Performance analysis of a PDEVS simulator supporting multiple synchronization protocols

Ben Cardoen† **Stijn Manhaeve**† **Tim Tuijn**†
{firstname.lastname}@student.uantwerpen.be

Yentl Van Tendeloo† **Kurt Vanmechelen**†
Hans Vangheluwe†‡ **Jan Broeckhove**†
{firstname.lastname}@uantwerpen.be

† University of Antwerp, Belgium
‡ McGill University, Canada

ABSTRACT
With the ever increasing complexity of simulation models, parallel simulation becomes necessary to perform the simulation within reasonable time bounds. The built-in parallelism of Parallel DEVS is often insufficient to tackle this problem on its own. Several synchronization protocols have been proposed, each with their distinct advantages and disadvantages. Due to the significantly different implementation of both protocols, most Parallel DEVS simulation tools are limited to only one such protocol. In this paper, we present a Parallel DEVS simulator, grafted on C++11, but based on PythonPDEVS, which supports both conservative and optimistic synchronization. We evaluate the performance gain obtained by choosing the most appropriate synchronization protocol. Performance results are compared to adevs, in terms of CPU time and memory usage.

Author Keywords
Simulation; C++11; Optimistic Synchronization; Conservative Synchronization; Performance; Parallel DEVS

ACM Classification Keywords
I.6.7 SIMULATION AND MODELING: Simulation support systems; I.6.8 SIMULATION AND MODELING: Types of simulation

1. INTRODUCTION

DEVS [25] is a popular formalism for modelling complex dynamic systems using a discrete-event abstraction. In fact, it can serve as a simulation "assembly language" to which models in other formalisms can be mapped [23]. A number of tools have been constructed by academia and industry that allow the modelling and simulation of DEVS models.

But with the ever increasing complexity of simulation models, parallel simulation becomes necessary to perform the simulation within reasonable time bounds. And while Parallel DEVS [5] was introduced to increase parallelism, this is often insufficient. Several synchronization protocols from the discrete event simulation community [9] have been applied to DEVS simulation. While several parallel DEVS simulation kernels exist, they are often limited to a single synchronization protocol. The exact reason for different synchronization protocols, however, is that their distinct nature makes them applicable in different situations, each outperforming the other in specific models. The applicability of parallel simulation capabilities of current tools, is therefore limited to the domain of simulation models.

Users that wish to simulate a wide variety of models, with different ideal synchronization protocols, are therefore out of luck: either they accept the lower performance for some of the models, or they use two different simulation kernels. Neither is acceptable for the simulation of complex models: performance can decrease to unacceptable levels, and simulation kernel APIs can diverge significantly.

In this paper, we introduce DEVS-Ex-Machina ("dxex"), our simulation tool which offers multiple synchronization protocols: no synchronization (sequential execution), conservative synchronization, or optimistic synchronization. The selected synchronization protocol is transparant to the simulated model. Users should merely determine, at the start of simulation, which protocol they wish to use. Our tool is implemented in C++11, to increase both performance and portability across different platforms.

We implemented a model that, depending on a single parameter, changes its ideal synchronization protocols. Dxex, then, is used to compare simulation using exactly the same tool, but with a different synchronization protocol. As dxes is able to simulate it both ways, users can always opt to use the fastest protocol available. To verify that our flexibility does not counter performance, we compare to adevs, currently one of the fastest DEVS simulation tools available [21, 7].

Dxex, as well as all of our benchmark models, results, and profiling results, can be found online at `https://bitbucket.org/bcardoen/devs-ex-machina`.

The remainder of this paper is organized as follows: Section 2 introduces the necessary background on synchronization protocols and our implementation language. Section 3 elaborates on some of our features, and our design that enables our flexibility. In Section 4, we evaluate performance of our tool by

SpringSim-TMS/DEVS 2016 April 3-6, Pasadena, CA, USA

comparing its different synchronization protocols, as well as a comparison to adevs. Related work is discussed in Section 5. Section 6 concludes the paper and gives future work.

2. BACKGROUND

This section briefly introduces two distinct synchronization protocols, as used by dxex. Furthermore, we make note of several new features of C++11.

2.1 Conservative Synchronization

The first synchronization protocol that we will introduce, is *conservative synchronization* [9]. In conservative synchronization, a node is allowed to progress in simulated time, independent of all other nodes, up to the point where it can guarantee that no causality errors can happen. When this point in time is reached, the node has to block until it is allowed to progress any further. In practice, this means that all nodes need to be aware of the current simulation time of all other nodes, and the time it takes an event to propagate (called *lookahead*). Several algorithms are defined in the literature to implement this behaviour. An overview is given in [9].

Deadlock can occur when a dependency cycle occurs and the amount of exchanged messages is low. Multiple algorithms are defined to handle this situation, such as deadlock avoidance and deadlock recovery.

The main advantage of conservative synchronization is that it has a low overhead if high parallelism exists between nodes. Each node can simulate in parallel, while sporadically notifying other nodes that they can progress even further. The disadvantage, however, is that the amount of parallelism is explicitly limited by the size of the lookahead. If a node can influence another (almost) instantaneously, no matter how rarely it occurs, the amount of parallelism is severely reduced. It is also up to the user to define the size of the lookahead, depending on how the model is known to behave. Slight changes in model behaviour can cause significant changes to the lookahead, and can therefore also have a significant influence on simulation performance.

