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 these 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 [24] 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 [22]. 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 [8] have been applied to DEVS simulation. While several parallel DEVS simulation kernels exist, they are often limited to a single synchronization protocol. The 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.

This paper introduces DEVS-Ex-Machina[1] ("dxex"), our simulation tool which offers multiple synchronization protocols: no synchronization (sequential execution), conservative synchronization, or optimistic synchronization. The selected synchronization protocol is transparent to the simulated model: users should merely determine, which protocol they wish to use. Users who simulate a wide variety of models, with different ideal synchronization protocols, can simply run the same model with different synchronization protocols.

Our tool is based on PythonPDEVS, but implemented in C++11 for increased performance. Unlike PythonPDEVS dxex only supports multicore parallelism.

We implemented a model that, depending on a single parameter, changes its ideal synchronization protocol. Dxex, then, is used to compare simulation using exactly the same tool, but with a varying synchronization protocol. With dxex 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 [20, 6].

The remainder of this paper is organized as follows: Section 2 introduces the necessary background on synchronization protocols. Section 3 elaborates on our design that enables this flexibility. In Section 4, we evaluate performance of our tool by comparing its different synchronization protocols, and by comparing to adevs. Related work is discussed in Section 5. Section 6 concludes the paper and gives future work.

2. BACKGROUND
This section briefly introduces the synchronization protocols used by dxex: conservative and optimistic synchronization.

[1] `https://bitbucket.org/bcardoen/devs-ex-machina`

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

2.1 Conservative Synchronization

The first synchronization protocol we introduce is *conservative synchronization* [8]. In conservative synchronization, a node progresses independent of all other nodes, up to the point in time where it can guarantee that no causality errors happen. When simulation reaches this point, the node blocks until it can guarantee a new time until which no causality errors occur. In practice, this means that all nodes are aware of the current simulation time of all other nodes, and the time it takes an event to propagate (called *lookahead*). Deadlocks can occur due to a dependency cycle of models. Multiple algorithms are defined in the literature to handle both the core protocol, as well as resolution schemes to handle or avoid the deadlocks [8].

The main advantage of conservative synchronization is its low overhead if the lookahead is high. Each node then simulates in parallel, and sporadically notifies other nodes about its local simulation time. The disadvantage, however, is that the amount of parallelism is explicitly limited by the lookahead. If a node can influence another (almost) instantaneously, no matter how rarely it occurs, the amount of parallelism is severely reduced. The user is required to define the lookahead, using knowledge about the model's behaviour. Defining lookahead is not always a trivial task if there is no detailed knowledge of the model. Even slight changes in the model can change to the lookahead, and can therefore have a significant influence on simulation performance.

2.2 Optimistic Synchronization

A completely different synchronization protocol is *optimistic synchronization* [12]. Whereas conservative synchronization prevents causality errors at all costs, optimistic synchronization allows them to happen, but corrects them. Each node simulates as fast as possible, without taking note of any other node. It assumes that no events occur from other nodes, unless it has explicitly received one at that time. When this assumption is violated, the node rolls back its simulation time and state to right before the moment when the event has to be processed. As simulation is rolled back to a time prior to the event must be processed, the event can then be processed as if no causality error ever occurred.

Rolling back simulation time requires the node to store past model states, such that they can be restored later. All incoming and outgoing events need to be stored as well. Incoming events are injected again after a rollback, when their time has been reached again. Outgoing events are cancelled after a rollback, through the use of anti-messages, as potentially different output events have to be 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 can have significant repercussions.

Further changes are required for unrecoverable operations, such as I/O and memory management. These are only executed after the lower bound of all simulation times, called *Global Virtual Time* (GVT), has progressed beyond their execution time.

The main advantage is that performance is not limited by a small lookahead, caused by a very infrequent event. If an (almost) instantaneous event rarely occurs, performance is only impacted when it occurs, and not at every simulation step. The disadvantage is unpredictable performance due to the arbitrary cost of rollbacks and their propagation. If rollbacks occur frequently, state saving and rollback overhead can cause simulation to grind to a halt. Apart from overhead in CPU time, a significant memory overhead is present: all intermediate states are stored up to a point where it can be considered *irreversible*.

