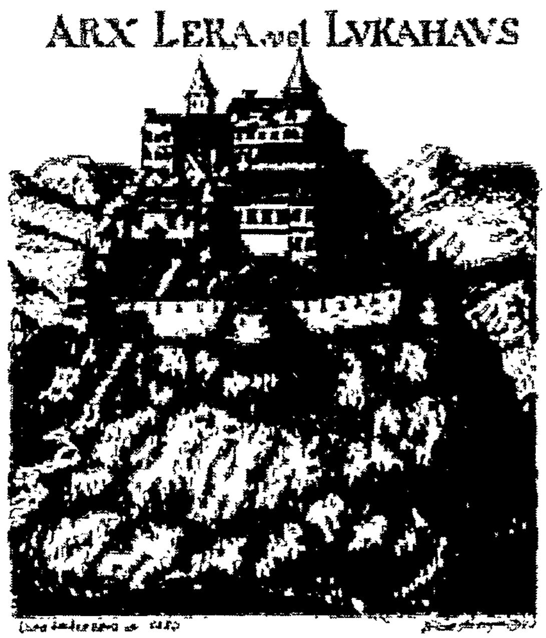

Session 10

Risk

Session Chair

Alois Ferscha
Universität Wien, Austria

The Dark Side of Risk
(What your mother never told you about Time Warp)

D. M. Nicol X. Liu

Department of Computer Science
Dartmouth College
Hanover, NH 03755

Abstract

This paper is a reminder of the danger of allowing "risk" when synchronizing a parallel discrete-event simulation: a simulation code that runs correctly on a serial machine may, when run in parallel, fail catastrophically. This can happen when Time Warp presents an "inconsistent" message to an LP, a message that makes absolutely no sense given the LP's state. Failure may result if the simulation modeler did not anticipate the possibility of this inconsistency. While the problem is not new, there has been little discussion of how to deal with it; furthermore the problem may not be evident to new users or potential users of parallel simulation. This paper shows how the problem may occur, and the damage it may cause. We show how one may eliminate inconsistencies due to lagging rollbacks and stale state, but then show that so long as risk is allowed it is still possible for an LP to be placed in a state that is inconsistent with model semantics, again making it vulnerable to failure. We finally show how simulation code can be tested to ensure safe execution under a risk-free protocol. Whether risky or risk-free, we conclude that under current practice the development of correct and safe *parallel simulation code is not transparent to the modeler; certain protections must be included in model code or model testing that are not rigorously necessary if the simulation were executed only serially.*

1 Introduction

Reynolds has argued that synchronization protocols for parallel discrete-event simulation are characterized by a spectrum of attributes [6]. In particular, he noted that protocols broadly categorized as "optimistic" really entail two different aspects; *aggressiveness* means executing an event before it is certain to be correct to do so, and *risk* means sending a message that might not be correct. That the two ideas could be separated was demonstrated by Reynolds with a variant on his SRADS protocol [3] the distinction was also used by Steinman in development of the SPEEDES system using the Breathing Time Buckets protocol [7]. Both methods employ aggressiveness, but not risk.

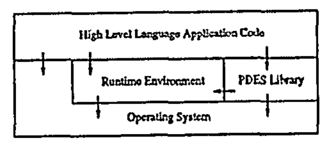

Figure 1: Hierarchy of software layers in typical parallel simulation discrete-event simulation (PDES) package

The most widely cited optimistic systems use risk, notably the Time Warp Operating System (TWOS) [4] and Georgia Time Warp (GTW) [2]. The TWOS effort ended some years ago; GTW typifies Time Warp simulators in current use. GTW is essentially a library whose classes and methods can be used in a C or C++ program to transform it into a simulation. One can think of a GTW simulation as code in some high level programming language that calls the GTW libraries, which use Unix facilities. Figure 1 illustrates this layering in the general case. Regions where some state-saving occurs are highlighted.

