*Proceedings of the 2025 Winter Simulation Conference*
*E. Azar, A. Djanatliev, A. Harper, C. Kogler, V. Ramamohan, A. Anagnostou, and S. J. E. Taylor, eds.*

# THE NEEDLE IN THE ONE-BILLION EVENT-HAYSTACK: A DEEP-COPY APPROACH FOR CHECKING REVERSE HANDLERS IN PARALLEL DISCRETE EVENT SIMULATION

Elkin Cruz-Camacho[1] and Christopher D. Carothers[1]

[1]Dept. of Computer Science, Rensselaer Polytechnic Institute, Troy, NY, USA

## ABSTRACT

Parallel Discrete Event Simulation (PDES) enables model developers to run massive simulations across millions of computer nodes. Unfortunately, writing PDES models is hard, especially when implementing reverse handlers. Reverse handlers address the issue of saving the entire state of Logical Processes (LPs), a necessary step for optimistic simulation. They restore the state of LPs through reverse computation, which reduces memory and computation for most common models. This forces model designers to implement two tightly coupled functions: an event handler and a reverse handler. A small bug in a reverse handler can trigger a cascade of errors, ultimately leading to the simulation's failure. We have implemented a strategy to check reverse handlers one event at a time. In a mature PDES model, we identified reverse handling bugs that would have required processing approximately 1 billion events to manifest, which is nearly impossible to detect using traditional debugging techniques.

## 1 INTRODUCTION

Parallel Discrete Event Simulation is a well-studied framework for modeling and simulation, with a focus on scaling up simulations for parallel execution (**?**). Unlike time-stepped simulations that advance uniformly in a locked-step fashion, PDES makes progress by processing events with timestamps at arbitrary, incremental time intervals. This makes PDES particularly effective for models with sparse temporal activity patterns like arrival and departure of flights, monetary transactions (**?**), agent-based modeling (**?**), city traffic simulation (**?**), and computer network simulation (**?**).

PDES models are composed of two simple elements: Logical Processes and events. An LP is a self-contained entity with local state variables that it alone can access. Any global variables in the system are treated as immutable. LPs communicate through *events* scheduled at specific virtual times. When scaling up, each LP is assigned to a different Processing Element (PE). A PE typically corresponds to a computing core, either in the same computer node or across the network on other nodes. To allow the simulation to advance and keep all LPs synchronized, two synchronization strategies exist: conservative and optimistic approaches. Conservative approaches (**?**; **?**) advance the simulation only when it is provably "safe" to do so, for example, allowing the simulation on each PE to progress only by a short virtual time step. Any violations of causality terminate the simulation. In contrast, optimistic approaches, such as Time Warp (**?**), allow speculative execution beyond safe boundaries. In Time Warp, each PE processes as many events as possible until communication with other PEs reveals that remote events should have been processed earlier. At this point, any event that happened after the unprocessed remote events is rolled back, undoing any effects and thus restoring causal consistency.

Notice that even though some models tolerate a certain amount of accumulated noise from lack of proper rollback (**?**), most models are highly sensitive to errors introduced on rollback, as we showcase in our experiments, in Section 3.

The Rensselaer Optimistic Simulation System (ROSS) is a framework and library for implementing and executing PDES models (**?**). ROSS can be scaled up to millions of cores through optimistic execution (**?**). To maintain simulation validity, ROSS incorporates a tie-breaker mechanism that guarantees deterministic

execution when multiple events share identical timestamps, ensuring reproducible results across different runs (**?**).

Since saving the entire state of an LP before event processing would be expensive for small simulations and prohibitively expensive for large-scale simulations, ROSS employs *reverse computation* to rollback the state of the LP. For this, developers must implement both (forward) event handlers and reverse event handlers. Reverse event handlers apply the inverse operations of event handlers to restore the previous state of the LP. This approach dramatically reduces memory and compute requirements, but creates a significant programming challenge.

Correctly implementing reverse handlers is significantly tricky. Even minor mistakes, such as failing to decrement a counter or restore a flag, are intolerable because the decision on when and how many events to rollback is non-deterministic. Thus, minor errors introduce non-deterministic changes to the model, which can lead to erroneous LP states and incongruent model states. In a turn of events, it is not segfault bugs that are the hardest to detect, but minor ones. Minor mistakes are much harder to catch, correct or even detect because they may not cause immediate simulation failures and can introduce subtle inconsistencies that propagate through the simulation, leading to invalid results that appear plausible.

Identifying subtle reverse handler bugs is extremely challenging, regardless of the debugging approach used. Errors might only occur after massively scaling a simulation, i.e, they could manifest after processing hundreds of millions of events in parallel across dozens of computing nodes. This makes manual tracing impractical. Furthermore, the distributed nature of PDES obscures the causal relationships between the initial error and its eventual manifestation, where the initial inconsistency might occur in a different computing node from which it manifests. This is why finding and fixing bugs requires significant developer effort, from determining the conditions under which the bug manifests to tracking down where the bug actually occurs.

