TIME- AND SPACE-CONSCIOUS OMNISCIENT DEBUGGING OF PARALLEL DEVS

Yentl Van Tendeloo
Simon Van Mierlo
University of Antwerp, Belgium
{firstname}.{lastname}@uantwerpen.be

Hans Vangheluwe
University of Antwerp, Belgium
Flanders Make, Belgium
McGill University, Montréal, Canada
hv@cs.mcgill.ca

ABSTRACT

Current Parallel DEVS simulation tools provide a wide set of debugging features. Omniscient debugging, or debugging backwards in time, is only rarely implemented, presumably due to its high resource consumption. Outside of DEVS simulation, omniscient debugging implementations are often lossy: some parts of the model or code are ignored, or they consider only a time window of most recent events. In this paper, we consider efficient and lossless omniscient debugging, particularly in the context of Parallel DEVS. We take inspiration from optimistic synchronization protocols, which can roll back simulation. We investigate how this technique can be tailored to omniscient debugging of Parallel DEVS models. Our algorithm limits simulation overhead and memory consumption, while remaining lossless. We compare the traditional approach with our approach. Our approach significantly decreases time and space overhead, at the cost of slightly slower debugging operations.

Keywords: DEVS, debugging, omniscient, memory, performance.

1 INTRODUCTION

Current Parallel DEVS (Chow and Zeigler 1994) simulators provide an extensive set of debugging features (Van Mierlo, Van Tendeloo, and Vangheluwe 2016, Van Tendeloo and Vangheluwe 2017), inspired by traditional programming language debugging (Zeller 2005). Stepping through the simulation trace is such an operation, which offers modellers insight in intermediate simulation states. The inverse operation, stepping back in simulated time, has recently received much attention in the programming language domain (*e.g.*, in GDB (GDB 2009)). Omniscient debugging overcomes problems commonly associated with breakpoint-based debugging (Lewis 2003, Pothier and Tanter 2009). The most prominent being that erroneous behaviour probably occured before the breakpoint was triggered, for which most information is lost already. Despite these advantages, no omniscient debugger exists for Parallel DEVS.

In previous work (Van Mierlo, Van Tendeloo, and Vangheluwe 2016), we added stepping back in time to our Parallel DEVS debugger. Stepping back in time, often termed omniscient debugging, aids modellers to gain insight in their model. With omniscient debugging, modellers can jump back to arbitrary points in past simulated time. Despite the many advantages of omniscient debugging, implementing it efficiently is challenging. Not only is the implementation far from trivial, there are significant performance implications as well. We analyze the performance of our existing omniscient Parallel DEVS debugger, revealing problems related to overhead in both computation and memory. And while omniscient debugging was functionally complete, the low performance made it unusable for anything but toy models. This is not uncommon: slowdowns of a factor 100 or more are noted in the literature (Lienhard, Gîrba, and Nierstrasz 2008).

Another domain where stepping back in time is required, is optimistic synchronization, for example in Time Warp (Jefferson 1985). We apply existing optimistic synchronization approaches, with minor modifications, in the context of omniscient debugging. Minor modifications are required, as omniscient debugging has different priorities: rollbacks can take some time and a complete history of the model must remain available. Contrary to other omniscient debugging approaches, this algorithm is lossless and has a low overhead in both time and space.

The remainder of this paper is organized as follows. Section 2 introduces omniscient debugging operations and how they are implemented in our previous versions of the Parallel DEVS debugger. Section 3 presents our algorithm and discusses time and space consumption. Section 4 presents our evaluation based on several criteria. Section 5 presents related work in both model and code debugging. Section 6 concludes the paper.

2 OMNISCIENT DEBUGGING

In contrast to the usual debugging operations, omniscient debugging goes back in time. This grants more freedom to modellers, as they can now traverse the simulation trace in two directions. As users are discovering the benefits, omniscient debugging is gaining popularity. Two ways of stepping back in time exist: taking a single simulation step back, or jumping to an arbitrary point in simulated time.

2.1 Parallel DEVS Debugger

In our previous work (Van Mierlo, Van Tendeloo, and Vangheluwe 2016), we presented a Parallel DEVS debugger with an unparallelled set of debugging features, based on the PythonPDEVS simulation kernel (Van Tendeloo and Vangheluwe 2014). Timing related features included small step, big step, backwards step, realtime simulation, and as-fast-as-possible simulation. Other features, less relevant to this paper, include god events, breakpoints, termination conditions, and event injection.

But whereas many other Parallel DEVS simulators support these basic features, omniscient debugging is one of our unique debugging features. In this paper, we focus on optimizing the implementation, to lower the threshold for the modeller to use it. The old implementation was based on *Full State Saving*: the complete model state is stored after each transition. This is done through serialization of the state, and subsequent deserialization when restoring the state.

A small, but significant, optimization was made: only the state of atomic models that execute a transition is stored. We call this slightly optimized algorithm *Copy State Saving*. This algorithm is used already in our earlier versions of the omniscient debugger. It doesn't store a single consistent state, but only timestamped partial states. These partial states can be combined into a single consistent global state. This decreases both time and memory consumption, as unchanged states are not serialized and stored again.