2.2 Optimistic Synchronization

A completely different synchronization protocol is *optimistic synchronization* [13]. Whereas conservative synchronization would prevent causality errors at all costs, optimistic synchronization will allow them to happen, but correct them afterwards. Each node is allowed to progress in simulated time as much as possible, without taking note of the state of any other node. When an event arrives at a node, which is already further in simulated time, the node will have to roll back its state to right before the event would normally have to be processed. As the simulation time is now rolled back to before the event is processed, the event can simply be processed as if no causality error ever occured.

Rolling back the simulation time requires the node to store past model states, such that they can be restored later on. Furthermore, all incoming and outgoing events need to be stored as well. Incoming events need to be passed to the models again, when the correct simulation time has again been reached, and outgoing events need to be cancelled, as potentially a different series of output events would normally have been generated. Cancelling events, however, can cause further rollbacks, as the receiving node might also have to roll back its state. In practice, a single causality error could have significant repercussions on the complete simulation.

Further changes are required for unrecoverable operations, such as I/O (*e.g.*, tracing, writing to file, printing output) and memory management. Lightweight algorithms are still required to determine the lower bound of all simulation times, through the computation of a *Global Virtual Time* (GVT).

The main advantage is that performance is not limited by a small lookahead, caused by a very infrequent event. If an (almost) instantaneous event occurs rarely, performance will only be impacted if it occurs, and not at every simulation step. The main disadvantage is unpredictable performance and arbitrary cost of rollbacks due to the propagation of causality errors. If rollbacks occur frequently, simulation quickly becomes slow, as the overhead of the recovery mechanisms becomes significant. Apart from overhead in CPU time, a significant memory overhead is present: all intermediate states qre stored up to a point where it can be considered *irreversible*.

While optimistic synchronization does not explicitly depends on the lookahead, simulation performance is still bound by the lookahead implicitly.

2.3 C++11 Parallelism Features

Apart from various other additions, C++11 adds a wide range of portable synchronization primitives to the Standard Library. In earlier versions of the language standard, one had to resort to non-portable C implementations. Furthermore, C++11 is the first version of the standard that explicitly defines a multi-threaded abstract machine memory model. Most modern compilers support the full standard, allowing the kernels to be portable by default on any standard compliant platform.

Simulation values that need to be shared between nodes, can now be coded portable and efficiently through the use of atomic fields. This avoids latencies caused by the exchange of synchronization messages. For an in-depth study, we refer to [6].

3. MULTIPLE SYNCHRONIZATION PROTOCOLS

Historically, dxex is based on PythonPDEVS [21]. While Python is a good language to create software prototypes, its performance has proven to be insufficient to compete with other simulation kernels [19]. Dxex implements only a subset of PythonPDEVS, but makes some of its own additions. The core simulation algorithm and optimizations, however, are highly similar.

While we will not make a detailed comparison with PythonPDEVS here, but will attempt to provide a brief overview of some supported features. Similarly to PythonPDEVS, dxex supports direct connection [4], Dynamic Structure DEVS [1], termination conditions, and a modular tracing and scheduling framework [21]. But whereas PythonPDEVS

only supports optimistic synchronization, dxex support multiple synchronization protocols. This is in line with the design principles of PythonPDEVS: offer users the option to pass information on how to efficiently simulate the model. In our case, it now becomes possible to pass the simulation kernel the "hint" as to which synchronization protocol would be ideal for this model. Furthermore, the implementation in C++11 allows many more (static) optimizations, which were plainly impossible when using an interpreted language.

Note that, since there is no universal DEVS model standard, dxex models are incompatible with PythonPDEVS and vice versa. This is due to dxex models being grafted on C++11, whereas PythonPDEVS models are grafted on Python.

In the remainder of this section, we will elaborate on our prominent new feature: support for multiple synchronization protocols within the same simulation tool.

3.1 Synchronization protocols

We have previously shown that different synchronization protocols exist, with each of them being optimized for a specific kind of model. As no single synchronization protocol is ideal, a general purpose simulation tool should support multiple situations. Currently, however, most parallel simulation tools chose only a single synchronization protocol due to the inherent differences between these approaches. An uninformed choice on which to implement is insufficient, as performance is likely to be bad. We therefore argue that a real general purpose simulation tool should support sequential, conservative, and optimistic synchronization.

Each of them is applicable in specific model configurations. Conservative synchronization is ideal when high lookahead exists between different nodes, and barely any blocking is necessary. Optimistic synchronization is ideal when lookahead is unpredictable, or there are rare (almost) instantaneous events. Finally, sequential simulation is still required for models where parallelism is bad, causing significant overhead.

Sequential

Our sequential simulation algorithm is very similar to the one found in PythonPDEVS, including most of its optimizations. However, the simulation algorithm is designed to be overloaded by different synchronization protocol implementations. This way, the DEVS simulation algorithm is implemented once, but parts can be overwritten as needed. In theory, more synchronization protocols (*e.g.*, other algorithms for conservative synchronization) can be added without altering our design.

