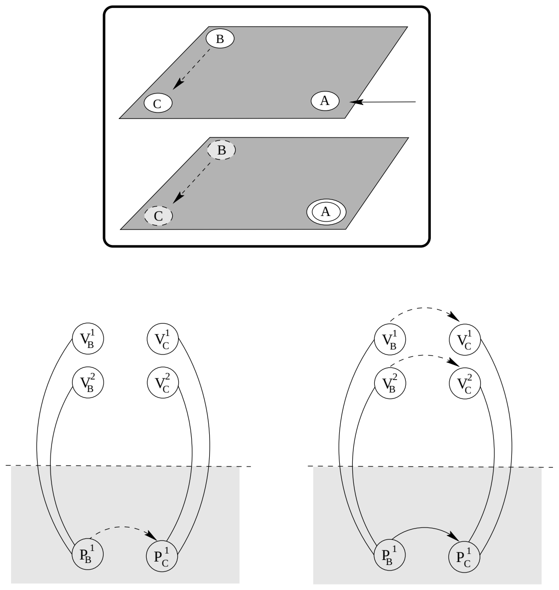


Figure 2: Transparent Send and Receive between Virtual Logical Processes

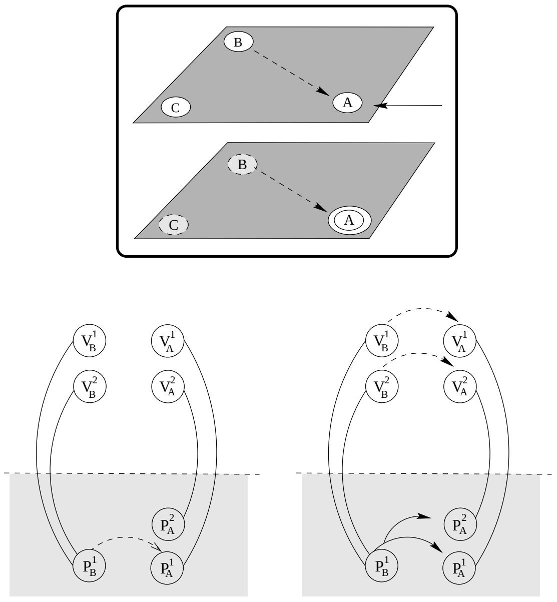


Figure 3: Multi-Casting a Message

(V_B^1). This corresponds to the sending of a physical message from P_A^1 to P_B^1 (see Figure 4). Since virtual process version 2 of processes B should not receive the message, version 2 of the virtual receivers should be prevented from being influenced by this message. This is achieved by changing the mapping between virtual and physical processes. The result is that the process is cloned *before* an event is processed so that the state of the physical process is not influenced by the incoming message. The new "version 2" of physical process of B is prevented from the misconception that A needs extra manpower.

Finally, the theater command (V_B^1) sends extra manpower to the damaged platoon (V_A^1). In the physical layer, P_A^1 again sends a message to P_B^1. This corresponds to virtual process V_B^1 sending a message to virtual process V_A^1. Here, no sending or receiving is echoed in other virtual versions because there is no sharing of physical processes and the message should only influence the simulated version that corresponds to the message send.

There are four general cases. All possible scenarios can be derived from the base cases. They are based on the mapping of the sender and of the receiver before receiving the message:

- The same physical sender and receiver is shared between virtual logical processes (see Figure 2, virtual processes V_B^1 and V_B^2 both maps to P_B^1 and both V_C^1 and V_C^2 maps to P_C^1).
- The same physical sender is shared between virtual senders but the virtual receivers do not share the same physical receiver (see Figure 3, V_B^1 and V_B^2 both maps to the sender P_B^1, but V_A^1 and V_A^2 maps to different physical processes).
- The physical sender is not shared between virtual senders but the virtual receivers share the same physical receiver (see Figure 4, virtual receiver V_B^2 that is initially mapped to physical receiver P_B^2 does see a send from virtual sender V_B^2).
- Virtual logical processes are independent of each other and do not share physical logical processes (see Figure 5, P_A^2 does not receive a message from P_B^2).

The cloning mechanism first determines the physical processes that receive a message. Next it considers the generation or cloning of a new physical process. As illustrated in the example above the reception of messages and creation of processes depends on the set of virtual processes mapped to the sending physical process and on the set of virtual processes mapped to the receiving physical process. The following sets and functions are defined:

`PRcvSet` (*msg*) The set of versions of physical logical processes that receive message *msg*.

`PSend` (*msg*) The single version of a physical logical processes that sends message *msg*.