2.2 Problems with Omniscient Debugging

Despite omniscient debugging's advantages, there are severe performance limitations. First, omniscient debugging is plagued with memory issues (Bousse, Corley, Combemale, Gray, and Baudry 2015, Pothier, Tanter, and Piquer 2007, Corley, Eddy, Syriani, and Gray 2017). Storing the complete simulation trace eventually leads to memory exhaustion, as trace size only increases. When simulators run out of memory, simulation halts. This makes it impossible to simulate large-scale models using omniscient debugging. Most omniscient debuggers tackle this problem in a lossy way. Either they use a time window, where only the most recent states are retained, or they only store part of the state. Both approaches are lossy: it is impossible to go beyond the fixed time window, or to access untracked parts of the model state.

Second, omniscient debuggers have low simulation performance (Pothier and Tanter 2009, Lienhard, Gîrba, and Nierstrasz 2008). The primary overhead is in serializing and storing model states after each transition. Contrary to memory consumption, high time overhead does not prevent a model from being simulated. Nonetheless, it might become impossible to simulate the model within reasonable time bounds.

2.3 Performance Evaluation

While the memory problem is obvious given the stored amount of data, the time problem is less obvious. For our Parallel DEVS debugger, Figure 1 shows the difference between turning omniscient debugging on and off, dependent on the size of the state. We see that execution time increases as the state history increases, due to the serialization overhead of state saving becoming the bottleneck. This increase is linear, as the serialization routine used has linear complexity in terms of the state size.

This overhead is always present when the option for omniscient debugging is provided, even when it is never actually used. The sporadic use of omniscient debugging, therefore, does not warant the significant overhead on the more frequent forward simulation operations. The other option would be for users to select upfront whether they want to use omniscient debugging or not. This is not always known upfront, and even when users know so, simulation becomes unusably slow when enabled.

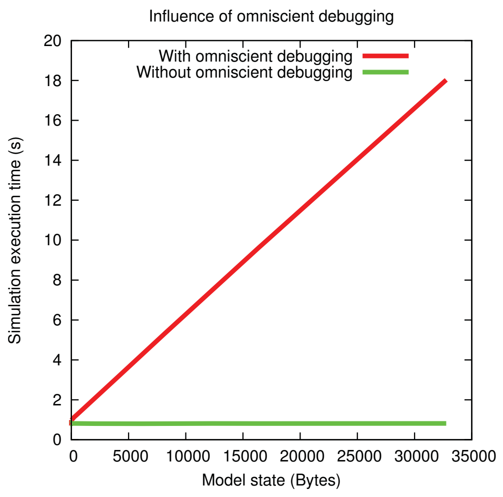

Figure 1: Overhead of omniscient debugging in forward simulation.

3 OPTIMIZATION

We now optimize the state saving algorithm described previously. Before we do so, we go back to the core problem: “how to jump back to an arbitrary point in history?”. This is the same problem encountered in Time Warp, which we will now further investigate.

3.1 Relation to Time Warp

In parallel simulation, synchronization protocols are frequently used to allow different simulation nodes to be at different points in simulated time (Fujimoto 1990, Fujimoto 1999), increasing parallelism. Time Warp (Jefferson 1985) is one such optimistic synchronization protocol. In Time Warp, each computational node simulates as fast as possible, ignoring the possibility for external events. If such an event occurs anyway, simulation is rolled back to a point right before the event was supposed to be processed. The event is then processed as usual, circumventing the causality problem. Rollbacks significantly resemble our core problem: a way to jump to arbitrary points in the past simulated time. We base our omniscient debugging algorithm on those defined for optimistic synchronization.

In the context of Time Warp, several algorithms were created (Cleary, Gomes, Unger, Xiao, and Thudt 1994, Rönngren and Ayani 1994, Preiss, Loucks, and Macintyre 1994), with varying degrees of stored data. *Full State Saving* stores a complete model snapshot at each transition, as discussed before. *Copy State Saving* stores a snapshot of a specific model that changes its state, as discussed before. *Incremental State Saving* stores only the difference between two subsequent states, in the form of a reverse operation. During a rollback, all state changes need to be undone in reverse order. This makes the length of the rollback influence the time taken for the rollback. *Periodic State Saving* will, instead of storing the model state after every transition, only store the state periodically. During a rollback, we select the closest state before the requested time, and simulate from then on. This assumes determinism in the simulation algorithm, as otherwise it is not guaranteed that the same choices are made.

There are different non-functional requirements between Time Warp and omniscient debugging. First, Time Warp solves the memory problem by using a window-based approach. Contrary to omniscient debugging, however, optimistic synchronization can place a lower bound on the states that will be accessed, using the Global Virtual Time (GVT), allowing it to use a window. This is not the case with omniscient debugging, as we cannot know what state the user wants to go back to. Second, rollbacks occur often in Time Warp, and need to be processed fast to prevent cascading rollbacks (Fujimoto 1999). This is again not the case with omniscient debugging, where backwards steps happen only rarely and performance is less of an issue.