An overview of this design is given in Figure 1. It shows that there is a simulation `Core`, which has several `AtomicModels` connected to it. The default `Core` is merely the sequential simulation core. Its subclasses define specific variants of the simulation algorithm, like `ConservativeCore` (conservative synchronization), `OptimisticCore` (optimistic synchronization), and `DynamicCore` (Dynamic Structure DEVS).

Conservative

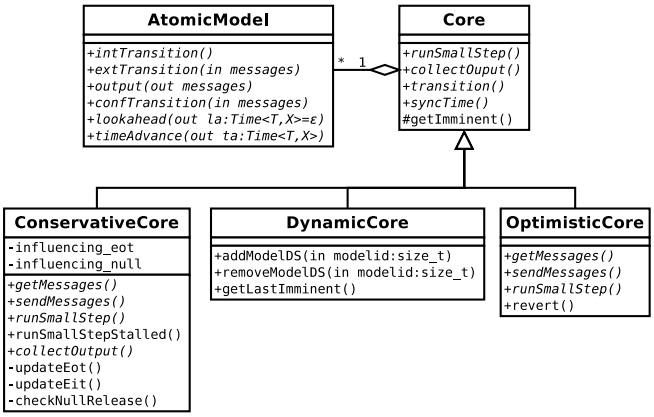

Figure 1. Dxex kernel design.

For conservative synchronization, each node determines the nodes it is influenced by. Each model provides a lookahead function, which determines the lookahead depending on the current simulation state. Within this time interval, it is an error if a model raises an event, thus violating its previous promise. The node uses this information to compute its earliest output time (EOT), and writes out the value in shared memory through the use of C++11 synchronization primitives.

In our implementation, we thus make explicit use of these new C++11 synchronization primitives. Whereas this was also possible in previous versions of the C++ standard, it was not a part of the standard itself. C++11 allows us to make our implementation portable, as well as more efficient: compilers might know of optimizations specific to atomic variables.

Optimistic

For optimistic synchronization, each node needs to keep track of all intermediate simulation states. This needs to be done carefully, in order to avoid unnecessary copies, and minimize the overhead induced for each transition function. We use Mattern's GVT algorithm [14] to determine the Global Virtual Time (GVT) using at most $2n$ synchronization messages. This process runs asynchronously from the simulation itself, thus there is no blocking whatsoever. Once the GVT is found, all nodes are informed of the new value, after which fossil collection is performed, as well as committing irreversible actions.

The main problem we encountered in our implementation is the agressive use of memory. Frequent memory allocation and deallocation caused significant overheads. This made us switch to the use of thread-local memory pools. Again, we made use of the specific features offered by C++11 that were not available in Python, or even previous versions of the C++ language standard.

3.2 Transparancy

A user must only provide one model, implemented in C++11, which can be simulated by each synchronization kernel. The exception is conservative synchronization: a lookahead function is required, whereas this is not possible in other synchronization kernels. Two options are possible: either a lookahead

function is always provided, even when it is not required and possibly not used, or using a default lookahead function if none is defined.

Always defining a lookahead function might seem redundant, especially if users will never use conservative synchronization. The more attractive option is for the simulation tool to provide a default lookahead function, defined by the minimum detected time advance. This lookahead value is most likely too small, but will prevent causality errors at the cost of performance. Depending on the model, simulation performance might still be faster than sequential simulation.

Defining a lookahead function is therefore recommended in combination with conservative synchronization, but is not a necessity.

4. PERFORMANCE

In this section, we evaluate the performance of our simulation tool. We show that the inclusion of multiple synchronization protocols does not decrease our performance to the point where it is unusable. To this end, we compare to adevs, one of the most efficient simulation kernels at this time [7]. A comparison is made for both the CPU and memory usage of both sequential simulation and parallel simulation.

We start of with a comparison of sequential simulation, to show how adevs and dxex relate in this simple case. We not only show that our approach does not influence performance negatively, but also that our main simulation algorithm, similar to the one of PythonPDEVS, is significantly faster than the one found in adevs. Similar differences, compared to adevs, can also be seen in the parallel simulation benchmarks. In the parallel simulation benchmarks, the benefit of our different synchronization protocols is also indicated.

For all benchmarks, results were all well within 1% deviation of the average, such that only the average is used in the remainder of this section. The same compilation flags were used for both adevs and dxex benchmarks ("`-O3`"). To guarantee comparable results, no IO was performed during the benchmarking phase. Simulation traces were used to verify that both adevs and dxex models have exactly the same behaviour. All benchmarks were performed using Linux, but our simulation tool works equally well on Windows and Mac.

4.1 Benchmarks

We use a selection of benchmarks, based on those found in the literature. Three different types of benchmark are defined, each for a different purpose:

1. *Queue* model, based on the HI model of DEVStone [11], creates a chain of atomic models, which are hierarchically nested in each other. A single generator will push events into the queue, which get processed by the processors after a fixed or random delay. It takes two parameters: width and depth, which determine the widht and depth of the hierarchy. This benchmark shows how the complexity of the simulation kernel behaves with an increasing amount of atomic models, and an increasingly deep hierarchy. If the processing delay is fixed for all processors, further insight is provided in the collision handling performance of the simulation kernel. An example for width 2 and depth 3 is shown in Figure 2.

