Journal Title
XX(X):1–23

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

A PDEVS Simulator Supporting Multiple Synchronization Protocols: Implementation and Performance Analysis

Ben Cardoen[1], Stijn Manhaeve[1], Yentl Van Tendeloo[1], and Jan Broeckhove[1]

Abstract

With the ever increasing complexity of simulation models, parallel simulation becomes necessary to perform 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 and based on PythonPDEVS, supporting both conservative and optimistic synchronization protocols. The simulator not only supports both protocols but also has the capability to switch between them at runtime. The simulator can combine each synchronization protocols with either a threaded or sequential implementation of the PDEVS protocol. We evaluate the performance gain obtained by choosing the most appropriate synchronization protocol. A comparison is made to adevs in terms of CPU time and memory usage, to show that our modularity does not hinder performance. We compare the speedup obtained by synchronization with that of the inherent parallelism of PDEVS in isolation and combination, and contrast the results with the theoretical limits. We further allow for an external component to gather simulation statistics, on which runtime switching between the different synchronization protocols can be based. The effects of allocation on our synchronization protocols is also studied.

Submission for the *Special Issue of Simulation: SpringSim 2016 special issue*.

[1]University of Antwerp, Belgium

Corresponding author:
Yentl Van Tendeloo
University of Antwerp
Middelheimlaan 1
2020 Antwerp, Belgium
Email: Yentl.VanTendeloo@uantwerpen.be

Prepared using sagej.cls *[Version: 2015/06/09 v1.01]*

Introduction

DEVS [1] 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 [2]. 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. Whereas

Parallel DEVS [3] was introduced to increase parallelism, its inherent parallelism is often insufficient [4]. Several synchronization protocols from the discrete event simulation community [5] have been applied to (Parallel) DEVS simulation [6]. With synchronization protocols, different simulation kernels can be at different points in simulated time, significantly increasing parallelism at the cost of synchronization overhead. While several parallel DEVS simulation tools 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 for specific models [7]. The parallel simulation capabilities of current tools are therefore limited to specific domains.

This paper introduces *DEVS-Ex-Machina*[*] ("*dxex*"): our simulation tool [8] which offers 6 synchronization protocols. The simulation tool can spawn multiple kernels, each of which has its own simulation time and simulation control, though they may exchange timestamped events with each other. Each kernel can spawn multiple threads, each of which will be mapped to a physical CPU core by the operating system. We note the presence of two kinds of synchronization: inter-kernel and intra-kernel. Inter-kernel synchronization uses either no synchronization, conservative synchronization, or optimistic synchronization. Intra-kernel synchronization uses either no synchronization or exploits the inherent parallelism of Parallel DEVS to allocate concurrent transitions to separate threads. The inherent parallelism in the intra-kernel synchronization protocol is used pSim in the remainder of this paper.

This results in the following set of synchronization protocol combinations:

- No inter-kernel synchronization, no intra-kernel synchronization (*NN*)
- Conservative inter-kernel synchronization, no intra-kernel synchronization (*CN*)
- Optimistic inter-kernel synchronization, no intra-kernel synchronization (*ON*)
- No inter-kernel synchronization, pSim intra-kernel synchronization (*NP*)
- Conservative inter-kernel synchronization, pSim intra-kernel synchronization (*CP*)
- Optimistic inter-kernel synchronization, pSim intra-kernels synchronization (*OP*)

When refering to a set of them, we use the notation *x* to denote any possible algorithm. For example, *xP* refers to NP, CP, and OP.

The selected synchronization protocol is transparent to the simulated model: users only determine the protocol to use. Users who simulate a wide variety of models, with different ideal synchronization protocols, can keep using the exact same tool, but with different synchronization protocols. As model behaviour, and thus the ideal synchronization protocol, might change during simulation, runtime switching of synchronization protocols is also supported. This runtime switching can be based on performance metrics, which are logged during simulation. Information is made available to a separate component, where a choice can be made about which synchronization protocol to use. Additionally, we investigate how model allocation influences the performance of our synchronization protocols. To this end, we have included an allocation component in our simulation kernel.

Our tool is based on *PythonPDEVS* [9], but implemented in C++11 for increased performance, using features from the new C++14 standard when supported by the compiler. Unlike *PythonPDEVS*, *dxex* only supports multicore parallelism, thus no distributed simulation.

Using several benchmark models, we demonstrate the factors influencing the performance under a given synchronization protocol. Additionally, we investigate a model which changes its behaviour (and corresponding ideal synchronization protocol) during simulation. *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, and through its modularity, users could even implement their own, or variants of existing ones. To verify that this modularity does not hamper performance, we compare to *adevs* [10], another Parallel DEVS simulation tool.

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

The remainder of this paper is organized as follows: Section BACKGROUND introduces the necessary background on synchronization protocols. Section DEVS-EX-MACHINA elaborates on our design that enables the flexibility to switch protocols. In Section PERFORMANCE EVALUATION, we evaluate the performance of our tool using the different synchronization protocols, and we also compare with *adevs*'s performance. We continue by introducing runtime switching of synchronization protocols and different options for model allocation in Section RUNTIME SWITCHING and Section MODEL ALLOCATION, respectively. Related work is discussed in Section RELATED WORK. Section CONCLUSIONS AND FUTURE WORK concludes the paper and presents ideas for future work.

Background

This section briefly introduces the two most prominent synchronization protocols: conservative and optimistic synchronization. Both algorithms are supported by dxex. These protocols synchronize between kernels, but do not specify the parallelization of events within a single kernel. For example, pSim is orthogonal to this as it decides upon the parallelization of events within a single kernel.