Note that, while optimistic synchronization does not explicitly depends on lookahead, performance still implicitly depends on lookahead.

3. MULTIPLE SYNCHRONIZATION PROTOCOLS

Historically, dxex is based on PythonPDEVS [20]. Python is a good language for prototypes, but performance has proven insufficient to compete with other simulation kernels [18]. Dxex is a C++11-based implementation of PythonPDEVS, but implements only a subset of PythonPDEVS, while making some of its own additions. So while the feature set is not too comparable, the architectural design, core simulation algorithm, and optimizations, are highly similar.

We will not make a detailed comparison with PythonPDEVS here, but only mention some supported features. Dxex supports, similarly to PythonPDEVS, the following features: direct connection [4], Dynamic Structure DEVS [1], termination conditions, and a modular tracing and scheduling framework [20]. But whereas PythonPDEVS only supports optimistic synchronization, dxex support multiple synchronization protocols (though only in parallel). This is in line with the design principle of PythonPDEVS: allow users to pass performance hints to the simulation kernel. In our case, a user can pass the simulation kernel the synchronization protocol to use for this model. Our implementation in C++11 also allows for optimizations which were plainly impossible in an interpreted language.

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, which are offered transparently to the model.

3.1 Synchronization protocols

We previously explained the existence of different synchronization protocols exist, each optimized for a specific kind of model. As no single synchronization protocol is ideal for all models, a general purpose simulation tool should support multiple protocols. Currently, most parallel simulation tools choose only a single synchronization protocol due to the inherent differences between protocols. An uninformed choice on which one to implement is insufficient, as performance

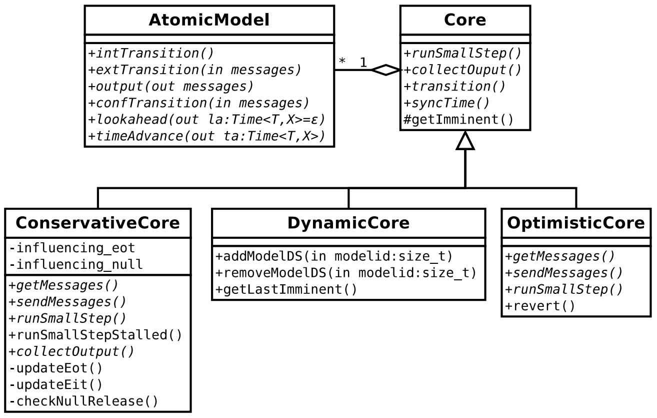

Figure 1. Dxex kernel design.

will likely be bad. We argue that a real general purpose simulation tool should support sequential, conservative, and optimistic synchronization, as is the case for dxex.

These different protocols relate to three different model characteristics. Conservative synchronization for when high lookahead exists between different nodes, and barely any blocking is necessary. Optimistic synchronization for when lookahead is unpredictable, or there are rare (almost) instantaneous events. Finally, sequential simulation is still required for models where parallelism is bad, where all protocols actually slow down simulation.

Sequential
Our sequential simulation algorithm is very similar to the one found in PythonPDEVS, including many optimizations. Minor modifications were made, though, such that it can be overloaded by different synchronization protocol implementations. This way, the DEVS simulation algorithm is implemented once, but parts can be overridden as needed. In theory, more synchronization protocols (*e.g.*, other algorithms for conservative synchronization) can be added without altering our design.

An overview of dxex's design is given in Figure 1. It shows that there is a simulation `Core`, which simulates the `AtomicModels` connected to it. The superclass `Core` is merely the sequential simulation core, but can be used as-is. Subclasses define specific variants, such as `ConservativeCore` (conservative synchronization), `OptimisticCore` (optimistic synchronization), and `DynamicCore` (Dynamic Structure DEVS).

Conservative
For conservative synchronization, each node must determine the nodes it is influenced by. Each model needs to provide a lookahead function, which determines the lookahead depending on the current simulation state. Within the returned time interval, the model promises not to raise an event. A node aggregates this information to computes its earliest output time (EOT). This value is written out in shared memory, where it can be read out by all other nodes.