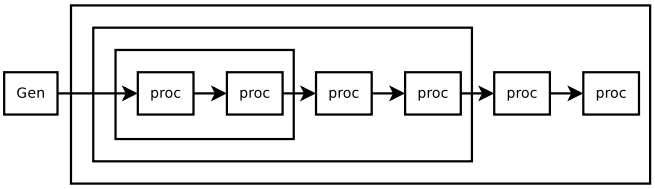

Figure 2. Queue model for depth 3 and width 2.

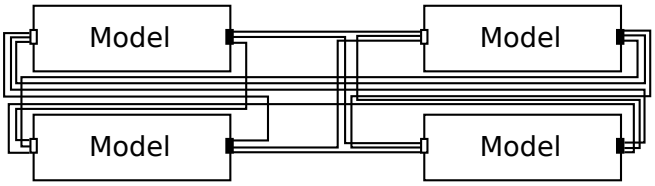

Figure 3. HighInterconnect model for four models.

2. *PHOLD* model, presented by [8]. It creates a set of n atomic models, and each model has exactly $n-1$ output ports: one for every other atomic model. Couplings are made such that each atomic model is directly connected to each other atomic model, such that every atomic model can directly send an event to every other atomic model. After a random delay, atomic models will send out an event to a randomly selected output port. Output port selection happens in two phases: first it is decided whether the event should be sent to an atomic model inside or outside of this coupled model. Afterwards, a uniform selection is made between the possible ports. It takes one parameter: the percentage of remote events, which influences the fraction of messages routed to other coupled models. This benchmark shows how the simulation kernel behaves in the presence of many local or remote events. An example for four models, split over two nodes, is shown in Figure 4.

3. *HighInterconnect* model, a merge of the HI model of DEVStone [11] and PHOLD [8]. It creates a structure similar to the one from PHOLD, but instead of $n-1$ output ports, every atomic model has only a single output port. All models are still connected to each other, but through the use of broadcasting: every model will receive a generated event. It takes one parameter: the number of models. This benchmark investigates the complexity of the routing algorithm. An example for four models is shown in Figure 3.

We opted to deviate from the DEVStone benchmark, as DEVStone tends towards unrealistic models since all internal and external transitions occur simultaneously. In our benchmark models, there is always the option for simultaneous transition functions (*fixed time advance*), or scattered transition functions (*random time advance*). Furthermore, they defined the use of an artificial load function, which easily skews the result, making the actual simulation algorithm barely comparable.

4.2 Sequential Simulation Execution Time

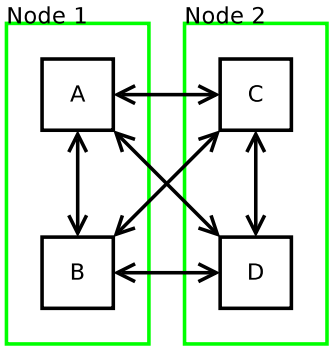

Figure 4. PHOLD model for four models, split over two nodes. In parallel simulation, each coupled model is simulated at a different node.

Despite our core contribution being mainly in the parallel simulation, we still value a comprehensive comparison of sequential simulation results. First, and foremost, as parallel simulation results are tightly linked to the sequential simulation results: parallel simulation merely adds a synchronization layer over different, essentially sequential, simulation kernels. Second, since parallel simulation results are validated through the use of adevs. To provide a more comprehensive comparison to adevs in the parallel simulation benchmarks, sequential simulation results need to be compared. Only the Queue and HighInterconnect models are relevant for sequential simulation.

Queue

In the Queue model, we increase both the width and depth simultaneously, causing a quadratic growth in the number of atomic models. As can be seen in Figure 5, dxex considerably outperforms adevs. Through careful analysis of profiling results, we determined that adevs spends much time in handling simulation messages, whereas this is mostly avoided due to the differently designed simulation algorithm of dxex. Both simulation tools have quadratically increasing execution times, though dxex is much faster thanks to its more efficient simulation control algorithms.

HighInterconnect

In the HighInterconnect model, we increase the number of atomic models, thus quadratically increasing the number of couplings. As can be seen in Figure 6, adevs now outperforms dxex by a fair margin. Analysis showed that this slowdown is caused by the high amount of exchanged events. Event creation is found to be much slower in dxex than it is in adevs, even despite the use of memory pools in dxex.

4.3 Parallel Simulation Execution Time

We now perform an analysis of parallel simulation execution times of our previously defined benchmarks. For dxex, we mention results for both conservative and optimistic synchronization. Since adevs supports only conservative synchronization, we don't mention optimistic synchronization results there. All experiments were performed using four simulation nodes, and executed on a quad-core machine.