In this paper, we present a straightforward yet effective strategy for checking the correctness of reverse handlers in optimistic PDES. Our approach systematically validates that each reverse handler properly restores LP states, catching inconsistencies before they propagate through the simulation. We implement and test our strategy using ROSS and CODES — a mature HPC network simulation framework built on ROSS. CODES presents an ideal test case due to the maturity of the code, the complexity of its LPs, and its (over)use of dynamically allocated memory.

The remainder of this paper is structured as follows. Section 2 details our strategy, including considerations and limitations. Section 3 presents experimental results demonstrating our approach's effectiveness, including detection of subtle bugs in CODES that would have required processing approximately one billion events before manifestation. Section 4 goes through the literature of debugging PDES. Finally, Section 5 concludes with a summary of our results.

## 2 STRATEGY TO CHECK REVERSE HANDLERS

This section describes our approach for validating reverse handlers in optimistic PDES. We first present our strategy (Subsection 2.1), then some implementation details in ROSS (Subsection 2.2), and finally, we discuss important considerations and limitations (Subsection 2.3).

### 2.1 Strategy

The fundamental challenge in developing reverse handlers is ensuring they perfectly restore an LP's state after rollback. Our approach systematically checks this property for every processed event during simulation. The strategy is straightforward: save the state of the LP before processing an event, then process the event and rollback it with the reverse event handler. The new state of the LP must be the same as the state that was saved! Algorithm 1 illustrates the strategy.

Telling the developer what failed and why helps reduce debugging time compared to traditional approaches, where identifying the source of non-determinism might require analyzing billions of events.

**Algorithm 1:** Strategy to check for proper behavior of reverse handler.

```
/* When processing each event e                                    */
prev_state := deep copy LP state (including random number seed state);
process(LP, e);
post_state := deep copy LP state;
rollback(LP, e) ;                                  // Using reverse-handler
check(prev_state, LP) ;              // Report failure if states are not equal
process(LP, e);
check(post_state, LP) ;              // Report failure if states are not equal
cleanup_state(prev_state);
cleanup_state(post_state);
```

Traditionally, one would try to find smaller model configurations that display erroneous behavior and, from there, poke the model until something reveals the source of the bug. This is not only tedious and labor-intensive, but also highly impractical when a bug is spurious and only appears at large scales. Thus, we must provide the developer with as much information as possible about a failure when comparing two LP states that should be identical (the previously stored state and the current state).

Therefore, when a check fails, execution must halt immediately, and diagnostic information will be generated. In our case, useful debugging information includes: the LP identifier, current event count, type of check failure (pre-state mismatch, post-state mismatch, or RNG inconsistency, the contents of both states, and the event processed. We found that this information helps the developer find exactly what part of the LP state was not properly restored.

Notice that: even though the reverse handler is designed to be executed only when rollback is necessary, i.e, in optimistic parallel simulations, we can run a simulation in sequential mode and inject our strategy at the event process. This significantly simplifies a possible parallel implementation of our strategy, yet it can capture most reverse handler bugs. Note that checking the reverse handlers in sequential instead of parallel environments might lead to even subtler bugs not being captured. We explore these situations in Subsection 2.3.

Implementing this strategy on complex models that allocate memory dynamically requires significant involvement from the model developer. For simple LP structures that contain only primitive data types and statically allocated memory, bitwise comparison of LP states may be sufficient. However, dynamic memory allocation, nested structures, and complex data structures might require additional effort to copy and compare the state of LPs. In these cases, developers must implement the following functions:

1. **Deep-copy** to create a complete copy of an LP.
2. **Deep-free** to clean and free a copy of an LP.
3. **Deep-check** to compare original and restored LP states.
4. **Deep-print** to generate meaningful diagnostic output when inconsistencies are detected.

### 2.2 Implementation Experience

Integrating our strategy into ROSS was straightforward, requiring approximately one week of developer time to implement the core functionality and fix a bug in the tie-breaker mechanism that was highlighted by the strategy. The ROSS architecture, because of its maturity, has been designed in a modular way that allows this debugging strategy as a new event scheduler execution mode. Our implementation added a fourth execution strategy: the sequential reverse handler check. ROSS previously implemented four other modes: sequential, parallel optimistic, parallel conservative, and optimistic debug. (Optimistic debug is an execution mode implemented in ROSS that helps developers find reversibility bugs.). It operates in a sequential mode, similar to ours. It runs the simulation forward until there is no more space to schedule

additional events, and then rolls back all events to the start of the simulation. Without being able to check which event cascaded a bug, this mode provides the developer with far less information to determine the source of a reversibility bug. However, it may allow for finding reversibility errors if the final LP state differs from the initial LP state.