For Time Warp, the main disadvantage of periodic state saving is that it requires forward simulation for each backward step. This makes a backward step take longer than a forward step (as one includes the other). But although this is a substantial problem for Time Warp, omniscient debugging is used interactively and only rarely. So wheras a latency of 0.1 seconds is too much for Time Warp, even latencies up to half a second might be tolerable during omniscient debugging. So since periodic state saving's disadvantages are minimal for omniscient debugging, we use this algorithm for our implementation of omniscient debugging.

3.2 Periodic State Saving for Omniscient Debugging

Our algorithm is based on Periodic State Saving: instead of storing the state of models at transition-time (as in copy state saving), we store the full simulation state after a fixed interval. This does not influence the forward simulation algorithm at all, as storing the model state happens independent of forward simulation. For backward steps, we search the most recently saved simulation state, revert to it, and forward simulate from there up to the requested time. Users can configure the interval, thus influencing performance.

The checkpointing interval is defined in wall-clock time (i.e., real world time), instead of simulation time (i.e., internal clock of the simulator). While simulation time provides deterministic points in the simulation where snapshots are made, using the wall-clock time takes into account a possibly changing simulation pace. Time efficiency, and latency, is expressed in wall-clock time, as that is the actual time that the modeller will have to wait for operations. Defining the interval in number of events executed would also be possible, but has similar disadvantages as basing it on simulation time.

We also allow users to configure the maximally allowed memory use. When simulation uses more memory, the oldest full model snapshots are compressed and persisted to disk. Since these old snapshots are very unlikely to be necessary, and responsive performance is all that we require, there is no significant disadvantage to disk storage. This way, the full disk space becomes available for use by omniscient debugging, without any noticable performance impact.

We consider this approach and its configurability in two dimensions: time and memory.

3.2.1 Time

Before we talk about optimizing time, we reason which operations to optimize: making one operation faster potentially makes other operations slower. In general, forward simulation steps are far more common than backward simulation steps. Forward steps happen at a high frequency, as they are automatically invoked. Backwards steps happen only rarely and are invoked interactively by the user. Whereas thousands of forward steps can be requested in a single second, backward steps occur only rarely: the modeller must analyze the model to decide whether to step back again or not, and must press a button to invoke the next step. Optimizing forward simulation should thus be our priority, on the condition that backwards steps remain responsive enough for interactive use.

It is clear from our approach that we prioritize forward simulation: the forward simulation algorithm is unaware of any omniscient debugging, and thus experiences no overhead. Sometimes, minor pauses are noted in the simulation, which are used to serialize the complete state. When rolling back, the latest snapshot is selected and simulation is restarted from there on. This is shown in Figure 2, where only three snapshots are made. Any rollback will be changed to a reset of a snapshotted state, and forward simulation continues from there on. For example, when rolling back to time 6, state $a4$ is missing, making us roll back to time 4, where a snapshot was previously made. From here, the transition function resulting in $a4$ is executed again, to yield the total state at time 6, as requested.

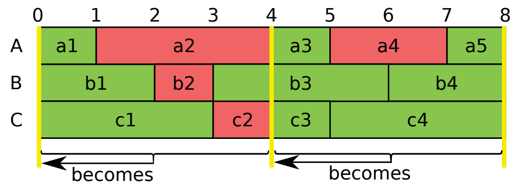

Figure 2: Overview of periodic state saving approach. Green states (light) are stored, red (dark) states are not. Yellow lines indicate a point at which a snapshot is made.

Backwards steps, however, are clearly penalized, as they now require a *forward simulation phase*, in which the latest consistent snapshot must be simulated up to the requested point in time. The time taken for the forward simulation phase is bounded by the snapshot interval: with snapshots every x seconds (of forward simulation), a rollback never requires more than x seconds of forward simulation to reach the desired state, as otherwise another snapshot would have been closer.

The user is free to configure the interval between consecutive snapshots. Setting a *longer interval* between two snapshots results in: (1) *lower memory consumption*, since snapshots are saved less frequently; (2) *faster forward simulation*, since less serialization pauses occur; (3) *slower backward simulation*, since less states are saved, requiring more forward simulation to reach the requested rollback time.

3.2.2 Memory

We consider two types of memory: main memory and disk. While main memory is fast and easy to use, it is relatively small compared to the hard disk. Main memory size is therefore a problem, as all objects go to main memory by default.

Our approach tackles this by writing old state snapshots to disk, freeing up main memory. It is unlikely for old snapshots to be accessed, and since the additional delay is only in the order of ten milliseconds, we can easily store old states to disk. Only a single snapshot is loaded from disk, as all snapshots are self-contained. Recent snapshots, having a higher chance of being accessed, stay in main memory. As we provide a lossless approach, we must keep these older states available for when a jump to them is requested, however small the chance of accessing them is.

Writing data to hard disk is possible for both copy state saving and periodic state saving. For copy state saving, the state of each model is managed individually, thus requiring as many disk accesses as there are atomic models. For periodic state saving, the full state at some point in time is managed as a single block of data, requiring a single disk access.