We highlight two main results: (1) dxex conservative synchronization is competitive with adevs; (2) dxex optimistic

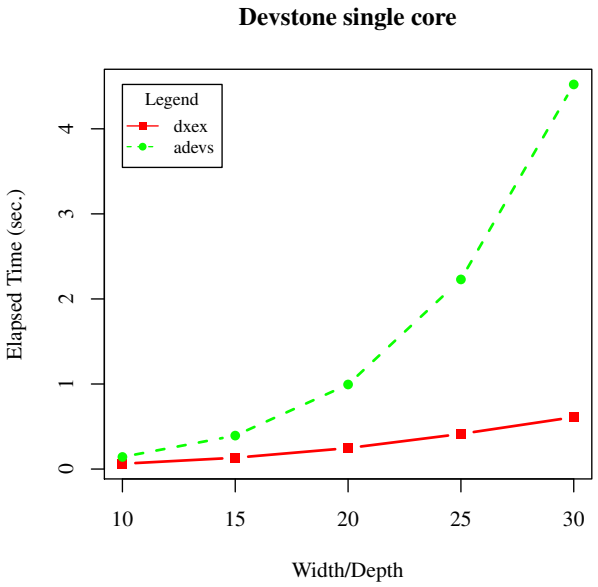

Figure 5. Queue benchmark results for sequential simulation.

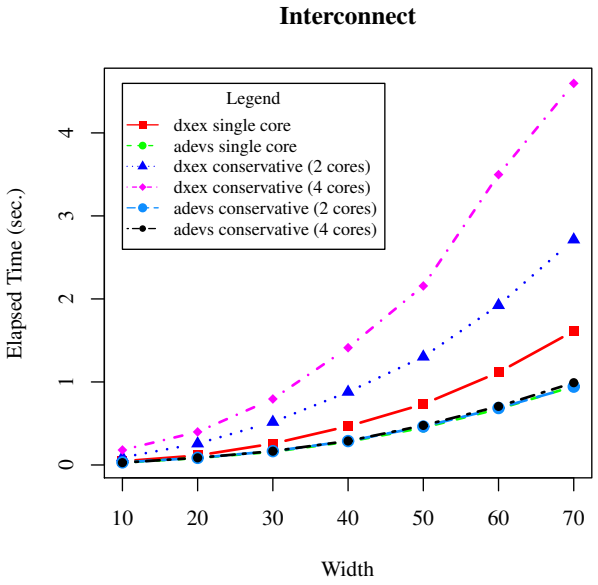

Figure 6. Interconnect benchmark results for sequential simulation.

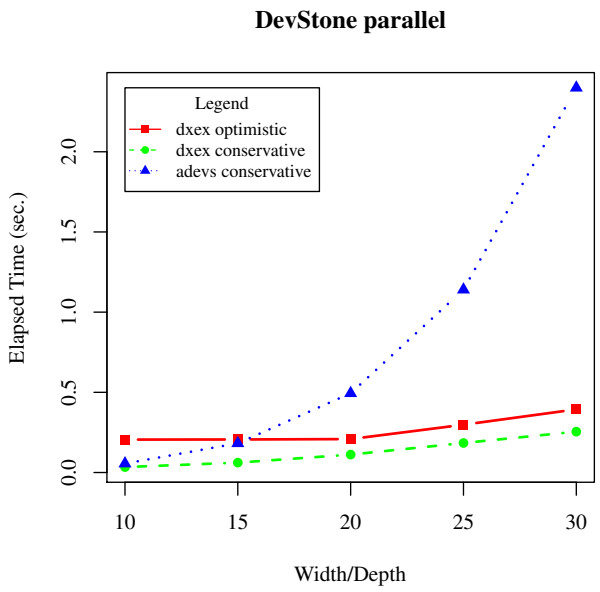

Figure 7. Queue benchmark results for parallel simulation using 4 cores.

synchronization is sometimes more efficient than conservative synchronization. This shows that our contribution, offering both conservative and optimistic synchronization, is indeed beneficial for a general-purpose simulation tools.

Queue

In the Queue model, we allocate the chain of models such that each node is responsible for a series of connected models. This minimizes the number of inter-node messages. As the model is a queue, however, the last models will only activate much later on in the simulation. Since these are allocated to seperate nodes, some nodes will remain idle until simulation has progressed sufficiently far.

Similar to the sequential benchmarks, Figure 7 shows that dxex again outperforms adevs, using both optimistic and conservative synchronization. Behaviour is exactly the same as in sequential simulation, but some speedup is achieved. In this case, conservative synchronization seems to be better than optimistic synchronization, at the cost of providing the lookahead.

PHold

In the Phold model, we first investigate the influence of the fraction of remote events on the speedup. When remote events are rare, optimistic synchronization will also have rare rollbacks, thus increasing performance. With more common remote events, however, optimistic synchronization quickly loses its advantage due to the more frequent rollbacks. Conservative synchronization, on the other hand, is largely unconcerned with the number of remote events: the mere fact that a remote event can happen, causes it to block. Even though a single synchronization protocol suffices in this case, it shows how different synchronization protocols respond differently to a changing model. Adevs, on the other hand, is significantly slower during conservative synchronization. Analysis of profiling results shows that this was caused by exception handling within the adevs simulation kernel.