However, implementing the necessary deep-copy, deep-check, and deep-print functions for complex models proved considerably more challenging. For our CODES case study, this process required approximately one month of development effort and resulted in approximately 3,500 lines of additional code. This effort was driven by two primary factors. First, ROSS is implemented in C and allocates memory for model LPs without knowledge of how that memory will be used. This separation between the simulation engine and the model implementation allows ROSS to be model-independent and as fast as possible, but complicates comprehensive state management.

Second, while CODES incorporates some C++ components, it predominantly uses C-style data structures and memory management. This includes classical data structures implemented in C headers, such as a Linux kernel-inspired linked list (derived from Linux's list.h) and a hashtable implementation built on top. These deeply embedded data structures required careful, manual implementation of copy, check, and print operations.

For models implemented in languages with built-in serialization, reflection, or deep-copy capabilities, this implementation burden would be significantly reduced. However, for C-based models like CODES, developers must manually track and manage all dynamically allocated memory and complex data structures within their LPs. Despite this implementation overhead, the benefits of detecting subtle bugs that might otherwise require billions of events to manifest justify the development investment.

### 2.3 Limitations and Considerations

While our strategy effectively identifies many reverse handler errors, several important limitations should be acknowledged. We identify five such cases:

1. Bugs might only appear in parallel execution,
2. Developers need to determine what in the LP state has to be copied and checked,
3. Checking a single event at a time might miss some (subtler) bugs,
4. The overhead of the deep LP operations, and
5. Not all event types in all conditions might be checked.

Let us explore each one of these limitations:

First, we implemented the strategy in sequential mode and thus it cannot detect bugs that manifest only during parallel execution. Two significant examples of where parallel execution could bring up errors that do not occur in sequential include: incorrect handling of global variables, and failure to correctly process "impossible states" that arise during optimistic simulation.

To prevent non-deterministic errors showing up, global variables should be seldom used in PDES. There are two possible ways to handle global variables in PDES: to use them as read-only, or to alter them only when processing non-rollbackable functions, such as the commit function or an operation at the GVT computation. The commit function is an additional function on an LP, besides the event handler and reverse event handler. The commit function is only executed after it can be guaranteed that the event will not be rolled back. The commit function is only executed once while the event handler can be executed more than once. Thus altering any global variables should only be done, if absolutely necessary, at commit functions.

As Nicol and Liu noted in "The Dark Side of Risk" (**?**), optimistic simulation can temporarily produce LP states that would never occur in sequential execution due to out-of-order event processing. When these states later encounter rollbacks, reverse handlers must properly restore previous states even from these "impossible" configurations. Our implementation is sequential and cannot replicate these conditions.

Second, developers must carefully determine which components of the LP state to check. Not all components require to be reversed. Some fields may serve as temporary storage used only during event processing. Others might be designed to track information needed precisely for the reverse computation itself. Some other components may be updated only by the commit function, making them irrelevant to the reverse handler check. Thus, the developer has to determine which components make up the model and have to be rolled back, and which do not need to be taken into consideration.

Third, our strategy checks only the processing and rollback of a single event. Some rare bugs may only appear when processing multiple events and rolling back all of them. Consider an LP containing a fixed-size array and an index tracking the number of values stored in the array. Let's assume that an event handler for this LP does one of two things: adds a value to the array (and increments the index) or removes a value from the list (and decrements the index). A naive reverse handler that only adjusts the index without restoring array contents would pass our single-event processing check! Yet it will fail if we process two events: one removing an element from the array and one adding an element to the array. Notice that rolling back these two events using our naive reverse handler would result in an array with (potentially) different contents than what it started with before processing the events.

Similarly, dynamic memory allocation can create subtle issues. If an event handler deallocates memory and then a reverse event handler reallocates memory of the same size, the operating system might reuse the same memory address. In this scenario, we might assume that the new LP with recycled memory and a copy of the LP look to be the same LP state, but in fact, they are the same just by happenstance. If the reverse handler were given a different chunk of memory, we could have realized that the reverse handler did not correctly recreate the state of the LP.

Four, copying complex LPs can be a time-consuming process. After all, the purpose of reverse computation is to forgo the memory and computation burden of copying and restoring LP states. This overhead, as we explain in our results, can reduce the simulation speed by one or two orders of magnitude. This might be an issue if an event is processed only after billions of previous events have been processed, and thus it could take a significant amount of time before we run a long enough simulation that triggers the event.

Five, our implementation relies on running a simulation and checking whether the reverse handlers work properly on that specific run. If a specific simulation path or event type is not executed, it cannot be checked. Because our strategy requires an upfront effort from the developer, subsequent checks are cheap. Thus, if the developer finds an initial configuration that seems buggy, they can easily check whether there are any reverse event handler bugs by running the strategy on the new configuration.