`VRcvSet` (*msg*) The set of versions of virtual logical processes mapped to a single version of a physical logical process receiving the message *msg*.

`VSendSet` (*msg*) The set of versions of virtual logical processes mapped to a single version of a physical logical process sending the message *msg*.

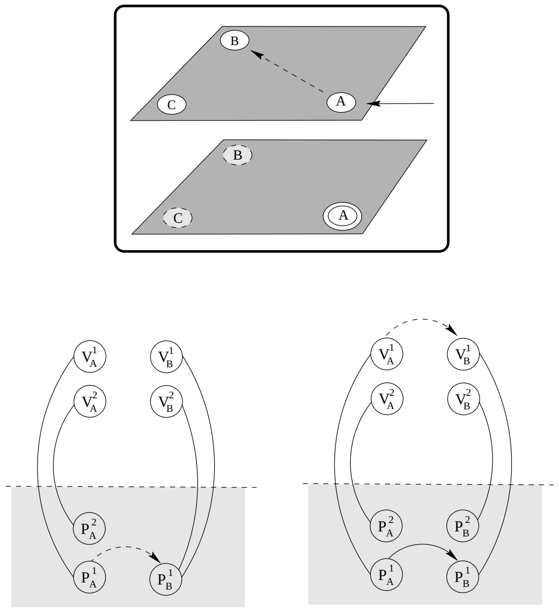


Figure 4: Creation of a Physical Logical Process

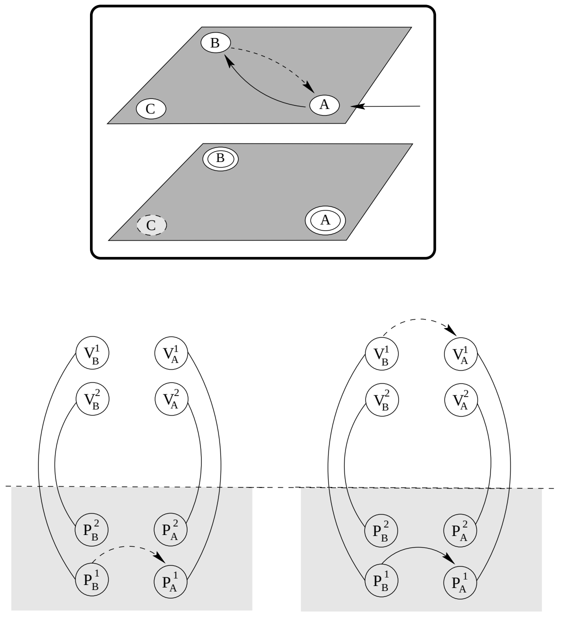


Figure 5: Un-Transparent Send and Receive between Virtual Logical Processes

`VSet` (P_x^i) The set of versions of virtual logical processes mapped to the single version of physical logical process P_x^i.

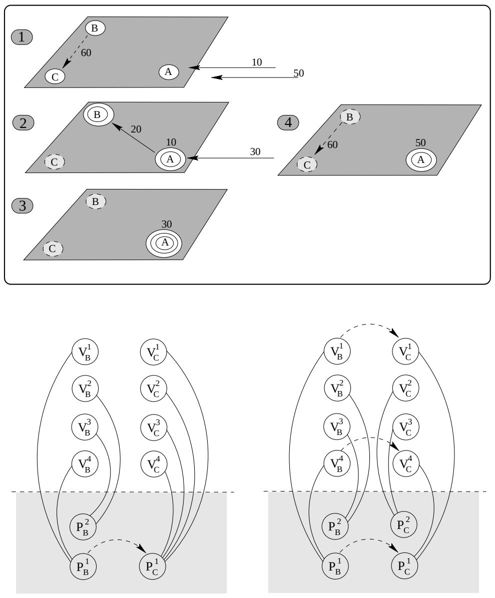
`PSet` $(V, \ x)$ The set versions of physical logical processes that maps to the set V of virtual logical processes x. Returns the version numbers of all physical processes at x that map to some virtual version $\in V$.

`VLow` (V) The lowest version number of the virtual processes in the set V.

The physical processes to receive a message are determined as it is sent. Each physical process that maps to a version that is also in the virtual send set (`VSendSet`) receives a message. In Figure 3 where physical logical process P_B^1 sends a message to physical logical process P_A^1 the `VSendSet` is $\{ \ 1, \ 2 \ \}$. A message sent from P_B^1 implies that both V_B^1 and V_B^2 sent a message, because both versions of the virtual process B are mapped to the same physical logical process. `PRcvSet` is thus `PSet` $(\ \{ \ 1, \ 2 \ \}, \ A \ ) = \{ \ 1, \ 2 \ \}$. Since `PRcvSet` has an additional physical process the message is copied and forwarded.