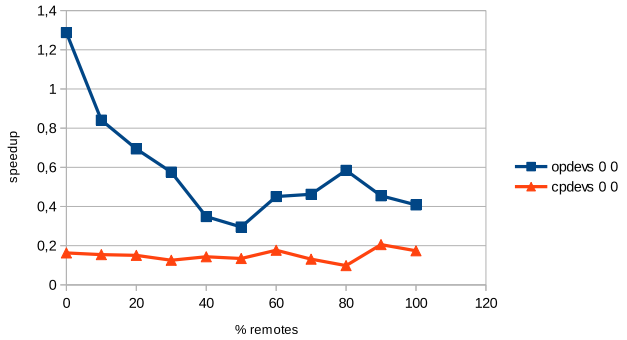

Figure 8. Phold benchmark results for parallel simulation using 4 cores, with varying fraction of remote events.

Secondly, we verify that our contribution fulfills our projected use case: a single model that can be tweaked to favor either conservative or optimistic synchronization. We slightly modified the Phold benchmark, to include high-priority events. Contrary to normal events, which offer a sufficiently large lookahead, high-priority events happen almost instantaneous, thus restricting lookahead to a very small value. Even though the normal events will occur most oftenly, a conservative implementation should always block since these high-priority events can always occur. An optimistic implementation, however, will simply go forward in simulation time and roll back the few times that these high-priority events happen. This situation closely mimics the case made in the comparison between both synchronization algorithms by [9].

Figure 9 shows how simulation performance is influenced by the fraction of high-priority events. If barely any high-priority events occur, conservative synchronization is penalized due to its excessive blocking, which often turned out to be unnecessary. Should many high-priority events occur, however, optimistic synchronization is penalized due to its mindless progression of simulation, which frequently needed to be reverted. These results show that there is no single perfect synchronization algorithm for this model. Depending on model configuration, either synchronization protocol might be better. We therefore claim that our contribution is invaluable for high performance simulation: depending on the observed communication behaviour, modellers can use the most appropriate synchronization protocol.

Interconnect

In the Interconnect model, we determine how broadcast communication is supported across multiple nodes. Results are shown in Figure 10 While conservative simulation delivers the expected speedups, dxex is still significantly slower than adevs. Interestingly, adevs does not achieve any noticable speedup from the use of multiple cores in this situation. Optimistic synchronization is not shown in this case: due to the high amount of exchanged events, all of which need to be stored in the sending *and* receiving node, it quickly runs out of memory. Several fixes to this problem have been proposed in the literature [9], but none of these is implemented by dxex.

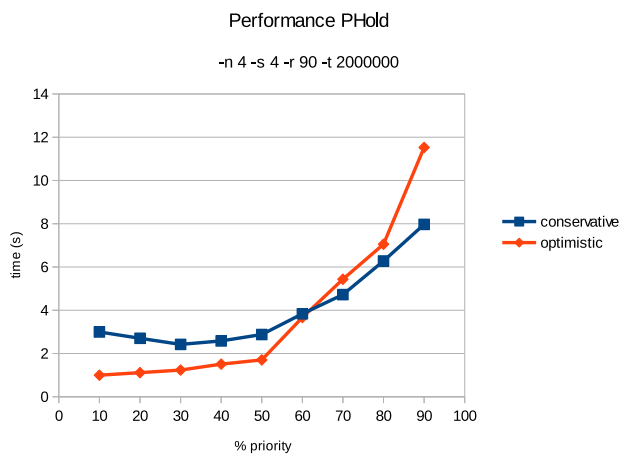

Figure 9. Phold benchmark results for parallel simulation using 4 cores, with varying amount of high-priority events.

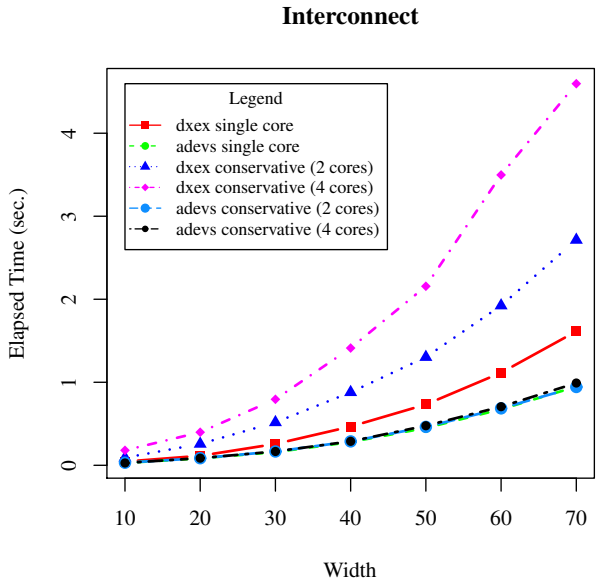

Figure 10. Interconnect benchmark results for parallel simulation.

4.4 Memory Usage

Apart from simulation execution time, memory usage during simulation is also of great importance. While execution time only becomes a problem if it takes way too long, coming short only a bit of memory can make simulation infeasible. We therefore also investigate memory usage of different synchronization protocols.