An advantage of our methodology is that it is platform-agnostic; the reverse handler check can be implemented in any (synchronous and time-warp-based) PDES library. Yet we ran all our experiments in a Linux environment. Although this might limit the applicability of our results, we posit that the core limitations we have discussed will be evident in any system. It is possible that memory managers on different platforms affect, in subtle ways, the memory recycling scenarios we discussed in limitation three. What this means is that a cross-compatible PDES model running on two different platforms might encounter different reverse handler bugs.

Despite these limitations, our approach captures many common reverse handler bugs based on our experience. In the next section, we present experimental results demonstrating the effectiveness of our technique on a mature, complex PDES model, where it identified subtle errors that would have required processing approximately one billion events before manifesting as simulation failures.

## 3 EXPERIMENTS AND RESULTS

To evaluate our reverse handler checking strategy, we applied it to CODES, a large-scale, mature PDES application. This section describes our experimental environment (Subsection 3.1), the application of our strategy to increasingly complex models (Subsection 3.2), and notable bugs we discovered and fixed (Subsection 3.3).

### 3.1 Experimental Setup

All experiments were conducted on an NVIDIA Grace Hopper GH200 chip equipped with 72 physical cores and 500 GiB of RAM located at the Center for Computer Innovation (CCI) in Rensselaer Polytechnic Institute (RPI).

CODES, our test case, is a mature and complex High-Performance Computing (HPC) network simulator implemented in C/C++ built on top of ROSS. CODES simulates the behavior of large-scale HPC networks under various traffic patterns, some of which mimic real scientific applications executed in HPC systems. It implements detailed models of network topologies and routing protocols.

In our experiments, we simulated dragonfly networks (**?**) of different sizes (72, 1056 and 8448 nodes) running two representative scientific applications: Jacobi3D and MILC. MILC was run for a total of 200 iterations and Jacobi3D for 25. These applications were configured to interfere with each other by allocating them randomly in the simulated network (**?**). In CODES there are three LP types: router nodes, computer nodes, and terminal nodes (which are Network Interface Controllers (NICs) for the computer nodes). Table 1 summarizes the topology parameters and LP allocation for each network configuration.

Table 1: Dragonfly network configurations and LP allocation.

| Nodes | Topology Parameters | | | | LP Distribution | | | |
|---|---|---|---|---|---|---|---|---|
| | p | a | h | g | Compute | Terminal | Router | Total |
| 72 | 2 | 4 | 2 | 9 | 72 | 72 | 36 | 180 |
| 1056 | 4 | 8 | 4 | 33 | 1056 | 1056 | 264 | 2376 |
| 8448 | 8 | 32 | 1 | 33 | 8448 | 8448 | 1056 | 17952 |

p = terminals per router, a = routers per group, h = global channels per router, g = total groups

For optimal PDES performance, LPs in CODES are typically mapped to the same PE if they communicate frequently. Our LP allocation strategy follows a repeating pattern: we allocate compute node LPs, terminal (NIC) LPs, and one router LP to consecutive positions, then repeat this pattern across all routers in the group. For example, for the 72-node network, we allocate ID 0 and 1 to the 2 first two compute node LPs, then ID 2 and 3 to the first 2 NIC LPs, and lastly ID 4 to the first router. This ensures that tightly coupled LPs (particularly compute nodes and their corresponding NICs) remain co-located when possible.

LPs belonging to the same dragonfly group should ideally be assigned to the same PE to minimize inter-PE communication and reduce rollbacks. For the 72-node network with 9 groups, optimal PE counts are 1, 3, or 9, as these are divisors of the group count and allow complete groups to be assigned to individual PEs. Similarly, for both 1056 and 8448-node networks with 33 groups, optimal PE counts are 1, 3, 11, and 33. Table 2 shows how Processing Elements (PEs) are allocated across the available hardware cores for different experimental configurations. Each PE is allocated to its own computer core; we do not use threads and all communication is done through MPI.

Table 2: Processing element allocation on hardware architecture.

| Network | PEs Used | LP Density |
|---|---|---|
| 72-node | 3 | 60 LPs/PE |
| 72-node | 9 | 20 LPs/PE |
| 1056-node | 33 | 72 LPs/PE |
| 8448-node | 33 | 544 LPs/PE |
| 8448-node | 55 | 326 LPs/PE |

In our stress test configuration, we deliberately ran the 8448-node network on 55 PEs, which forces suboptimal LP distribution. Since 33 groups cannot be evenly divided among 55 PEs, some compute node LPs become separated from their corresponding NIC LPs, and router nodes within the same dragonfly

group are split across different PEs. This suboptimal mapping increases the frequency of rollbacks, creating more opportunities to test reverse handler correctness under stress conditions.