Optimistic methods are able to recover from temporal errors by saving enough "state" to return them to a prior simulation time. The modeler is responsible for identifying state variables in user code; the Time Warp state is comprised of these variables and some internal state information. Much of the operating environment in which an optimistic simulation executes is not considered to be state, and is not saved or restored. The significance of this fact cannot be overstated. The trend in parallel simulators is to link together simulation libraries, user model code, and user code libraries. The user code libraries in particular may not have been developed for use with optimistic parallel

1084-4097/97 $10.00 © 1997 IEEE

simulation in mind. A case in point is the TeD simulation language being developed at Georgia Tech for the simulation of telecommunication networks; TeD resides astride GTW [1]. Very explicit mechanisms are provided in that tool to link to potentially large blocks of ordinary C/C++ code; the language by design is a "meta-language" that allows certain simulation constructs to be embedded in a high level language. The problem we consider is that under normal Time Warp operating rules, user model code and user libraries may be executed using data arguments that are utterly inconsistent with the logical state of the code. We will see that the consequences may be that the program crashes (at best), or leads to incorrect behavior or results without hint of error (at worst).

These disturbing consequences are made possible because by accepting risk, Time Warp allows a logical process (LP) to execute a message that is inconsistent with the state of the LP (throughout the paper we will say that the *message* is inconsistent, although the fault of the inconsistency may actually lie with the LP). Such a message is destined to be canceled (or the LP is destined to be rolled back), but the LP has no way of knowing this. Processing the message, the state of the LP is combined with the data in the message to produce potentially nonsensical data and actions that have the ability to corrupt the operating environment in a way that will not be fixed by any rollback. Under the normal Time Warp execution rules there is no way, in general, for the LP to automatically recognize that the message is inconsistent with the LP state.

Any general purpose Time Warp system that simply links into the C or C++ run-time system has the potential to behave in this way. So long as a Time Warp system allows a modeler to include completely general code with the ability to write into non-checkpointed areas such as user library space or run-time system space, that system has an accident waiting to happen.

We have not come to bury Time Warp, but to praise it[1]. For a very important set of real simulation problems, it offers the only hope of high-fidelity parallel simulation. The reason we are emphasizing this particular aspect of Time Warp based simulation is that parallel simulation technology has matured to the point where major complex systems are being proposed that will use it. The complexity of these systems is such that verification and validation just assuming serial execution is a major exercise in software engineering. Our point is that engineering of parallel simulations additionally requires one to ensure the *safety* of the simulation in all states in which it might be placed. We do not believe it is widely understood that this is a larger and more difficult problem for optimistic parallel simulation than it is for serial or conservatively synchronized simulation. But it most certainly is, because the space of potential code states is much larger, and may include states that defy the model semantics or physical constraints of the modeled system.

The threat of this behavior was foreseen by the TWOS designers, and their solution is noteworthy[2]. They recognized that a possible consequence of processing an inconsistent message would be to engage in an action causing a runtime error, e.g., pass a negative value to a square root function, suffer a segmentation fault by indexing outside of array bounds, divide by zero, generate an arithmetic underflow or overflow, etc. . The TWOS system caught Unix signals generated by these occurrences and placed the faulting LP in an "error" state, from which it could be released by a rollback. An error was reported to the user only if the error state was committed, e.g., the GVT passed its time-stamp. This solution goes part way, but is not complete, at least in environments similar to Figure 1. By processing an inconsistent message, an LP can damage the heap, stack, or static data area, without generating a signal. The damage may not affect execution until long after its occurrence, and in any case, is permanent.

TWOS was implemented on four platforms (Mark II Hypercube, Mark III Hypercube, BBN Butterfly, UNIX network), the last three fit the model of Figure 1. Interestingly, in the Mark II Hypercube, TWOS was the fundamental operating system and so was better able to protect itself against erratic LP behavior. In all versions dynamic memory management was a touchy issue, so touchy in fact that TWOS programmers were advised to eschew pointers altogether. While the Mark II Hypercube version monitored stack usage (blowing the stack while executing an inconsistent message was a regular occurrence—remember that the Mark II had only 256K memory/node), the later versions did not. The threat of infinite loops was not addressed, although it was thought that this could be handled if a message arrival triggered an interrupt that was permitted to halt the LP's processing of an event. GTW does not filter Unix signals nor does it offer protection against any of these possibilities.