Although reading and writing data to disk does not seem very attractive from a performance view, performance is good due to asynchronous I/O. During forward simulation, where performance really matters, we only do sporadic writes to main memory. We only write do disk when main memory usage passes the defined threshold. These writes happen asynchronously and can therefore be considered (almost) instantaneous, as it will be buffered by the Operating System (OS). Reading data is the more expensive operation, but that only happens once per backward step. Including access and transfer times, reading data still feels interactive, as it only adds milliseconds to the total time of a rollback. The cost of the forward simulation phase is many times higher.

3.3 Long-running Simulations

Even now, our approach cannot handle arbitrarily long running simulations: just like main memory, disk space eventually runs out. Despite optimizations to increase the capacity of our storage media, such as file compression, this only delays the point where memory inevitably runs out. We believe that there are two directions to solve this problem.

The first possibility is to further extend storage media through existing technologies, such as adding more disks or storing it in the cloud. Whereas this technology exists and is sufficiently mature, this again merely delays the point where memory runs out: neither of these approaches has an infinite capacity, though it can be increased on-demand.

The second possibility truly tackles infinitely running simulations, at the cost of increased latency for omniscient debugging operations. By pruning away intermediate snapshots persisted to disk, we gain more storage space for future snapshots. This comes at a cost, since each snapshot was there to guarantee the initially defined latency. Whereas our approach still works even with less snapshots, latency increases (but remains bounded). For example, when removing every other snapshot, average latency doubles, though memory consumption halves. This can keep going on, though latency doubles each time. Nonetheless, we can bound the time it takes to reach the requested state. This differs from a window-based approach in that we remain lossless. Our approach is also guaranteed to never be slower than restarting the simulation completely, as a restart is just the worst case situation, in which there is no closer snapshot available.

4 PERFORMANCE EVALUATION

We now evaluate this algorithm in our Parallel DEVS debugger (Van Mierlo, Van Tendeloo, Barroca, Mustafiz, and Vangheluwe 2015, Van Mierlo, Van Tendeloo, and Vangheluwe 2016). As Time Warp was already implemented in PythonPDEVS (Van Tendeloo and Vangheluwe 2015), these algorithms could be mostly reused for omniscient debugging. We evaluate several kinds of resources, operations, and models. For resources, we measure CPU time needed for operations, main memory consumption, and disk space used. For operations, we consider the time needed for a forward step, a backward step, and a random jump to the past. For models, we define a minimal, synthetic benchmark model with a configurable state size. As the model is synthetic, we have disabled compression for these benchmarks, as it would skew the results: the data would either be trivially compressable (*e.g.*, all zero values), or not compressable at all (*e.g.*, full random).

First, we focus on the initial goal of this paper: minimizing time and space overhead of omniscient debugging. Second, we discuss the trade-off made for this: omniscient debugging operations become slower. Finally, we vary the size of the benchmark model to measure the influence on performance.

For all dimensions, our benchmark model is the same simple Coupled DEVS model with a configurable number of atomic models. Each of these atomic models has a configurable state size. The atomic models are configured to do an internal transition after a (uniformly distributed) time advance has passed. The structure of the coupled model (i.e., how these atomic models are coupled) is irrelevant to this paper and is therefore not discussed further. Our implementation, as well as our benchmark models and configuration, are available online (https://msdl.uantwerpen.be/git/yentl/PDEVSOmniscientDebugger).

4.1 Omniscient Debugging Overhead

The first advantage of our approach is decreased simulation overhead for forward simulation. In our problem statement, Figure 1 indicated the performance impact of omniscient debugging. Such overhead is unacceptable for complex simulations. Certainly since it is always imposed, even if no omniscient debugging features are actually used when debugging.

Our solution is periodic state saving, which significantly influences forward simulation speed, as shown in Figure 3. Figure 3 is an updated version of Figure 1, now including results for periodic state saving with two different configurations. Depending on the desired responsiveness of omniscient debugging operations, a latency of 0.5 seconds might be tolerable. In that case, forward simulation has barely any overhead, while a backwards step takes up to 0.5 seconds. Even with a latency of only 0.1 seconds forward simulation significantly outperforms copy state saving.

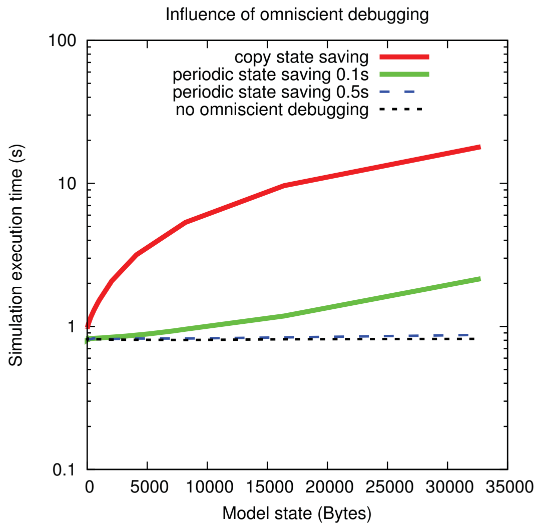

Figure 3: Overhead of omniscient debugging in function of state size (logaritmic scale).