### 3.2 Bug Detection and Fortified Models

Our journey into debugging CODES started when we tried scaling CODES up from a small 72-node simulation into a 1056-node simulation and found out some jarring inconsistencies. When running a 72-node dragonfly network on 3 PEs, everything appeared to work properly, with consistent results across multiple runs (see left side of Table 3). However, when we scaled up to a 1056-node configuration running on 33 PEs, we encountered significant non-determinism.

As shown in Table 3 (right side), the number of net events processed in the 1056-node configuration is not stable across different runs with identical inputs. More concerning still is that in 5 out of 10 runs the simulation terminated prematurely without segfaulting or producing any clear error messages.

Table 3: Total number of net events after running two dragonfly networks ten times while keeping the initial conditions the same. Running CODES before fixing any reversibility bugs.

| Run # | 72-node | 1056-node | |
|---|---|---|---|
| | Net events processed | Net events processed | Premature end |
| 1 | 357636915 | 8129411742 | No |
| 2 | 357636915 | 8129452588 | No |
| 3 | 357636915 | 2949055904 | Yes |
| 4 | 357636915 | 8129565619 | No |
| 5 | 357636915 | 1604444856 | Yes |
| 6 | 357636915 | 8129743388 | No |
| 7 | 357636915 | 8129557666 | No |
| 8 | 357636915 | 8058353041 | Yes |
| 9 | 357636915 | 1604444856 | Yes |
| 10 | 357636915 | 3806348122 | Yes |

Figure 1 illustrates this premature termination. The top graph shows a run where both applications (MILC and Jacobi3D) completed all iterations as expected, while the bottom graph shows Jacobi3D stopping prematurely while MILC completed its run. Notice that the runtime of a 1056-node simulation that ran to completion was of about 1700 seconds. Saving and going through traces for 33 cores over the span of 30 minutes is a gargantuan task.

Facing these issues, we implemented all necessary deep functions for the LPs in CODES. We then ran the new sequential mode (reverse handler check) on a small 72-node dragonfly network capped to a small number of iterations for each application (MILC and Jacobi3D). Despite the smaller configuration showing no obvious signs of error in normal parallel execution (3 PEs), our approach (running in sequential) uncovered five significant bugs in CODES' reverse handlers.

The fixes for these bugs required only about 20 lines of code changes, yet they completely eliminated the non-deterministic behavior in the larger simulations. After implementing the fixes, we ran the previously problematic 1056-node configuration again, and it executed deterministically every time, consistently processing exactly 8,129,787,720 net events.

To further validate our fixes, we conducted a stress test with an even larger configuration: an 8448-node dragonfly network running on 55 PEs. This configuration deliberately increased the number of rollbacks through suboptimal LP-to-PE mapping. Even under the larger model with more rollbacks and possible erroneous states to uncover, the simulation ran deterministically every time (same number of total net events and network statistics across multiple runs).

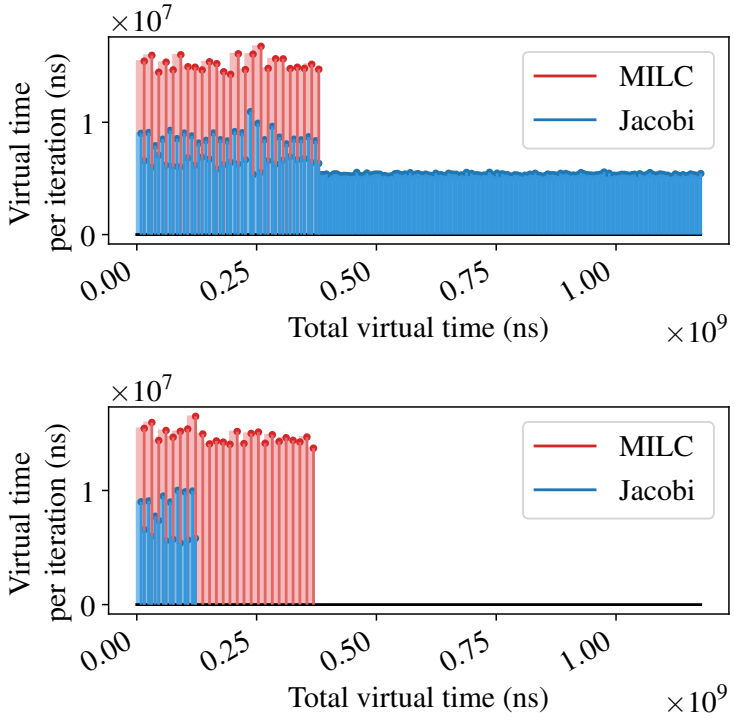


Figure 1: Number of job iterations and their duration for MILC and Jacobi in 1056-node dragonfly. Top, MILC and Jacobi3D ran to completion (run number 4 in Table 3); bottom, Jacobi3D stopped prematurely and MILC ran to completion (run number 9 in Table 3).