This paper first illustrates concretely how this problem can arise and discusses its ramifications, provid-

[1] *Julius Caesar*, III.ii, with apologies to Mark Antony.

[2] We acknowledge and thank Peter Reiher and Fred Wieland for providing us with the following description of the TWOS approach to dealing with this problem.

ing a review of a problem long known to Time Warp cognoscenti. Our contribution is to explore how to eliminate two sources of the problem—lagging rollbacks and stale state—and then to show that an LP can still be pushed into a state that is inconsistent with model semantics. We look at risk-free protocols, and show that if all LP code is safe with respect to "internal speculation", then the simulation is safe.

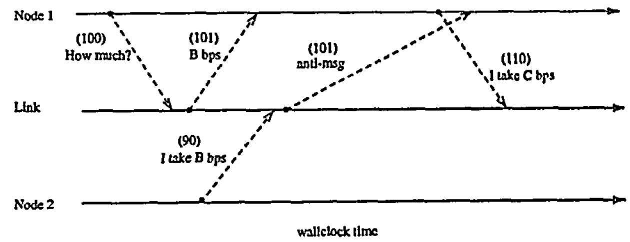

Figure 2: Timing diagram of how an inconsistent message may occur.

2 The Problem

We encountered the possibility of processing an inconsistent message when using GTW to simulate multi-cast resource allocation algorithms. In our model, a "link" LP is responsible for storing the current usage and availability of a network link. Such an LP may be queried "how much bandwidth do you have available", may be instructed "give me B bits per second bandwidth", and may be notified "I'm returning B bits per second bandwidth". "Node" LPs simulate the arrival of multi-cast requests; the multi-cast acceptance procedure involves (i) querying the link LPs that would be involved in the broadcast, analyzing the available bandwidth and (if the broadcast can be accepted) (ii) computing how much bandwidth should be requested from each link, (iii) instructing each link to allocate this computed amount to the broadcast. When the broadcast session is completed, the node LP restores the used bandwidth to the link LPs involved in the broadcast.

This paper resulted from our observation that link LPs with zero available bandwidth were sometimes instructed to allocate some non-zero amount of bandwidth to a new connection, and that link LPs whose bandwidth was completely available were sometimes instructed to add additional available bandwidth.

Close analysis revealed that these nonsensical situations were transient, either the LP was rolled back and when approaching the message again was in a state consistent with the message's instruction, or the message itself was annihilated. We also came to understand how these situations could arise. Figure 2 illustrates an example. At (simulation) time 100 a link LP would be queried by node 1 for availability information, and would report its available bandwidth, $B > 0$, in response. Then, the link LP is rolled back to an earlier time, say 90, by a message from node 2 that claims all B bits per second of the available bandwidth at time 90. Node 2 does not release this bandwidth any time soon. As part of the processing of that rollback, an anti-message is sent after the response message to node 1, but is recognized only after node 1 uses the first response in a decision to claim bandwidth C bits per second bandwidth from the link LP, before the anti-message catches up with it (N.B., our understanding of GTW is that event processing routines are atomic, meaning that so long as the anti-message arrives *after* the initiation of the event that generates the bandwidth claim message, that erroneous message will be sent). Node 1's bandwidth claim is destined to be annihilated, because node 1 is destined to be rolled back to deal with the link LP's new (empty) state. But, the link LP processes the bandwidth claim from a state where no bandwidth is available. Of course, eventually the link LP will be rolled back when the bandwidth claim message is annihilated.

So why should we be concerned if an LP processes a funny looking message, since that will all get sorted out with rollbacks? We ought to be concerned because the code processing that message may not have been designed to deal with anomalous messages. We ought to be concerned because by combining the state of the message with the state of the LP it is conceivable that the LP code will do any one of a number of bad things, including

- indexing outside of array bounds to damage memory or cause a segmentation fault,
- call a recursive function with arguments that do not yield a "bottom" to the recursion,
- enter an infinite loop,
- commit some numerical error such as divide-by-zero, (under/over)-flow, negative arguments to an library routine expecting positive arguments,
- delete an object from an empty data structure,
- just about any bone-headed error you've ever done or ever seen done in a C or C++ program.