We do not tackle the problem of states that become too large for a single machine to hold. This problem can be mitigated by distribution over multiple machines, which neither dxex or adevs support. Comparison of memory usage should thus only be seen in the context of which models are feasible to be simulated by the tool.

Remarks

Both dxex and adevs use tcmalloc as memory allocator. Additionally, dxex uses memory pools to further reduce the frequency of expensive system calls (*e.g.*, malloc and free). Tcmalloc only gradually releases memory back to the OS, whereas our pools will not do so at all. If memory has been allocated once, it is, at least from a performance point of view, better to keep that memory in the pool indefinitely. Due to our motivation for memory usage analysis, we will only measure peak allocation. Profiling is done using Valgrind's massif tool [15].

Results

Figure 11 shows the memory used by each different benchmark. Results are in megabytes, and show the total memory footprint of the running application (*i.e.*, text, stack, and heap).

Unsurprisingly, optimistic synchronization results show very high memory use due to the saved states. Note the logarithmic scale that was used for this reason. Also, results for optimistic synchronization vary heavily depending on thread scheduling by the operating system, as this influences the drift between different nodes. Comparing similar approaches though, we notice that dxed and adevs have very similar memory use.

Conservative simulation always uses more memory than sequential simulation, as is to be expected. Additional memory is required for the multiple threads, but also to store all events that are processed simultaneously.

For the Phold benchmark, adevs using conservative synchronization took too long using our profiling tool, and was therefore aborted. Therefore, no results are shown for adevs.

5. RELATED WORK

Several DEVS simulation tools have already been implemented, which bear some similarity to our simulation tool. We discuss several dimensions of related work, as we try to compromise between different tools.

In terms of code design and philosophy, dxex is most related to PythonPDEVS [21]. Performance of PythonPDEVS was still decent, through the use of simulation and activity hints from the modeller. This allowed the kernel to optimize its internal data structures and algorithms for the specific model being executed. All changes were completely transparant to the model itself, and were completely optional. In this spirit,

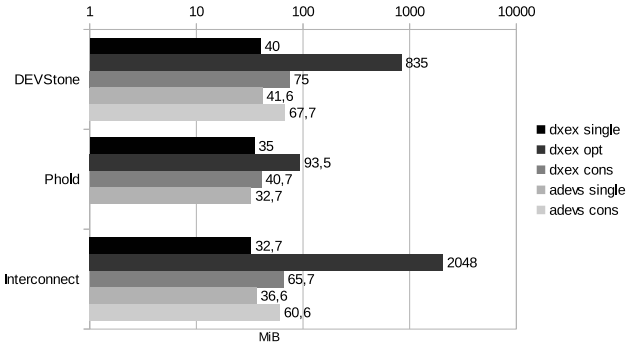

Figure 11. Memory usage results.

we now offer users the possibility to chose between different synchronization protocols. This allows them to chose for the most appropriate synchronization, depending on the actual model being simulated. Contrary to PythonPDEVS, however, dxex doesn't support model migrations [22] or activity hints [20].

While PythonPDEVS offers very fast turnaround cycles, due to the use of an interpreted language, simulation performance was easily outdone by compiled simulation kernels. In terms of performance, adevs [16] offers faster simulation, at the cost of a significant compilation time. The turnaround cycle in adevs is much slower though, specifically because the complete simulation kernel is implemented using templates, and thus in header files. This prevents use of the simulation kernel as a shared library. A similar compilation cycle is seen in PowerDEVS [3]. As a result, the complete simulation kernel has to be compiled again everytime. Dxex compromises, as was the case in other compiled simulation kernels, such as vle [17], by seperating the simulation kernel into a shared library. After the initial compilation of the simulation tool, only the model has to be compiled and linked to the library. This significantly shortens the turnaround cycle, while still offering good performance In terms of performance, it therefore comes close to, and sometimes beats, adevs. While a more extensive set of benchmarks is required to make accurate comparisons, initial results are promising. Despite its high performance, adevs does not support optimistic synchronization, which we have shown to be highly relevant.

Previous DEVS simulation tools have already implemented multiple synchronization protocols, though mostly in an adhoc way. For example CD++ [24] has both a conservative (CCD++ [12]) and optimistic (PCD++) [18]) variant. Despite the implementation of both, they are different projects entirely, and while they might support a similar set of features, they remain different from each other. Dxex, on the other hand, is a single project, where switching between different synchronization protocols is as simple as switching any other configuration parameter. CD++, however, implements both conservative and optimistic synchronization for distributed simulation, whereas we limit ourself to parallel simulation. By limiting our approach to parallel simulation, we are able to achieve higher speedups through the use of shared memory communication.

In summary, dxex tries to find the middle ground between the concepts of PythonPDEVS, the performance of adevs, and the multiple synchronization protocols of CD++.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced dxex, a new C++11-based Parallel DEVS simulation tool. Our main contribution is the implementation of different pluggable synchronization kernels for parallel simulation. We have shown that there are indeed models which can be simulated significantly faster using either synchronization protocol. Dxex allows the user to chose between both conservative and optimistic synchronization, as simply as any other configuration option. Notwithstanding this modularity, we have shown that dxex achieves performance similar to adevs, another very efficient DEVS simultion tool. Performance is measured both in CPU time, and memory usage.