### 3.3 Types of Bugs Discovered

Our strategy allowed us to find several bugs in CODES in record time (even when accounting for the time that it took to implement the deep LP functions). These bugs can be classified into:

1. Catastrophic: A single bug caused the premature termination of the simulation. Each iteration for each of the applications consists in a sequence of (simulated) MPI operations, some of which are "MPI all reduce" operations. To handle them, a counter inside each LP would keep track of them. The counter was never properly rolled back by the reverse handler. Yet, the reverse handler was seldom executed, leading to the false believe that the bug did not exist when executing the small simulations.
2. Noise/nondeterminism: These bugs would not cause the premature termination of the simulation. Instead, they would add noise and small errors to the simulation. We found two types of this kind of bug:
   (a) Failure on counter rollback: There were multiple instances where reverse handlers failed to properly decrement or increment counters during reverse computation. These small arithmetic errors would accumulate over time.
   (b) Determining the conditional branch taken by the event handler: In one occasion, an assumption on a data structure lead to a reverse handler that would reverse the incorrect conditional branch taken by the (forward) event handler.

A particularly subtle bug to highlight is one that barely produced any indications of its existance. The conditions for it to trigger were rare. After incrementally checking most LP types, and fixed most bugs,

we noticed that the number of net events between two runs of a 1056-node network differed by only two events out of eight billion events! As with most bugs we found, the error came from a failure to decrement (or increment) a single counter.

Besides the bugs found in CODES, the strategy helped us to find out a bug within ROSS itself. As a first test of the utility of the strategy, we tested it on a simple model (phold). We were able to catch all bugs we introduced to phold for testing, but weirdly enough, the strategy would detect a bug outside of phold. The bug came from a flaw in the implementation of the tie-breaker mechanism. The tie-breaker mechanism ensures deterministic event processing when multiple events share the same timestamp by assigning a random signature value during event scheduling. We identified a specific condition where the random number generator state was not properly restored: when an event was scheduled beyond the simulation end time.

ROSS optimizes memory usage by immediately discarding events that are scheduled beyond the simulation end time instead of storing them. Yet, ROSS failed to track how many of these events had been dropped. Consequently, when rolling back the event that scheduled these dropped events, the random number generator state was not properly restored. In simulations without timestamp ties, this bug would never manifest. However, for models with event timestamp ties, the improper rollback of the random generator state could lead to subtly different event sequences across runs, breaking the simulation's deterministic guarantee.

These bugs had been dormant possibly for over a decade. Git blame is an utility that allows us to see when an implicated line of code was last modified. Thus, we are able to determine that the ROSS tie-breaker bug was dormant since the initial implementation of the mechanism, in 2021, 4 years ago. Dating the CODES bugs is substantially more difficult because of the sheer magnitude of changes in the model and its assumptions since any implicated line was introduced. Although the implicated lines could be traced back to 2015 (10 years ago), it is not clear whether the bug was present at the moment of implementation or if it was introduced later when an assumption in the code was broken. Regardless, reversibility bugs can stay dormant for years until they are finally triggered.

It is then not at all surprising that we uncovered these bugs in mature and complex PDES models, which demonstrates that even widely scrutinized PDES applications can contain subtle reversibility bugs. Our strategy provides the developer with a simple tool to identify and correct these errors before they manifest in large-scale simulations even years in advance.

## 4 RELATED WORK

Debugging PDES applications presents unique challenges, particularly for optimistic simulations where reverse computation plays a critical role. Existing approaches can be roughly categorized into three main strategies: trace-based debugging, checking determinism of simulation, and checkpoint-replay techniques.

Trace-based approaches like Lanese et al. and Li et al. attempt to capture program behavior by automatically injecting logging statements for variables of interest (**?**; **?**). Lanese et al. developed a debugging framework which inserts fprintf statements to track state changes when handling events. An additional tool was developed to go through the history of changes of each variable of interest and review their individual changes. Similarly, Li et al. built a tool for Erlang that logs both variable changes and the order in which messages are processed. With this information they can revisit what happened in the execution after.

While effective for small-scale simulations, these methods face significant limitations when applied to large PDES models. The volume of trace data becomes unmanageable as model size increases.

A different approach is checking for determinism across multiple runs. Schordan et al. developed a highly sensitive PDES model where any changes on event processing order would be detected (**?**). In their model every LP has two matrices, each is altered every single time an event is processed. This is akin to counters for tracking the number of events processed by an LP. At the end of the simulation, a final matrix is computed from all individual LPs. This final matrix serves as a unique signature of the event processing

order (and the number of them), which allows the model developers to precisely check for determinism. The developer runs the same experiment multiple times under the same starting conditions and should obtain on every single run the same signature.