The point should be clear, the danger of processing an inconsistent message is that the code developer has a certain model of system behavior in mind that derives from physical reality. Processing of an inconsistent message can push the LP into an unforeseen state that does not correspond to physical reality. While the code may be safe and correct in all states corresponding to physical reality (the only states in which it will find itself in a correct serial simulation or a conservatively synchronized parallel simulation), it might not be safe in non-physical states. As it processes an inconsistent message it may behave in unforeseen ways, damaging memory in the runtime system or even in the parallel simulation libraries. Damaged memory that is not considered to be part of the Time Warp "state" is damaged forever, and has the potential to crash the simulation, alter the behavior of the simulation, or corrupt some data associated with the simulation.

3 Safety through Methodology

Once we understood the source of the problem we saw that we could try to detect when a message was inconsistent and not act upon it, or just allow inconsistent messages to make the LP link state nonsensical and try to prevent the simulator from damaging the runtime environment. We now explore these options. Both solution approaches rely on the LP code doing checking that may detect that the LP should be suspended until rolled back to an earlier time. We presume then that the Time Warp system includes a call to implement self-suspension.

To require that a code check the logical consistency of each message it processes is philosophically unsettling. Time Warp has long been advertised as making synchronization transparent to the user; it most definitely is not transparent if in order to ensure that it runs without crashing we must augment the code with extensive consistency checks. While good software engineering practice calls for extensive checks, in practice little PDES code we've ever seen is actually written this way. It may be technically difficult or even impossible to always determine whether a message is consistent. Asynchronous code is hard enough to develop correctly when one knows what the correct messages look like, let alone having to anticipate and protect against the potential of arbitrarily inconsistent messages.

The second approach would include the TWOS signal trapping trick, but would also try to detect problems before they occur. Arguments for all library functions would have to be tested for reasonableness, to avoid things like taking the logarithm of a negative number. Nearly every memory access has to be tested for reasonableness, even if pointers are banned. Every single read or write to an array must first check that the index is within the array bounds. This holds true even if the array is declared to be simulation "state"—its memory space is in the heap somewhere and stepping outside of its confines can cause damage. But it gets worse. Some array writes are done tacitly, using C/C++ library calls. One can imagine a string being created using **sprintf** but for that string to exceed the allocated space because the string contents depend on an inconsistent message. The ability for detecting memory access problems has gotten quite comprehensive, it is conceivable that most such problems could be caught using the sort of technology behind commercial tools like Purify and insure++. To develop that technology independently for a parallel simulation engine is a daunting task.

A "catch-it-in-the-act" approach will also have to deal with bottomless recursion and infinite loops. There is a serious theoretical problem here, in that the well-known halting problem asserts that there is no algorithm that can take an arbitrary loop and detect whether it terminates. Protection from infinite looping has to be ad-hoc. To be completely safe one would have to be able to detect when processing of a message could lead to an infinite loop or bottomless recursion, and this is definitely non-trivial.

In summary, to rely on user-supplied consistency checking alone to filter out inconsistent messages is to court disaster. On the other hand, the implementation cost of monitoring executing C/C++ code to protect the runtime system from all possible ways of being trashed is overwhelming. In either case, substantially more work is involved to ensure that the parallel simulation runs safely than is required for a serial or conservatively synchronized simulation. It is clearly not enough to test a parallel simulation on one processor and expect it to always run safely in parallel.

The only hope for a comprehensive solution is to ensure that each LP's code is run in isolation and cannot damage the environment. At no time can the code be left to run on its own until "completion", because "completion" may never come. These are actually the same issues behind running Java applets safely; a truly comprehensive solution might be based on using Java to express and execute LP code, or use some other interpreted language. In the meantime, what can we do with the C/C++ based Time Warp systems?

4 Rollback Consistency and Stale States

A first step towards dealing with the consistency problem is to define our terms. The root cause of the problem in the multi-cast simulation was that a rollback or cancelation process occurred, but the LP reflected the post- (alternatively, pre-) rollback system state and the message reflected the pre- (alternatively, post-) rollback system state. This leads us to define rollback consistency.