After determining which processes receive a message, the generation of new physical processes is considered. A physical process is created if there is a virtual logical process in the receive set that does not have a corresponding virtual sender or if the receiving physical logical process found above has not yet been created. For instance, consider a logical process that has been cloned (say the clones are version 1 and version 2). Next, version 1 of the physical processes proceeds to send a message to a physical process that has not yet been cloned. Now since version 2 of the virtual processes did not send the message, version 2 of the virtual receivers should be prevented from receiving a message from physical logical process version 1. The first version should not influence the second virtual version. This is achieved by creating a new copy of the physical logical process before receiving a message.

Determining the set of physical logical processes that need to be cloned (`PClone`) includes inspecting the `VRcvSet` of each receiving physical process. `PClone` is determined by adding the lowest version number of each set that remains from subtracting `VSendSet` from all `VRcvSets`. Recall that the version number of a physical logical process is the same as the *lowest* version number of the virtual logical process that is mapped to it. In order to illustrate this case refer to Figure 4. Here, physical logical process P_A^1 sends a message to physical logical process P_B^1. `PRcvSet` is $\{1\}$. Virtual versions 1 and 2 map to the physical process P_B^1, so `VRcvSet` is $\{1, \ 2\}$. The set of versions of virtual logical processes that maps to the physical sender P_A^1 is $\{1\}$. Thus, we need to create a logical process to map to virtual process $\{2\}$ before processing the message because `VRcvSet - VSendSet` $= \{2\}$ (it is only one version in the difference so this is also the lowest version number). A case where the lowest version number is important is shown in Figure 6.



1
2
3
4
A
B
C
60
10
50
20
30
V_B^1 V_B^2 V_B^3 V_B^4
V_C^1 V_C^2 V_C^3 V_C^4
P_B^1 P_B^2 P_C^1 P_C^2

Figure 6: Four Cloned Simulations

The remaining two cases do not require messages to be copied nor the creation of physical processes (see Figure 2 and Figure 5). Thus, to implement the model we furnish functions that decide when to create a physical logical process and when a message is copied and forwarded to additional physical logical processes.

5- IMPLEMENTATION

As discussed in the previous sections, an algorithm to implement the cloning mechanism needs two steps: The first is to determine the set of physical receivers (`PRcvSet`); the second is to determine which of the physical logical receivers are cloned. The pseudo-code below shows an algorithm for the first step:

```
Input: PRcvSet, VSendSet, PSend
Output: PRcvSet

function GetPRcvSet
PRcvSet = PSend
WorkSet=VSendSet-VSet ( $P_{rcv\_lp}^{\texttt{VLow(PRcvSet)}}$ )
while ( WorkSet $\neq$ $\emptyset$ )
    $x$ $\in$ VLow (WorkSet)
    PRcvSend = PRcvSend $\cup$ $x$
    WorkSet = WorkSet - VSet ( $P_{rcv\_lp}^{x}$ )
end_while
return PRcvSet
```

The function *GetPRcvSet* determines the virtual receive set from the virtual send set. The receiving physical processes is then determined by inspecting the mapping between virtual logical processes and physical logical processes of the virtual receive set found in the previous step.

After determining the physical logical processes to receive a message, the processes that need to be cloned are found. Pseudo-code for an algorithm to accomplish this step is listed below:

```
Input: PRcvSet, VSendSet
Output: PCloneSet

function GetPCloneSet
    if ( | PRcvSet | = 1 and VSet(PRcvSet) = $\emptyset$ )
        $PCloneSet$ = PRcvSet
        return $PCloneSet$
    end_fi
    $PCloneSet = \emptyset$
    for each member $x \in$ PRcvSet
        VRcvSet = VSet ( $P_{rcv\_lp}^{x}$ )
        $PCloneSet = PCloneSet$
            $\cup$ VLow ( VRcvSet $-$ VSendSet )
    end_for
    return $PCloneSet$
end_function
```

The function *GetPCloneSet* retrieves the virtual receive set from each physical processor that will receive a message. The virtual send set is then subtracted from the virtual receive set since these are virtual logical processes that should not be influenced by a message receive from a different simulation version. Each set difference then clones a physical process that is equal to the lowest version number of its processes (the other processes in this set are mapped to the new process).