This approach complements our work by providing an additional checking method for determinism. We currently use the total net number of events processed as a proxy for determinism. After all, if an event is incorrectly processed it might alter the number of events it induces later on in the simulation and it will alter the total number of events (assuming no event ties ever occur without the tie-breaking mechanism being active). Thus, to gain additional trust that the order of events stays the same across different simulation runs, the developer can implement the signature, matrix-based model in their computation from Schordan et al. However, this highly sensitive, matrix-based method cannot pinpoint alone the source of non-determinism even if it detects it, because it only verifies the end result rather than the specific events or handlers causing inconsistencies.

Most closely aligned with our work are checkpoint-replay approaches like (**?**), which combine state saving with functions that check the state of the model. By checkpointing the model state at intervals and using binary search to isolate bugs, these methods can identify specific events causing model corruption. The key challenge with this approach lies in implementing effective checks that can detect subtle state inconsistencies (like a single bit flip or a counter that was not correctly reversed). In our work, we check for state inconsistencies by comparing two states element by element. In checkpoint-replay, one is expected to determine if an (isolated) state is correct or incorrect (i.e, we do not compare it against another state), an ostensibly hard task.

Our approach differs from these previous works by focusing specifically on validating reverse handlers in optimistic simulation. We systematically check each event handler/reverse event handler pair during simulation. This provides immediate feedback about state inconsistencies produced by reverse event handlers at their source, significantly reducing the debugging effort required.

## 5 CONCLUSION

In this paper, we presented a simple yet powerful strategy for checking reverse handlers in PDES. By systematically validating state consistency through forward-rollback-forward cycles for every event processed, our approach detects subtle errors that might otherwise remain hidden until the right conditions trigger it, such as significantly scaling a simulation up, which could happen even years in advance. The effectiveness of this approach has been demonstrated on CODES, a mature and complex HPC network simulator, where we uncovered several critical bugs including one that manifested only after processing approximately one billion events and another that produced discrepancies of just two events out of eight billion in 30 minute runs. We even identified a subtle bug in ROSS's tie-breaker mechanism itself, demonstrating that core simulation infrastructure is susceptible to reverse computation errors that can undermine determinism guarantees.

While implementing our strategy required integrating a new execution mode into ROSS and developing custom deep-copy functions for CODES's complex LP structures, this one-time investment significantly reduces the effort typically required to track down non-deterministic behavior. By checking each event individually and operating in sequential mode, our approach identifies errors precisely at their source rather than after propagation. As PDES continues to expand to more domains and modeling scenarios, helping developers to find bugs becomes critical for them to work on more important tasks than debugging.

Future work could extend our checking strategy to optimistic parallel execution in two ways. The first involves performing our check within the optimistic simulation loop on each PE. Just as we injected an additional reverse and a forward event handler call for every event, we can do the same for each loop within each PE. The second alternative would be to process events in batches, reverse them collectively, and then reprocess them. This batch approach might catch additional bugs that only emerge during multiple event interactions, but would make isolating specific problematic events harder.

## ACKNOWLEDGMENTS

This work is supported by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-06CH11357. We thank Claude.ai for providing feedback on the writing of this manuscript, and helping out with the tedious and long task of copying, comparing and printing the state of large C structs.

## REFERENCES

Barnes, P. D., C. D. Carothers, D. R. Jefferson, and J. M. LaPre. 2013. "Warp Speed: Executing Time Warp on 1,966,080 Cores". In *Proceedings of the ACM SIGSIM PADS 2013 Conference*. May 19th-22nd, Montreal, Quebec, Canada, 327-336.

Bryant, R. E. 1977, November. *Simulation of Packet Communication Architecture Computer Systems*. Ph. D. thesis, MIT.

Carothers, C. D., D. Bauer, and S. Pearce. 2002. "ROSS: A High-Performance, Low-Memory, Modular Time Warp System". *Journal of Parallel and Distributed Computing* 62(11):1648–1669.

Chandy, K. M. and J. Misra. 1979. "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs". *IEEE Transactions on Software Engineering* SE-5(5):440–452.

Collier, N. and M. North. 2013. "Parallel Agent-based Simulation with Repast for High Performance Computing". *SIMULATION* 89(10):1215–1235.

Cope, J., N. Liu, S. Lang, P. Carns, C. Carothers and R. Ross. 2011. "CODES: Enabling Co-design of Multilayer Exascale Storage Architectures". In *Proceedings of the Workshop on Emerging Supercomputing Technologies*. May 31st, Tucson, Arizona, USA.

Fujimoto, R. M. 2000. *Parallel and Distributed Simulation Systems*. "John Wiley and Sons, Inc.".