One can think of Time Warp's execution on an LP as defining a tree, branches occurring where rollbacks are induced. Each rollback creates a new arm that is followed, until a rollback splits it. This is illustrated in Figure 3 where a sequence of four time-lines shows the evolution of this tree. The extent of each arm denotes the distance in simulation time the LP advanced before being rolled back. The solid line denotes the current state evolution path, dotted lines denote "dead" paths. Labeled black boxes identify points where the LP generated a message; arrows identify points in simulation time where the LP is rolled back. A circle marks the branch in the tree caused by a rollback, and is labeled with the simulation time of the rollback.

One can imagine sampling the state of the LP at different points in execution time, each sample would map to the end of the currently "active" execution arm; e.g., one could sample the state at the point message A is sent, and again at the point message B is sent. At the instant the state was sampled, the active tree arm would be at the label A in the first case, and B in the second case. The execution tree gives us a way of reasoning about the consistency of LP state sampled at two different points in simulation (and real) time. We'll say that two samples are *rollback consistent* if they lie on a common path (including branches) in the execution tree. Samples associated with messages C and D are consistent (even after the rollback at time 11), as are samples associated with messages C and E. No other pair of labeled points are consistent. Given two consistent samples, we can order them by simulation time.

Next we think about how an LP's state evolves as a function of the states of other LPs at various points in simulation time. Conceptually, at any instant we could describe how the state of LP i has been influenced by all other LPs, in terms of the data state of all other LPs at the various last (in real time) times they affected LP i. We can track these dependencies, as follows. Associate with each LP i a dependency vector (DV) that contains a component for every LP. The j^{th} component holds a code describing the last state of LP j to affect LP i. Initially the DV has null components, save for the LP's own component; the DV is updated when an ordinary message is accepted, to reflect the new influences on the LP's state (anti-messages are excluded from this discussion). This is accomplished by associating with every message the sender's DV at the time of transmission. As the message is processed we update the recipient's DV by merging it with the

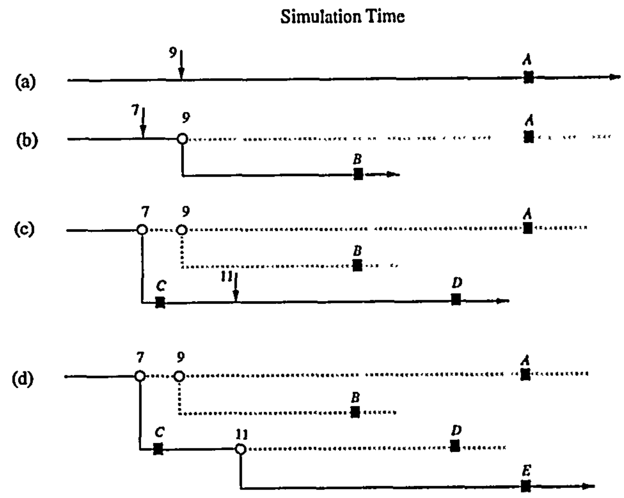

Figure 3: The evolution of an execution tree

message's DV. For each component we save the most "up-to-date" of the two. This way of thinking about dependency is basically Lamport's idea of distributed clocks [5]. It is important to keep in mind that these vectors are descriptive devices we use to reason about the system; the solutions we propose do not need to implement them.

Within this conceptual framework we can identify situations where inconsistencies arise. If for any component j the recipient's code for j is rollback-inconsistent with the message's component for j, then we know that LP j underwent a rollback that affected and is reflected in the state of either the sender or receiver, but not both. Returning to the time-line of Figure 2, we see that at the point that the link LP accepts the inconsistent message, the state of the link LP reflects the time 90 rollback, but that the link component for the message's DV does not reflect it. The link state component of the recipient is rollback-inconsistent with the link state component of the message.