An architecture utilizing the cloning model is now described. A diagram of the architecture is shown in Figure 7. There are five modules: An interactive module; a user application (describes the system being simulated), the cloning application layer, the simulation kernel, and the cloning kernel layer.

The interactive module is an interface between the user and the simulation. The interface includes two actions, the first is the request to clone a simulation, the second is to expire a particular clone. The request to clone a simulation includes parameters of the new simulation.

In response to a request, a cloning manager in the cloning application layer clones the entire simulation into two virtual versions that run concurrently. The simulation kernel interface copies and creates objects iteratively. The cloning manager maintains the mapping of virtual versions to physical versions. The simulation kernel which drives the application contains

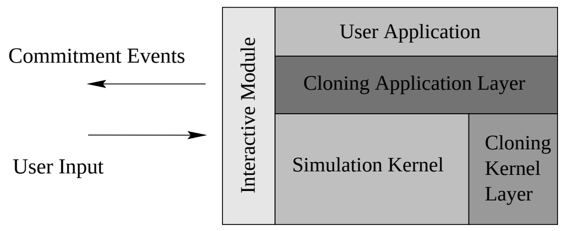


Figure 7: Architecture

either an optimistic or conservative protocol.

6- CONCLUSION AND FUTURE WORK

This paper introduces the paradigm of virtual logical processes to explore different possible futures as an efficient computational model of interactive parallel simulation. The paradigm suggests a cloning scheme, which is described. To avoid copying the *entire* state upon instantiating a clone, the state is cloned incrementally. This is done by assigning the state of a virtual logical process to a physical logical process.

The model is applicable to conservative and optimistic simulation protocols. Our goal is to demonstrate the effectiveness of the model by implementing it on top of the Georgia Tech Time Warp simulator (GTW). A method for re-merging cloned physical LPs if they later converge is also being explored.

Currently, it is assumed that a user interjects decision points into the simulator. Another area of future research is defining a mechanism for automating the interjection of decision points, perhaps, by balancing the "running ahead" and "branching" depending on the probability of particular branches against available computational resources and real time constraints. This may offer an effective tool to explore the ordering of simultaneous events (Wieland 1997) and in the analysis by stochastic methods.

ACKNOWLEDGMENT

This work was supported by U.S. Army Contract DASG60-95-C-0103 funded by the Ballistic Missile Defense Organization.

REFERENCES

Bestavros, A. 1994. Multi-version speculative concurrency control with delayed commit. In *Proceedings of the 1994 International Conference on Computers and their Applications*, Long Beach, California.

Fujimoto, R. M. 1990. Parallel discrete event simulation. *Communications of the ACM* 33 (10): 30-53.

Franks, S., F. Gomes, B. Unger, and J. Cleary. 1997. State saving for interactive optimistic simulation. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, 72-79. Burg Lockenhaus, Austria.

Goldberg, A. P. 1992. Virtual time synchronization of replicated processes. In *Proceedings of the 6th Workshop on Parallel and Distributed Simulation*, 107-16. Newport Beach, California,

Glasserman, P., P. Heidelberger, and P. Shahabuddin. 1996. Splitting for rare event simulation: Analysis of simple cases. In *Proceedings of the 1996 Winter Simulation Conference*, 302-308. Coronado, California.

Glynn, P. W., and P. Heidelberger. 1991. Analysis of parallel replicated simulations under a completion time constrain. *ACM Transactions on Modeling and Computer Simulations* 1 (1): 3-23.

Vaikili, P. 1992. Massively parallel and distributed simulation of a class of discrete event systems: A different perspective. *ACM Transactions on Modeling and Computer Simulations* 2 (3): 214-238.

Schneider, F. 1990. Implementing fault-tolerance services using the state machine approach: A tutorial. *ACM Computer Surveys* 22 (4): 299-320.

Steinman, J. 1991. SPEEDES: Synchronous parallel environment for emulation and discrete event simulation. In *Proceedings of the SCS Western Simulation Multi-Conference on Advances in Parallel and Distributed Simulation*, 95-103. Anaheim, California.

Wieland, F. 1997. The threshold of event simultaneity. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, 56-59. Burg Lockenhaus, Austria.

AUTHOR BIOGRAPHIES

MARIA HYBINETTE is a Ph.D. student in the College of Computing at the Georgia Institute of Technology. Her research interests include parallel simulation, parallel algorithms, and real time systems.

RICHARD FUJIMOTO is a Professor in the College of Computing at the Georgia Institute of Technology. He is working on performance issues related executing discrete-event simulation programs on multiprocessor and distributed computing platforms.