4.2 Memory and Disk Consumption

The prime concern with omniscient debugging is memory consumption, as the full state trace must remain accessible. But whereas copy state saving needs many different states to create a consistent snapshot, this is not needed with periodic state saving. Only one snapshot is stored every so often, which is guaranteed to be consistent. This drastically decreases memory consumption.

Results for copy state saving are shown in Figure 4, plotting the evolution of main memory and disk consumption. Main memory consumption increases linearly as new data is saved at each simulation step, and the cumulative number of simulation steps increases linearly due to the uniform time between transitions. After some time, data is moved from main memory to disk in big chunks. These points in time are clearly identifiable in the figure as the points where main memory consumption decreases and disk consumption increases. Note that, at simulation time 1000, approximately 70 megabytes of disk space is used to store all intermediate states. While this does not seem much, these figures are obtained after less than a minute of simulation on a simple model. For long running simulations with realistic models, disk space quickly runs out. Additionally, writing this much data to disk quickly becomes too much to be buffered by the OS.

Figure 5 shows results for exactly the same model, but with periodic state saving. Our first observation is that main memory remains constant most of the time, only increasing at points where a full snapshot is taken. Since a single copy of the simulation state is already too large according to our defined space constraint, the data is immediately swapped to disk afterwards. Another interesting observation is the total memory: a mere 800 kilobytes. Only four copies of the state are stored, being much less memory intensive than saving

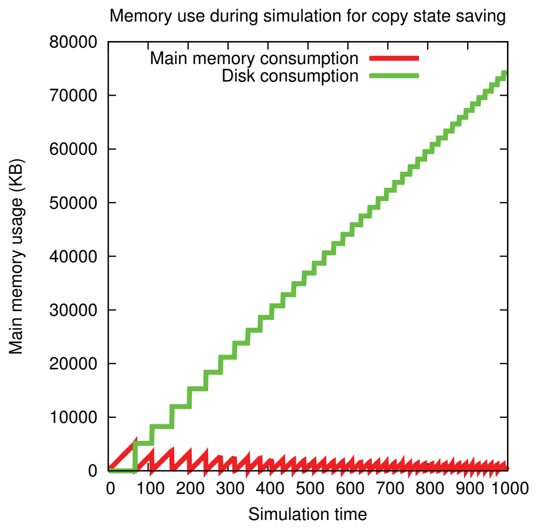

Figure 4: Memory use of copy state saving.

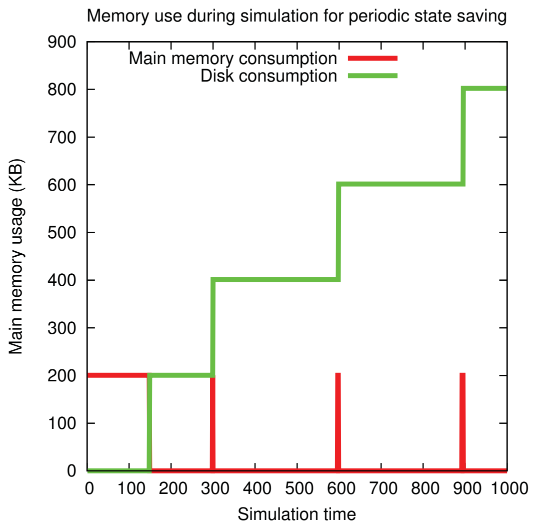

Figure 5: Memory use of periodic state saving.

all intermediate states, even incrementally. As the time between two transitions is uniformly distributed, there is some (emergent) synchronization between the wall-clock time and simulation time, resulting in equidistantly spaced snapshots.

4.3 Jump Latency

Periodic state saving excels in memory consumption and simulation overhead. The drawback is slower backward steps: some forward simulation is necessary to get to the desired state.

The latency for jumps backward in time is shown in Figure 6. For this benchmark, the model is simulated up to simulation time 100. From that point, we measure how long it takes to jump to a specific point in its state history. This point is seen on the *x* axis. Copy state saving has a near-constant delay, no matter to which point in simulated time is being jumped. This is as expected, as each state is stored in memory and can just be retrieved. Cost of retrieval from memory, and even from disk, is negligible compared to the cost of forward simulation.

For periodic state saving, results are far worse: most points in simulated time require computation time for the forward simulation phase. This is as expected: only some full states are stored in memory, and these will be the points to which a rollback occurs. From our results, these points seem to be somewhere around simulation time 0, 39, 75, and 89. Executing a rollback could, in the worst case, be to a point in time right before a snapshot is made (*e.g.*, time 38). We can also deduce from the results that our interval between two snapshots was 0.5 seconds, as a jump never takes longer than 0.5 seconds. Note the snapshot at time 89, which should not have occured until the latency reaches almost 0.5s. This is probably caused by other functionality of the simulator (such as compressing and writing snapshots to persistent storage), which is taken into account in the interval, but doesn't contribute to simulation progression.