A concrete example of a state code that detects rollback-inconsistency is a certain type of list. The first list element gives the simulation time of the sampled state. Following this is an ordered list of simulation times at which the LP was rolled back prior to sampling the state. Rollback times appear in this list in the order that the rollbacks were applied. For example, in Figure 3 the code for the LP state when message E is sent is the send-time of that message, followed by $(9, 7, 11)$. Let (t_1, L_1) and (t_2, L_2) two codes for an LP, where t_1 and t_2 are simulation times and L_1 and L_2 are ordered lists of rollback times. If these lists have different lengths, then the shorter list

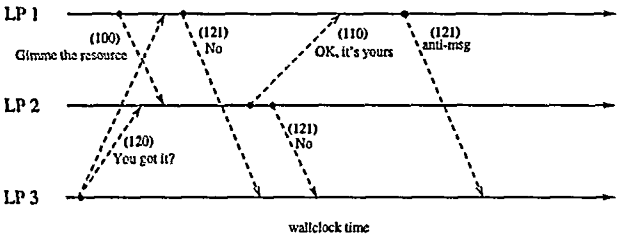

Figure 4: Inconsistency due to stale state.

is necessarily a proper prefix of the longer, e.g., (without loss of generality) $L_1 = (L_2, L_1')$. If there is any rollback time in L_1' that is less than t_2, the codes are rollback-inconsistent. (t_1, L_1) was sampled later in real-time than was (t_2, L_2), and reflects a rollback that brought the LP behind the simulation time of (t_2, L_2)'s sample. If instead there is no rollback in L_1' with a time-stamp less than t_2, then the additional rollbacks reflected in L_1 happen too late in simulation time to affect the sample at (t_2, L_2), and so the codes are rollback-consistent.

However, there is another source of inconsistency, due instead to what we'll call "stale states". An simple 3 LP example illustrates this possibility. A time-line for the scenario is given in Figure 4. There is a resource that at any point in simulation time is held by either LP 1 or LP 2. The resource is held by an LP until the other requests it, and some random time after the request, the resource is released to the LP requesting it. Now imagine that at time 100 LP 2 has the resource and LP 1 requests it. LP 1 does not know when it will actually acquire the resource, and so continues optimistically on to an event at time 120, under the assumption that it does not have the resource at time 120. The event processed at time 120 responds to a query by LP 3, "Do you have the resource now?". Assuming not, LP 1 replies "No" at time 121. Sometime after this exchange LP 2 decides to yield the resource, at simulation time 110. Immediately after sending the "It's yours now" message to LP 1, it processes a query event from LP 3 at simulation time 121. "No", it replies. Upon receipt of this second negative response, LP 3 is in an illogical (with respect to the model semantics) state. We cannot pin this situation on a rollback inconsistency, because no rollbacks have yet occurred.

The problem in this case is that LP 3's state is "stale" with respect to the query response message it receives from LP 2. It is stale in the sense that some action has been initiated that ultimately will roll LP 3 back, that the LP 2 data dependency component of LP 2's message to LP 3 is of a state that follows that initiation, but the LP 2 component of LP 3's data dependency vector will be affected by that initiation, but has not yet been affected by it.

More formally, if M is a message processed by LP j, we'll say that LP j's state is stale with respect to M if M's dependency vector component for some LP k reflects a state of LP k that follows its transmission of a message that initiates a rollback, the penultimate result of which is to rollback LP j, but which result has not yet occurred. The definition of M being stale with respect to LP j's state is entirely similar, save that the penultimate result is that M is annihilated.

Rollback-inconsistency and stale state are related concepts in that rollback inconsistency occurs when a completed rollback makes two sampled states inconsistent, whereas stale state occurs when an initiated but as yet uncompleted rollback chain makes two sampled states inconsistent. For example, whereas states associated with C and D in Figure 3 are rollback consistent, once the first message responsible for initiating the rollback chain that causes the rollback at time 11 is sent, state D becomes stale, even before the time 11 rollback is recognized.

The problems of rollback inconsistency and stale state can be both eliminated by the simple mechanism of requiring message acknowledgments. The discussion below assumes the use of aggressive cancelation (also a modified lazy cancelation is possible so long as before a rolled back LP sends any new post-rollback message, all previously sent messages with larger time-stamps are aggressively canceled, with acknowledgments all received).