Reading and writing to shared memory is done through the use of the new C++11 synchronization primitives. Whereas this was also possible in previous versions of the C++ standard, by falling back to non-portable C functions, it was not a part of the C++ language standard. C++11 further allows us to make the implementation portable, as well as more efficient: the compiler might know of optimizations specific to atomic variables.

Optimistic
For optimistic synchronization, each node must be able to roll back to a previous point in time. This is often implemented through the use of state saving. This needs to be done carefully in order to avoid unnecessary copies, and minimize the overhead. We use the default: explicitly save each and every intermediate state. Mattern's algorithm [13] is used to determine the GVT, as it runs asynchronously and uses only $2n$ synchronization messages. Once the GVT is found, all nodes are informed of the new value, after which fossil collection is performed, and irreversible actions are committed.

The main problem we encountered in our implementation is the aggressive use of memory. Frequent memory allocation and deallocation caused significant overheads, certainly when multiple threads do so concurrently. This made us switch to the use of thread-local (using `tcmalloc`) memory pools. Again, we made use of specific new features of C++11, that were not available in Python, or even previous versions of the C++ language standard.

3.2 Transparency
We define simulation kernel transparency as having a single model, which always can be executed on each supported synchronization kernel, without any modifications. User should thus only provide one model, implemented in C++11, which can be either using sequential execution, using conservative synchronization, or using optimistic synchronization. Switching between simulation kernels is as simple as altering the simulation termination time. The exception is conservative synchronization, where a lookahead function is required, which is not used in other synchronization kernels. Two options are possible: either a lookahead function must always be provided, even when it is not required and possibly not used, or we use a default lookahead function if none is defined.

Always defining a lookahead function might seem redundant, especially if users will never use conservative synchronization. Especially since defining the lookahead is often nontrivial and dependent on intricate model details. The more attractive option is for the simulation tool to provide a default lookahead function, such that simulation can run anyway, but likely not at peak 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, as a default $epsilon$ (*i.e.*, the smallest representable timestep) is used otherwise.

4. PERFORMANCE
In this section, we evaluate the performance of different synchronization protocols in dxex. We also compare to adevs,

currently one of the most efficient simulation kernels [6], to show that our modularity does not impede performance. CPU time and memory usage is compared for both sequential and parallel simulation.

We start off with a comparison of sequential simulation, to show how adevs and dxex relate in this simple case. For the parallel simulation benchmarks, results are presented for both conservative and optimistic synchronization.

For all benchmarks, results are well within a 5% 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 -flto`"). To guarantee comparable results, no I/O was performed during benchmarks. Before benchmarking, simulation traces were used to verify that adevs and dxex return exactly the same simulation results. Benchmarks were performed using Linux, but our simulation tool works equally well on Windows and Mac. The exact parameters for each benchmark can be found in the repository, as well as the data used in this paper.

4.1 Benchmarks

We use three different benchmarks, which cover different aspects of the simulation kernel:

1. The *Queue* model, based on the *HI* model of DEVStone [10], creates a chain of hierarchically nested atomic DEVS models. A single generator pushes events into the queue, which are processed by the processors after a random delay. It takes two parameters: width and depth, which determine the width and depth of the hierarchy. This benchmark shows how the complexity of the simulation kernel behaves for an increasing amount of atomic models, and an increasingly deep hierarchy. An example for width and depth 2 is shown in Figure 2.

2. The *PHOLD* model, presented by [7], creates n atomic models, where each model has exactly $n - 1$ output ports. Each atomic model is directly connected to every other atomic model. After a random delay, an atomic model sends 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 at the same node. Afterwards, a uniform selection is made between the remaining ports. The model takes one parameter: the percentage of remote events, which determines the fraction of messages routed to other nodes. 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. The *HighInterconnect* model, a merge of PHOLD [7] and the *HI* model of DEVStone [10], creates n atomic models, where each model has exactly one output port. Similar to PHOLD, all models are connected to one another, but all through the same port: every model receives each generated event. The model takes one parameter: the number of models. This benchmark investigates the complexity of event routing, and how the simulation kernel handles many simultaneous events. An example for four models is shown in Figure 3.