Future work exists in several directions. First, we wish to make optimistic synchronization more tolerant to low-memory situations. In its current state, simulation will simply halt with an out-of-memory error. Having simulation control, which can throttle the speed of nodes that use up too much memory, has been shown to work in these situations [9]. In our implementation specifically, the out-of-memory problem is aggravated by a slow GVT implementation. Algorithms as those presented by [10] or [2] might help to alleviate this problem. Second, the idea of activity can be implemented for our simulation kernels, making it possible to dynamically switch between conservative and optimistic synchronization when changes in the model behaviour are detected. Third, activity algorithms, as already implemented by PythonPDEVS, could also be implemented in dxex, to determine how they influence simulation performance.

ACKNOWLEDGMENTS

This work was partly funded with a PhD fellowship grant from the Research Foundation - Flanders (FWO).

REFERENCES

1. Barros, F. J. Modeling formalisms for dynamic structure systems. *ACM Transactions on Modeling and Computer Simulation 7* (1997), 501–515.

2. Bauer, D., Yaun, G., Carothers, C. D., Yuksel, M., and Kalyanaraman, S. Seven-o'clock: A new distributed gvt algorithm using network atomic operations. In *Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation*, PADS '05, IEEE Computer Society (Washington, DC, USA, 2005), 39–48.

3. Bergero, F., and Kofman, E. PowerDEVS: a tool for hybrid system modeling and real-time simulation. *Simulation 87* (2011), 113–132.

4. Chen, B., and Vangheluwe, H. Symbolic flattening of DEVS models. In *Summer Simulation Multiconference* (2010), 209–218.

5. Chow, A. C. H., and Zeigler, B. P. Parallel DEVS: a parallel, hierarchical, modular, modeling formalism. In

Proceedings of the 26th Winter Simulation Conference, SCS (1994), 716–722.

6. De Munck, S., Vanmechelen, K., and Broeckhove, J. Revisiting conservative time synchronization protocols in parallel and distributed simulation. *Concurrency and Computation: Practice and Experience 26*, 2 (2014), 468–490.

7. Franceschini, R., Bisgambiglia, P.-A., Touraille, L., Bisgambiglia, P., and Hill, D. A survey of modelling and simulation software frameworks using Discrete Event System Specification. In *2014 Imperial College Computing Student Workshop*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2014), 40–49.

8. Fujimoto, R. M. Performance of Time Warp under synthetic workkloads. In *Proceedings of the SCS Multiconference on Distributed Simulation* (1990).

9. Fujimoto, R. M. *Parallel and Distributed Simulation Systems*, 1st ed. John Wiley & Sons, Inc., New York, NY, USA, 1999.

10. Fujimoto, R. M., and Hybinette, M. Computing global virtual time in shared-memory multiprocessors. *ACM Trans. Model. Comput. Simul. 7*, 4 (Oct. 1997), 425–446.

11. Glinsky, E., and Wainer, G. DEVStone: a benchmarking technique for studying performance of DEVS modeling and simulation environments. In *Proceedings of the 2005 9th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications* (2005), 265–272.

12. Jafer, S., and Wainer, G. Flattened conservative parallel simulator for DEVS and CELL-DEVS. In *Proceedings of International Conferences on Computational Science and Engineering* (2009), 443–448.

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

14. Mattern, F. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing 18*, 4 (1993), 423–434.

15. Nethercote, N., and Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not. 42*, 6 (jun 2007), 89–100.

16. Nutaro, J. J. ADEVS. `http://www.ornl.gov/~1qn/adevs/`, 2015.

17. Quesnel, G., Duboz, R., Ramat, E., and Traoré, M. K. VLE: a multimodeling and simulation environment. In *Proceedings of the 2007 Summer Simulation Multiconference* (2007), 367–374.

18. Troccoli, A., and Wainer, G. Implementing Parallel Cell-DEVS. In *Proceedings of the 2003 Spring Simulation Symposium* (2003), 273–280.

19. Van Tendeloo, Y. Activity-aware DEVS simulation. Master's thesis, University of Antwerp, Antwerp, Belgium, 2014.

20. Van Tendeloo, Y., and Vangheluwe, H. Activity in pythonpdevs. In *Activity-Based Modeling and Simulation* (2014).

21. Van Tendeloo, Y., and Vangheluwe, H. The Modular Architecture of the Python(P)DEVS Simulation Kernel. In *Spring Simulation Multi-Conference*, SCS (2014), 387 – 392.

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

23. Vangheluwe, H. DEVS as a common denominator for multi-formalism hybrid systems modelling. *CACSD. Conference Proceedings. IEEE International Symposium on Computer-Aided Control System Design* (2000), 129–134.

24. Wainer, G. CD++: a toolkit to develop DEVS models. *Software: Practice and Experience 32*, 13 (2002), 1261–1306.

25. Zeigler, B. P., Praehofer, H., and Kim, T. G. *Theory of Modeling and Simulation*, second ed. Academic Press, 2000.