When an LP sends an ordinary message, it blocks from further processing until it receives an acknowledgment for that message (actually, all that is required is that it not send any new message before that acknowledgment is received). If a message (ordinary or anti-) is placed in LP j's input buffer and does not cause rollback, that message is acknowledged immediately. If instead the message triggers a rollback, say at simulation time t, LP j does not acknowledge the message until it has itself received an acknowledgment for all anti-messages sent with time-stamps greater than t.

We must argue that this mechanism does not cause deadlock, and that it does indeed eliminate rollback inconsistency and stale data. Freedom from deadlock is easily seen, by considering the unacknowledged message with largest receive-time. It ultimately must be acknowledged, breaking any deadlock cycle.

Theorem 1 *Under message acknowledgments for anti-messages, rollback-inconsistency is eliminated.*

Proof: For the sake of contradiction, suppose a message M is delivered to LP k such that the message's dependency vector is rollback-inconsistent with LP k's dependency vector in component j. Without loss of generality (and using the list code described earlier), suppose that LP k's DV code for LP j, (t_1, L_1), is a later (with respect to wallclock time) reflection of LP j's state than is the corresponding component (t_2, L_2) in M's DV. Rollback-inconsistency implies that $L_1 = (L_2, L_1')$ for some non-empty list L_1', and that L_1' contains some rollback time t_3 with $t_3 < t_2$. We take t_3 to be the first such rollback time in L_1'. Ultimately we can trace back the appearance of code (t_2, L_2) in M's DV to a transmission at time t_2 by LP j. When LP j processes the rollback at t_3 after that transmission, it does not send any further messages before receiving an acknowledgment for the anti-message it sends to cancel the time t_2 message. That anti-message, possibly triggering other anti-messages that must be acknowledged, erases all dependence of any LP on the state of LP j reported in the time t_2 message. LP j cannot advance to any state represented by code (t_1, L_1) before that cancelation is complete. Hence, code (t_2, L_2) may not be associated with M if LP k's code is (t_1, L_1), establishing the contradiction and proving the theorem.
□

It is worth noting that a measure of safety can be gained just by acknowledging anti-messages. Additionally requiring acknowledgments for ordinary messages brings freedom from stale states as well.

Theorem 2 *Under message acknowledgments for ordinary and anti-messages, the system is free from stale states.*

Proof: Suppose that LP k's state is stale with respect to message M in the dependency vector component for LP j. This means that LP j sent some message M' before achieving the state reflected in M's DV component for LP j, a message that triggers a rollback chain that will ultimately affect LP j, but has not. This cannot occur, as LP j awaits an acknowledgment for M' before sending any further messages. A similar argument shows that M cannot be stale with respect to LP j's state.
□

Two important points should be noted here. First, Time Warp pioneers recognized that if the system contained messages that would cause an LP to rollback and messages that would cause it to move forward, then the rollback should have precedence. This could be done heuristically by always processing anti-messages before other messages. Some schemes even proposed preemptive rollbacks of all LPs within some "distance" of a temporal error without bothering to see if anti-messages would indeed trigger rollbacks there. Our acknowledgments are the logical extension of such thinking. Second, waiting for acknowledgments before allowing an LP to send another message need not impact performance greatly. This is a matter of latency hiding. If there are many LPs on each processor, then after suspending one we may well be able to find many that have useful work to perform while the first is blocked. In any case, the requirement is that an LP not send another message without appropriate acknowledgments, not that it cannot process. It is viable to allow the LP to continue executing, and buffer its messages until appropriate acknowledge releases are obtained.

5 Safety and Risk-Free Protocols

Despite the measure of safety offered by requiring acknowledgments, it is easy to see that a "risky" message can still be drive an LP into error. For instance, the first message LP A sends to LP B may be inconsistent with LP B's initial state, *unless* LP A first receives and processes a message from LP C. Risky processing allows LP A to befoul LP B with the errant message.

In our view, the real danger of allowing risky messages is that in a complex code the space of possible nonsense states into which an LP might be driven is too large to anticipate. Intuition will guide a model developer so long as the LP data state can be assumed to reflect a real possible state in the modeled physical system. Even though banning risk *still* allows one to get into states that are inconsistent with model semantics, we observe that if the model code is safe under all "internally speculative" scenarios, then it is safe under a risk-free protocol. This point is so obvious it hardly needs mentioning, yet given the perhaps ill-considered impetus in the PDES community to attempt every bit of parallelism possible, perhaps it does need mentioning.