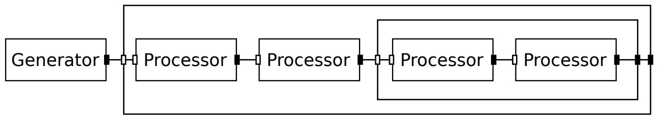

Figure 2. Queue model for depth and width 2.

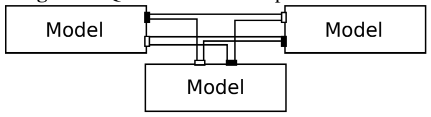

Figure 3. HighInterconnect model for three models.

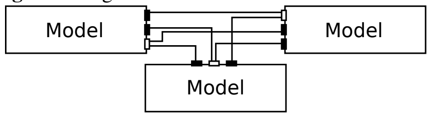

Figure 4. PHOLD model for three models.

4.2 Sequential Simulation Execution Time

Despite our core contribution being on parallel simulation, we still value a comparison of sequential simulation results. First, and foremost, parallel simulation results are tightly linked to sequential simulation results. Parallel simulation is achieved through synchronization of multiple sequential simulation kernels. Second, parallel simulation results are validated through the use of adevs. To provide a well-founded comparison to adevs in the parallel simulation benchmarks, sequential simulation results also need to be compared.

Only the Queue and HighInterconnect models are relevant for sequential simulation, so we will not touch upon the PHold model yet.

Queue

In the Queue model, we increase both width and depth simultaneously. For example, the 400 models configuration is obtained with a width and depth of 20. As can be seen in Figure 5, dxex considerably outperforms adevs. Through profiling, we found that adevs spends much time routing the simulation messages (*e.g.*, *, @) through the hierarchy, whereas dxex avoids this throuhg its flattened model structure. Both simulation tools have similar complexities, 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 and the number of external transitions. As can be seen in Figure 6, adevs outperforms dxex by a fair margin. Analysis showed that this is caused by the high amount of exchanged events: event creation is much slower in dxex than it is in adevs, despite dxex's use of memory pools.

4.3 Parallel Simulation Execution Time

Next we analyse parallel simulation of our previously defined benchmarks. For dxex, we mention results of both conservative and optimistic synchronization. Since adevs supports only conservative synchronization, we don't mention optimistic synchronization results there. All experiments were performed using up to six simulation nodes, executed on a hexa-core machine.

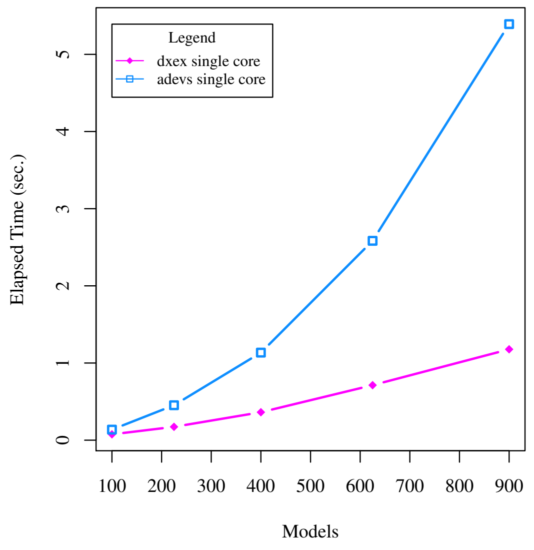

Figure 5. Queue benchmark results for sequential simulation.

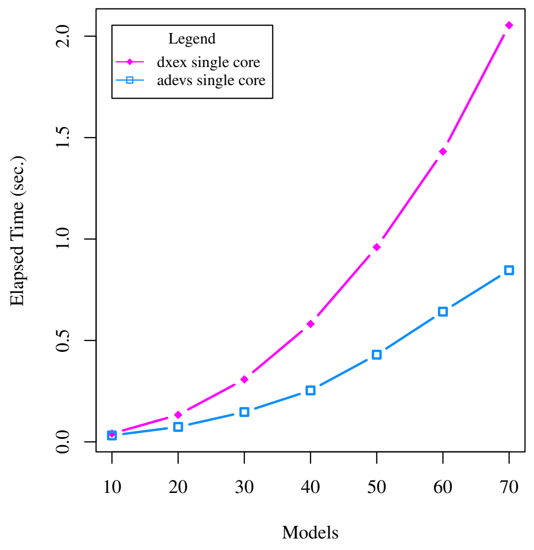

Figure 6. Interconnect benchmark results for sequential simulation.