Galano, G., S. Giammusso, and M. Nardelli. 2024. "Modeling Central Bank Digital Currency over Payment Channels: A Parallel ROSS-based Approach". In *Proceedings of the 38th ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. June 24th-26th, Atlanta, Georgia, USA, 30-34.

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

Kim, J., W. J. Dally, S. Scott, and D. Abts. 2008. "Technology-driven, Highly-scalable Dragonfly Topology". *ACM SIGARCH Computer Architecture News* 36(3):77–88.

Lanese, I. and G. Gössler. 2024. "Causal Debugging for Concurrent Systems". In *Proceedings of the 2024 Reversible Computation Conference*. July 4th-5th, Torun, Poland, 3-9.

Li, T., Y. Zhao, S. Bao, and Y. Yao. 2017. "A Debugging Framework for Parallel Discrete Event Simulation Application". In *Modeling, Design and Simulation of Systems*, edited by M. S. M. Ali, H. Wahid, N. A. M. Subha, S. Sahlan, M. A. M. Yunus, and A. R. Wahap, 656–665. Singapore: Springer.

McGlohon, N. and C. D. Carothers. 2021. "Toward Unbiased Deterministic Total Ordering of Parallel Simulations with Simultaneous Events". In *2021 Winter Simulation Conference (WSC)*, Article 5. 1–15 https://doi.org/10.5555/3522802.3522807.

Nicol, D. M. and X. Liu. 1997. "The Dark Side of Risk (What Your Mother Never Told You About Time Warp)". In *Proceedings of the PADS 1997 Conference*. June 10th-13th, Lockenhaus, Austria, 188-195.

Rao, D., N. Thondugulam, R. Radhakrishnan, and P. Wilsey. 1998. "Unsynchronized Parallel Discrete Event Simulation". In *1998 Winter Simulation Conference (WSC)*, 1563–1570 https://doi.org/10.1109/WSC.1998.746030.

Schordan, M., T. Oppelstrup, M. K. Thomsen, and R. Glück. 2020. "Reversible Languages and Incremental State Saving in Optimistic Parallel Discrete Event Simulation". In *Reversible Computation: Extending Horizons of Computing: Selected Results of the COST Action IC1405*, edited by I. Ulidowski, I. Lanese, U. P. Schultz, and C. Ferreira, 187–207. Cham: Springer International Publishing.

Xu, Y., H. Aydt, and M. Lees. 2012. "SEMSim: A Distributed Architecture for Multi-scale Traffic Simulation". In *Proceedings of the 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*. July 15th-19th, Zhangjiajie, China, 178-180.

Yang, X., J. Jenkins, M. Mubarak, R. B. Ross and Z. Lan. 2016. "Watch Out for the Bully! Job Interference Study on Dragonfly Network". In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. November 13th-18th, Salt Lake City, Utah, USA, 750-760.

Zhu, F. and Y. Yao. 2012. "A Bug Locating Method for the Debugging of Parallel Discrete Event Simulation". In *Proceedings of the 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*. July 15th-19th, Zhangjiajie, China, 81-83.

## AUTHOR BIOGRAPHIES

**ELKIN CRUZ-CAMACHO** is a PhD Candidate in the Department of Computer Science at Rensselaer Polytechnic Institute in Troy, New York, USA. His research interests include high-performance computing (HPC), parallel discrete event simulation

(PDES), software verification, programming languages, and cognitive architectures for robotics. His work has focused on accelerating large-scale simulations, achieving up to 70× performance improvements for HPC network simulators. He has also contributed to software verification for aircraft systems, developed static analysis tools for shapes of NumPy array, designed cognitive architectures for embodied robotics, and created GUI-based sequential simulators for chemical kinetics. His email address is cruzce@rpi.edu.

**CHRISTOPHER D. CAROTHERS** is a faculty member in the Computer Science Department at Rensselaer Polytechnic Institute. He received the Ph.D., M.S., and B.S. from Georgia Institute of Technology in 1997, 1996, and 1991, respectively. Prior to joining RPI in 1998, he was a research scientist at the Georgia Institute of Technology. His research interests are focused on massively parallel computing which involve the creation of high-fidelity models of extreme-scale networks and computer systems. These models have executed using nearly 2,000,000 processing cores on the largest leadership class supercomputers in the world. Additionally, Professor Carothers serves as the Director for the Rensselaer Center for Computational Innovations (CCI). CCI is a partnership between Rensselaer and IBM. The center provides computational and storage resources to a diverse network of researchers, faculty, and students from Rensselaer, government laboratories, and companies across a number of science and engineering disciplines. The flagship supercomputer is an 8 peta-flop (PF) IBM AC922 hybrid CPU/GPU supercomputer named "AiMOS" which serves as the primary testbed for the IBM AI Hardware Center. His email address is chrisc@cs.rpi.edu and his website is https://www.cs.rpi.edu/~chrisc/.