Constraining synchronization to be risk-free vastly simplifies the programmer's job in verifying the safety of the program. For, let $\mathcal{S}$ denote the set of "real" LP data states, those corresponding to physical reality. During testing and verification of correctness, one ensures that the transitions between states in $\mathcal{S}$ is correct in response to a set of *external* messages M_E and in response to a set of *internal* messages M_I. If a risk-free protocol is used, to test or verify safety, one needs only to ensure that the LP is safe within all states $\mathcal{S}' \supseteq \mathcal{S}$, obtained by considering all ways of starting with a state in S and repeatedly making transitions by accepting messages from M_I. Contrast this with the need to test or verify the set of states $S'' \supseteq S'$, obtained by considering all ways of starting with a state in S and repeatedly making transitions driven by messages of any kind or content. In the former case it is reasonable to expect the code developer to have intuition about the behavior of the LP code in S', in the latter case it is a much harder job.

6 Conclusions

We remind the community that optimistic processing implies that LPs may be temporarily driven into nonintuitive states. We point out that with current trends in using C and C++ as the basis for parallel simulation, this creates a danger of damaging the simulation while it executes in one of these states, for the executing code has access to portions of the stack, heap, and static data areas that are not checkpointed.

Our concern is that parallel simulation is on the threshold of being used to build large complex models, but that the safety ramifications of optimistic processing in general, and risky messages in particular are not well understood by those who would build those simulators. For, the model code has to be resilient enough not to crash the system, even when pushed into transient, but unexpected non-physical non-intuitive states.

We observe that lagging rollbacks and stale states are two important causes of an LP entering a nonsense state, and show that requiring acknowledgments from anti-messages eliminates lagging rollbacks, and that additionally requiring acknowledgments from ordinary messages eliminates stale states. We note that the problem of verifying an LP code's safety is much reduced if it can be assumed that a risk-free synchronization protocol will be used. The programmer needs only consider the effects of speculative computation that are entirely internal to the LP. This eliminates the need to anticipate against processing arbitrarily illogical messages.

Optimistic synchronization provides the only way some important problem classes can be simulated with fidelity on parallel machines. However, it is important to realize that the software engineering burden of using Time Warp is higher than for serial simulation. So long as the LP code can adversely affect its operating environment, synchronization concerns are **not** transparent to the modeler.

Acknowledgments

This research was supported in part by NSF grants CCR-9308667 and CCR-9625894, and DARPA Contract N66001-96-C-8530.

References

[1] College of Computing, Georgia Institute of Technology. *MetaTeD—A Meta Language for Modeling Telecommunication Networks*, October 1996.

[2] S. Das, R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. GTW: A Time Warp system for shared memory multiprocessors. In *1994 Winter Simulation Conference Proceedings*, pages 1332–1339, December 1994.

[3] P. Dickens and P. Reynolds, Jr. SRADS with local rollback. In *Distributed Simulation*, volume 22, pages 161–164. SCS Simulation Series, Jan. 1990.

[4] D. R. Jefferson, B. Beckman, F. Wieland, L. Blume, M. DiLorento, P. Hontalas, P. Reiher, K. Sturdevant, J. Tupman, J. Wedel, and H. Younger. The Time Warp Operating System. *11th Symposium on Operating Systems Principles*, 21(5):77–93, November 1987.

[5] L. Lamport. Time, clocks, and the ordering of events in distributed systems. *Communications of the ACM*, 21(7):558–565, July 1978.

[6] P.F. Reynolds, Jr. Comparative analyses of parallel simulation protocols. In *Proceedings of the 1989 Winter Simulation Conference*, Washington, D.C., December 1989.

[7] J.S. Steinman. SPEEDES: Synchronous parallel environment for emulation and discrete event simulation. In *Advances in Parallel and Distributed Simulation*, volume 23, pages 95–103. SCS Simulation Series, Jan. 1991.