We highlight two main results: (1) dxex conservative synchronization is competitive with adevs; (2) dxex optimistic 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, models further in the chain only activate later in the simulation. Since these are allocated to separate nodes, some nodes remain idle until simulation has progressed sufficiently far.

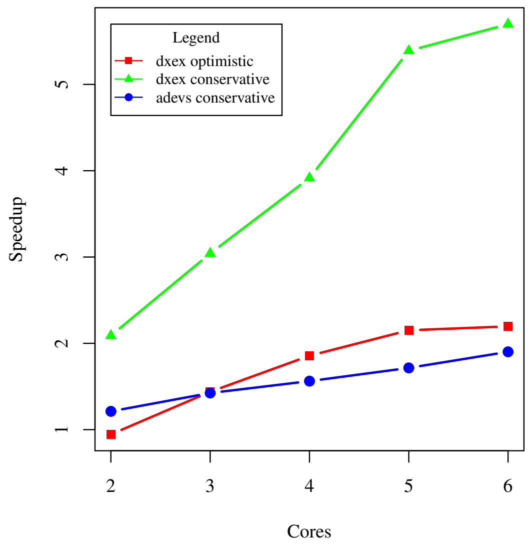

Figure 7. Queue benchmark results for parallel simulation of a model of fixed size.

Similar to the sequential benchmarks, Figure 7 shows that dxex outperforms adevs, but now in terms of speedup. Results indicate that our implementation of conservative synchronization achieves much higher speedups than adevs. Approaching near linear speedup, this simulation is the ideal case for our conservative implementation, as there are no dependency cycles between models. Conservative synchronization also seems to be better than optimistic synchronization in this case, 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 rarely has to roll back, thus increasing performance. With more common remote events, however, optimistic synchronization quickly slows down due to frequent rollbacks. Conservative synchronization, on the other hand, is mostly 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 is always ideal in this case, it already shows that different synchronization protocols respond differently to a changing model. Adevs is again significantly slower during conservative synchronization. Analysis of profiling results shows that exception handling in adevs is the main cause. To keep the models equivalent, the adevs version does not provide the {begin,end}Lookahead methods, which accounts for the exception handling.

We further 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, high-priority events happen almost instantaneously, restricting lookahead to a very small value. Even when normal events occur most often, conservative synchronization always blocks until it can make guarantees. Optimistic synchronization, however, simply goes

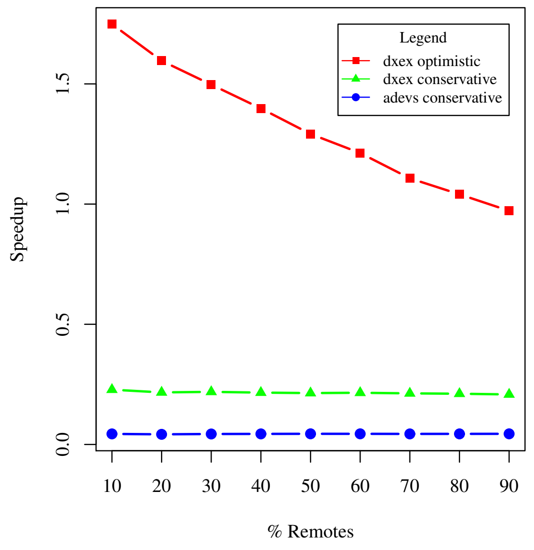

Figure 8. Phold benchmark results for parallel simulation using six cores, 4 atomics per node, with varying fraction of remote events.