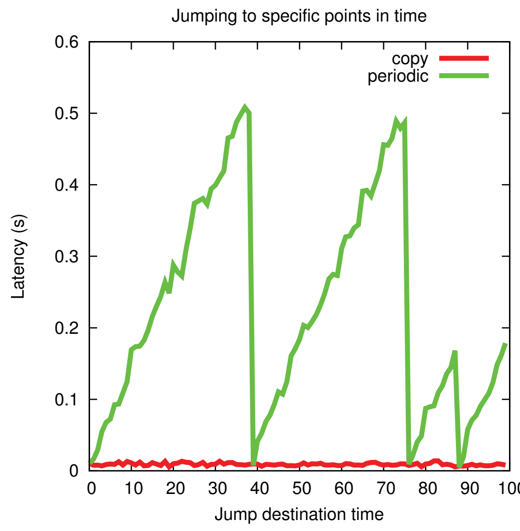

Figure 6: Jump latency for copy and periodic state saving.

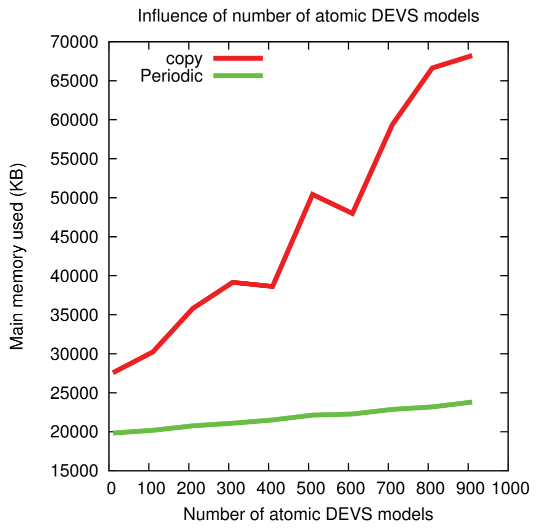

Figure 7: Influence of number of atomic models on memory consumption.

4.4 Influence of Model Size

Finally, we analyze the influence of the total size of the model on both state saving options, in terms of memory consumption. There are two dimensions influencing the size of the model: the number of atomic DEVS models, and the size of each model's state. Model state size was previously studied in Figure 3.

Figure 7 presents result for a varying number of atomic DEVS models. Increasing the number of atomic DEVS models barely influences periodic state saving, as snapshots are only taken infrequently. Copy state saving is significantly impacted, as it stores, for *n* models, *n* history queues containing all previous states of the model. Increasing the number of atomic DEVS models also increases the number of executed transitions, requiring even more serialization and storage.

5 RELATED WORK

Omniscient debugging was first explored in the context of General-Purpose Languages (GPLs). The main difference is their non-determinism: user input and I/O events have to be stored in order to correctly step back in time. Pothier *et al.* use events to monitor the running execution (Pothier, Tanter, and Piquer 2007). These events are stored in a database, which can be distributed to further increase performance. Boothe explores techniques for efficient bidirectional debugging of GPL programs (Boothe 2000). These techniques are based on event traces (to deal with non-determinism) and snapshotting (for increased performance). Older snapshots are progressively removed as the program is executed (for memory efficiency). Ultimately, however, memory runs out, as removing events from the trace is not an option. The only solution is to become lossy: dropping events from the trace, or limiting the number of backward steps the modeller can make (*i.e.*, a window). If it turns out that the user is interested in these events after all, the program needs to be reset and executed again. Some approaches, such as reverse computation (Zelkowitz 1973), partly avoid the problem of memory consumption. But while they avoid one problem, reverse computation is computationally more intensive for long jumps, and not even always possible. Engblom presents an overview of different techniques for omniscient debugging of GPLs (Engblom 2012). Recently, support for omniscient debugging has also been included in mainstream tools, such as GDB.

Whereas it is possible to use a general purpose debugger on the generated code of our Parallel DEVS simulator, there is a mismatch in abstraction levels. Modellers, for example Parallel DEVS modellers, are unable to see the relation between the failing code and their model. Several tools support debugging operations for Parallel DEVS. This is covered in our previous paper, where we introduce an advanced, visual, interactive Parallel DEVS debugger (Van Mierlo, Van Tendeloo, and Vangheluwe 2016), which serves as the basis for our work in this paper. Our prototype was compared to adevs (Nutaro 2013), DEVS-Suite (Kim, Sarjoughian, and Elamvazhuthi 2009), MS4 Me (Seo, Zeigler, Coop, and Kim 2013), VLE (Quesnel, Duboz, Ramat, and Traoré 2007), and X-S-Y (Hwang 2012). A more in-depth comparison (including performance comparison) was made in (Van Tendeloo and Vangheluwe 2017), which also included PowerDEVS (Bergero and Kofman 2011). All tools support debugging at some level, but none support omniscient debugging, or "backwards clock".