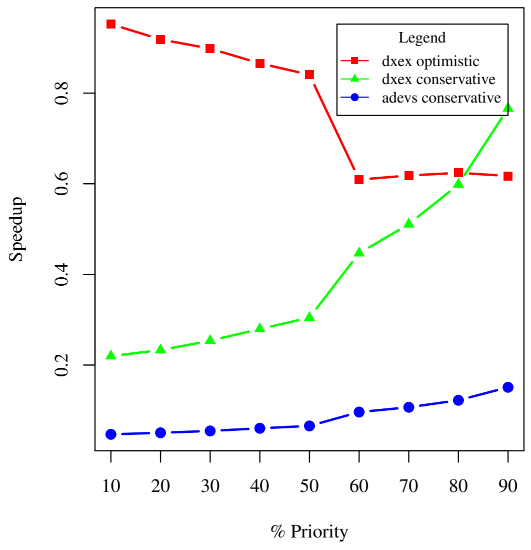

Figure 9. Phold benchmark results for parallel simulation using six cores, with varying amount of high-priority events.

forward in simulation time and rolls back when these high-priority events happen. This situation closely mimics the case made in the comparison between both synchronization algorithms by [8].

Figure 9 shows how simulation performance is influenced by the fraction of these 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. When many high-priority events occur, optimistic synchronization is penalized due to its mindless progression of simulation, which frequently needs to be rolled back. Results show that there is no single perfect synchronization algorithm for this model: depending on configuration, either synchronization protocol might be better. We have shown that our contribution is invaluable for high performance simulation: depending on the expected behaviour, modellers can choose the most appropriate synchronization protocol.

Interconnect
In the Interconnect model, we determine how broadcast communication is supported across multiple nodes. The number of models is now kept constant at eight. Results are shown in Figure 10. When the number of nodes increases, performance decreases due to increasing contention in conservative simulation and an increasing number of of rollbacks in optimistic simulation. All models depend on each other and have no computational load whatsoever, negating any possible performance gain by executing the simulation in parallel.

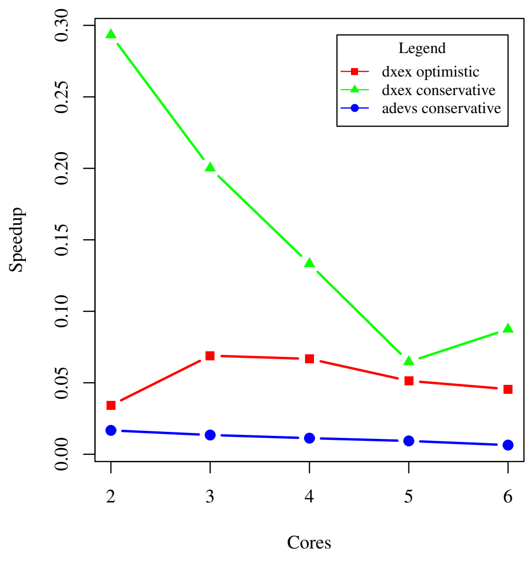

Figure 10. Interconnect benchmark results for parallel simulation.

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 unfeasible. We therefore also investigate memory usage of different synchronization protocols, and again compare to adevs.

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.

Remarks
Both dxex and adevs use `tcmalloc` as memory allocator, allowing for thread-local allocation. 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. Due to our motivation for memory usage analysis, we will only measure peak allocation. Profiling is done using Valgrind's massif tool [14].

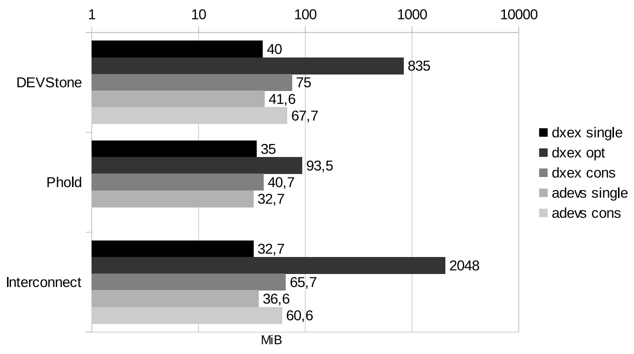

Figure 11. Memory usage results.

Results
Figure 11 shows the memory used by the different benchmarks. 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 usage due to the saved states. Note the logarithmic scale that was used for this reason. Optimistic synchronization results vary heavily depending on thread scheduling by the operating system, as this influences the drift between nodes. Comparing similar approaches, we notice that dxex 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 similar DEVS simulation tools have already been implemented, though they differ in key aspects. 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 [20]. Performance of PythonPDEVS was still decent, through the use of simulation and activity hints from the modeler. This allowed the kernel to optimize its internal data structures and algorithms for the specific model being executed. All changes were completely transparent to the model, and were completely optional. In this spirit, we offer users the possibility to choose between different synchronization protocols. This allows users to choose the most appropriate synchronization protocol, depending on the model. Contrary to PythonPDEVS, however, dxex doesn't support distribution, model migrations [21], or activity hints [19].

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 [15] offered much 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 in header files. As a result, the complete simulation kernel has to be compiled again every time. Dxex compromises, as vle [16] or PowerDEVS [3], by separating 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, dxex is shown to be competitive with 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 none have done it in a strictly modular way that allows straightforward protocol switching for a single given model. For example CD++ [23] has both a conservative (CCD++ [11]) and optimistic (PCD++) [17]) variant. Despite the implementation of both protocols, they are different projects entirely, and are incompatible with modern compilers. 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 ourselves 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 DEVS-Ex-Machina ("dxex"), a new C++11-based Parallel DEVS simulation tool. Our main contribution is the implementation of multiple synchronization protocols for parallel multicore simulation. We have shown that there are indeed models which can be simulated significantly faster using either synchronization protocol. Dxex allows the user to choose between either conservative or optimistic synchronization as simple as any other configuration option. Notwithstanding this modularity, dxex achieves performance competitive to adevs, another very efficient DEVS simulation tool. Performance is measured both in elapsed time, and memory usage.

Future work is possible in several directions. Firstly, our implementation of optimistic synchronization should be 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 [8]. Faster GVT implementations, such as those presented by [9] and [2], might further help to minimize this problem. Secondly, the idea of activity can be implemented for our simulation kernels, making it possible to dynamically switch between conservative and optimistic synchronization when behavioural changes are detected. Thirdly, activity algorithms, as already implemented by PythonPDEVS, can 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. 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.
7. Fujimoto, R. M. Performance of Time Warp under synthetic workkloads. In *Proceedings of the SCS Multiconference on Distributed Simulation* (1990).
8. Fujimoto, R. M. *Parallel and Distributed Simulation Systems*, 1st ed. John Wiley & Sons, Inc., New York, NY, USA, 1999.
9. Fujimoto, R. M., and Hybinette, M. Computing Global Virtual Time in shared-memory multiprocessors. *ACM Transactions on Modeling and Computer Simulation 7*, 4 (Oct. 1997), 425–446.
10. 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.
11. 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.
12. Jefferson, D. R. Virtual time. *ACM Transactions on Programming Languages and Systems 7*, 3 (July 1985), 404–425.
13. Mattern, F. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing 18*, 4 (1993), 423–434.
14. Nethercote, N., and Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not. 42*, 6 (jun 2007), 89–100.
15. Nutaro, J. J. ADEVS. `http://www.ornl.gov/~1qn/adevs/`, 2015.
16. 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.
17. Troccoli, A., and Wainer, G. Implementing Parallel Cell-DEVS. In *Proceedings of the 2003 Spring Simulation Symposium* (2003), 273–280.
18. Van Tendeloo, Y. Activity-aware DEVS simulation. Master's thesis, University of Antwerp, Antwerp, Belgium, 2014.
19. Van Tendeloo, Y., and Vangheluwe, H. Activity in PythonPDEVS. In *Activity-Based Modeling and Simulation* (2014).
20. 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.
21. 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.
22. 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.
23. Wainer, G. CD++: a toolkit to develop DEVS models. *Software: Practice and Experience 32*, 13 (2002), 1261–1306.
24. Zeigler, B. P., Praehofer, H., and Kim, T. G. *Theory of Modeling and Simulation*, second ed. Academic Press, 2000.