Omniscient debugging techniques have been explored in the context of modelling languages. Corley *et al.* have implemented omniscient debugging for model transformations and analysed its efficiency (Corley, Eddy, and Gray 2014, Corley, Eddy, Syriani, and Gray 2017). Since model transformations are non-deterministic, their implementation logs each change at the end of a transformation step. By inverting these changes, users step back to previous states. Overhead is limited, as it is incremental in nature, though it eventually runs out of memory unless old events are dropped or persisted to disk. Neither of these handles long running simulations losslessly: disk space can still run out. Time can also present a minor problem during a rollback, as it is linear in the length of the rollback: history unrolls step by step.

In contrast to model transformations, Parallel DEVS is a deterministic formalism, meaning that we can remove arbitrary intermediate states: they can always be computed again. In (Bousse, Corley, Combemale, Gray, and Baudry 2015), the authors explore the debugging of domain-specific languages. Their approach is based on the saving of a trace during execution, which can be explored backwards and forwards by the modeller. They allow for domain-specific languages to specialize the generic trace algorithm, to gain efficiency in space and time.

The literature on omniscient model debugging is rather sparse and does not include many lossless optimizations or solutions for the memory management problem. Existing approaches mostly focus on code debugging, or rely on lossy techniques. And while we agree that lossy techniques are sometimes necessary (*i.e.*, for non-deterministic formalisms), we can make additional optimizations in the case of Parallel DEVS. The literature on optimistic synchronization protocols (Fujimoto 1990, Fujimoto 1999), however, has extensive work on optimizing rollbacks in a deterministic and lossless way. Many variations to, and evaluations of, state saving algorithm exist (Cleary, Gomes, Unger, Xiao, and Thudt 1994, Rönngren and Ayani 1994, Preiss, Loucks, and Macintyre 1994). We have based ourselves on these algorithms to define a lossless, time- and space-conscious omniscient debugging algorithm.

6 CONCLUSION

Despite the increasing popularity of omniscient debugging in the programming language domain, other domains are reluctant to incorporate it. For Parallel DEVS, PythonPDEVS is, to the best of our knowledge, the only simulation tool to implement it. Although fully implemented, performance was a major consideration, even if omniscient debugging is only used sparingly. Performance considerations exist in terms of memory and time efficiency. With naive algorithms, even small models become difficult to simulate due to its resource consumption. We therefore set out to find algorithms to cut down the overhead in terms of forward simulation performance and memory consumption.

We presented periodic state saving, combined with swapping to disk, as a lossless technique to restore state histories in a tolerable time. Forward simulation overhead was significantly reduced at the cost of slower omniscient debugging operations. Omniscient debugging operations do become slower, though they remain responsive for interactive use. Our synthetic benchmarks validated our expectations.

In future work, more advanced techniques for the swapping and pruning of snapshots on disk can be considered. Advanced optimistic synchronization algorithms might not be applicable as-is, due to the differing requirements. For example, recurring states could be automatically detected (*e.g.*, through the use of hashes) and linked together. Other future operations include reverse breakpointing.

ACKNOWLEDGMENTS

This work was partly funded by PhD fellowships from the Research Foundation - Flanders (FWO) and Agency for Innovation by Science and Technology in Flanders (IWT). This research was partially supported by Flanders Make vzw, the strategic research centre for the manufacturing industry.

REFERENCES

Bergero, F., and E. Kofman. 2011. "PowerDEVS: a tool for hybrid system modeling and real-time simulation". *SIMULATION* vol. 87, pp. 113–132.

Boothe, B. 2000. "Efficient Algorithms for Bidirectional Debugging". In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pp. 299–310.

Bousse, E., J. Corley, B. Combemale, J. Gray, and B. Baudry. 2015. "Supporting Efficient and Advanced Omniscient Debugging for xDSMLs". In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2015, pp. 137–148.

Chow, A. C. H., and B. P. Zeigler. 1994. "Parallel DEVS: a parallel, hierarchical, modular, modeling formalism". In *Proceedings of the 26th Winter Simulation Conference*, pp. 716–722.

Cleary, J., F. Gomes, B. Unger, Z. Xiao, and R. Thudt. 1994. "Cost of State Saving & Rollback". *SIGSIM Simululation Digest* vol. 24 (1), pp. 94–101.

Corley, J., B. P. Eddy, and J. Gray. 2014. "Towards Efficient and Scalabale Omniscient Debugging for Model Transformations". In *Proceedings of the 14th Workshop on Domain-Specific Modeling*, DSM '14, pp. 13–18, ACM.

Corley, J., B. P. Eddy, E. Syriani, and J. Gray. 2017. "Efficient and scalable omniscient debugging for model transformations". *Software Quality Journal* vol. 25, pp. 7–48.

Engblom, J. 2012. "A review of reverse debugging". In *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*, pp. 1–6.

Fujimoto, R. M. 1990. "Parallel discrete event simulation". *Communications of the ACM*, pp. 30–53.

Fujimoto, R. M. 1999. *Parallel and Distribution Simulation Systems*. 1st ed. John Wiley & Sons, Inc.

GDB 2009. "GDB Reversible Debugging". https://www.gnu.org/software/gdb/news/reversible.html.

Moon Ho Hwang 2012. "X-S-Y". https://code.google.com/p/x-s-y/.

Jefferson, D. R. 1985. "Virtual time". *ACM Trans. Program. Lang. Syst.* vol. 7 (3), pp. 404–425.

Kim, S., H. S. Sarjoughian, and V. Elamvazhuthi. 2009. "DEVS-Suite: a simulator supporting visual experimentation design and behavior monitoring". In *Proceedings of the Spring Simulation Conference*.

Lewis, B. 2003. "Debugging Backwards in Time". *arXiv preprint cs/0310016* (September), pp. 225–235.

Lienhard, A., T. Gîrba, and O. Nierstrasz. 2008. "Practical Object-Oriented Back-in-Time Debugging". In *ECOOP 2008 – Object-Oriented Programming*, pp. 592–615.

Nutaro, James J. 2013. "adevs". http://www.ornl.gov/~1qn/adevs/.

Pothier, G., and E. Tanter. 2009. “Back to the Future: Omniscient Debugging”. *IEEE Software* vol. 26 (6), pp. 78–85.

Pothier, G., E. Tanter, and J. Piquer. 2007. “Scalable Omniscient Debugging”. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA ’07, pp. 535–552, ACM.

Preiss, B. R., W. M. Loucks, and I. D. Macintyre. 1994. “Effects of the Checkpoint Interval on Time and Space in Time Warp”. *ACM Trans. Model. Comput. Simul.* vol. 4 (3), pp. 223–253.

Quesnel, G., R. Duboz, E. Ramat, and M. K. Traoré. 2007. “VLE: a multimodeling and simulation environment”. In *Proceedings of the 2007 summer computer simulation conference*, pp. 367–374.

Rönngren, R., and R. Ayani. 1994. “Adaptive Checkpointing in Time Warp”. *SIGSIM Simul. Dig.* vol. 24 (1), pp. 110–117.

Seo, C., B. P. Zeigler, R. Coop, and D. Kim. 2013. “DEVS modeling and simulation methodology with MS4Me software”. In *Symposium on Theory of Modeling and Simulation - DEVS (TMS/DEVS)*.

Van Mierlo, S., Y. Van Tendeloo, B. Barroca, S. Mustafiz, and H. Vangheluwe. 2015. “Explicit Modelling of a Parallel DEVS Experimentation Environment”. In *Proceedings of the 2015 Spring Simulation Multiconference*, SpringSim ’15, pp. 860–867, Society for Computer Simulation International.

Van Mierlo, S., Y. Van Tendeloo, and H. Vangheluwe. 2016. “Debugging Parallel DEVS”. *SIMULATION*.

Van Tendeloo, Y., and H. Vangheluwe. 2014. “The Modular Architecture of the Python(P)DEVS Simulation Kernel”. In *Proceedings of the 2014 Symposium on Theory of Modeling and Simulation - DEVS*, pp. 387–392.

Van Tendeloo, Y., and H. Vangheluwe. 2015. “PythonPDEVS: a distributed Parallel DEVS simulator”. In *Proceedings of the 2015 Spring Simulation Multiconference*, SpringSim ’15, pp. 844–851, Society for Computer Simulation International.

Van Tendeloo, Y., and H. Vangheluwe. 2017. “An evaluation of DEVS simulation tools”. *SIMULATION* vol. 93 (2), pp. 103–121.

Zelkowitz, M. V. 1973. “Reversible Execution”. *Commun. ACM* vol. 16 (9), pp. 566–566.

Zeller, A. 2005. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc.

AUTHOR BIOGRAPHIES

YENTL VAN TENDELOO is a PhD student in the department of Mathematics and Computer Science at the University of Antwerp (Belgium). He is a member of the Modelling, Simulation and Design (MSDL) research lab. In his Master’s thesis, he worked on MDSL’s PythonPDEVS simulator, a simulator for Classic DEVS, Parallel DEVS, and Dynamic Structure DEVS, grafted on the Python programming language. The topic of his PhD is the conceptualization, development, and distributed realization of a new (meta-)modelling framework and model management system called the Modelverse.

SIMON VAN MIERLO is a PhD student at the University of Antwerp, Department of Mathematics and Computer Science, Antwerp, Belgium. He is a member of the Modelling, Simulation and Design (MSDL) research lab. The topic of his PhD is studying how modelling formalisms, environments, and simulators can be enhanced with debugging support.

HANS VANGHELUWE is a Professor in the department of Mathematics and Computer Science at the University of Antwerp (Belgium), an Adjunct Professor in the School of Computer Science at McGill University (Canada) and an Adjunct Professor at the National University of Defense Technology (NUDT) in Changsha, China. He heads the Modelling, Simulation and Design (MSDL) research lab. He has a long-standing interest in the DEVS formalism and is a contributer to the DEVS community of fundamental and technical research results.