Conservative Synchronization

The first synchronization protocol we introduce is *conservative synchronization* [5]. 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 the 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 base protocol, as well as resolution schemes to handle or avoid the deadlocks [5].

The main advantage of conservative synchronization is its low overhead if lookahead is high. Each node then simulates in parallel, notifying 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 throughout the complete simulation run. The user is required to define the lookahead, using knowledge about the model's behaviour. Defining an accurate and high lookahead is far from a trivial task if there is no detailed knowledge of the model. Even slight changes in the model or its allocation can change the lookahead, and can therefore have a significant influence on simulation performance.

Optimistic Synchronization

A completely different synchronization protocol is *optimistic synchronization* [11]. Whereas conservative synchronization prevents causality errors at all costs, optimistic synchronization allows them to happen, but corrects them as soon as they are detected. Each node simulates as fast as possible, independent of other nodes. It assumes that no events occur from other nodes, unless it has explicitly received one at that time. When this assumption is violated (*i.e.*, an event arrives that should have been processed in the past), 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 point in time before the event has to be processed, the event can be processed as if no causality error ever occurred.

Rolling back simulation time requires the node to store past model states, so 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 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 on performance.

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) [5], has progressed beyond their execution time.

The main advantage is that a small lookahead, caused by infrequent events, does not limit performance. 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. Even the occurence of rollbacks is non-deterministic, as it is caused by the interleaving of different simulation nodes and their communication. If rollbacks occur frequently, state saving and rollback overhead can cause simulation to grind to a halt. Nonetheless, it can be proven that simulation always progresses, and eventually always terminates. Apart from overhead in CPU time, a significant memory overhead is present: intermediate states are stored up to a point where they can be considered *irreversible*. Note that, while optimistic synchronization does not explicitly depends on lookahead, performance still implicitly depends on lookahead. Instead of depending on the theoretically defined safe lookahead, performance is related to the actually perceived lookahead.

DEVS-Ex-Machina

Historically, *dxex* is based on *PythonPDEVS* [9]. Python is a good language for prototypes, but performance has proven insufficient to compete with C++-based simulation kernels [12]. *Dxex* is a C++11-based Parallel DEVS simulator, based on the design of *PythonPDEVS*. Whereas the feature set is not too comparable, the architectural design, simulation algorithms, 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 [13], Dynamic Structure DEVS [14], termination conditions [15], and a modular tracing and scheduling framework [9]. We do not elaborate on these features in this paper. But whereas *PythonPDEVS* only supports optimistic synchronization, *dxex* support multiple synchronization protocols (though not distributed). This is in line with the design principle of *PythonPDEVS*: allow users to pass performance hints [16] to the simulation kernel. In our case, a user can configure the simulation kernel with the synchronization protocol to use, or even switch the synchronization protocol during runtime. Our implementation in C++11 also allows for (compiled) optimizations which were plainly impossible in an interpreted language, such as Python. *Dxex* uses new optimizations from the C++14 standard when possible. The C++11 standard threading primitives are used to run the different simulation kernels. Within a single simulation kernel, OpenMP [17] is used to parallelize the transition functions, as is usual in Parallel DEVS.

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 elaborate on our prominent new feature: the efficient implementation of multiple synchronization protocols within the same simulation tool, offered transparently to the model.

Synchronization protocols

We previously explained the existence of different synchronization protocols, each optimized for a specific kind of model. As no single synchronization protocol is ideal for all domains, a general purpose simulation tool should support multiple protocols. We argue that a general purpose simulation tool should support all six standard synchronization protocols, as is the case for *dxex*.

Different protocols relate to different model characteristics. For example, *Cx* is for when high lookahead exists between different nodes, whereas *Ox* is for when lookahead is unpredictable. It is possible for the synchronization overhead to become larger than the achieved parallelism, resulting in slower simulation than fully sequential execution (*NN*).

Data exchange between different simulation kernels happens through shared memory, using the new C++11 synchronization primitives. This was also possible in previous versions of the C++ standard by falling back to non-portable C functions. C++11 further allows us to make the implementation portable, as well as more efficient: the compiler makes further optimizations to heavily used components.

Inter-Kernel Synchronization Our core 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.

An overview of *dxex*'s design is given in Figure 1. It shows that there is a simulation `Kernel`, which simulates the `AtomicModel`s connected to it. The superclass `Kernel` represents a standalone simulation kernel (*Nx*). Subclasses define specific variants, such as `ConservativeKernel` (conservative synchronization), `OptimisticKernel` (optimistic synchronization), and `DynamicKernel` (Dynamic Structure DEVS). In theory, more synchronization protocols (*e.g.*, other algorithms for conservative synchronization) can be added without altering our design.

The following inter-kernel synchronization protocols are implemented.

None (Nx) No inter-kernel synchronization is the base case, implemented in the `Kernel`. It can be overloaded by any of the other simulation kernels, which augment it with inter-kernel synchronization methods.

Conservative (Cx) For conservative synchronization, each kernel determines the kernels 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.

Optimistic (Ox) For optimistic synchronization, each node must be able to roll back to a previous point in time. This is implemented with 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 [18] 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 (*e.g.*, printing) 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 memory pools. Again, we made use of specific new features of C++11, that are not available in Python, or even previous versions of the C++ language standard.

Intra-Kernel Synchronization In our tool, each simulation kernel is capable of executing concurrent transitions in parallel, whether they are external, internal, or confluent.

None (xN) No intra-kernel parallelism is used, meaning that all concurrent transitions are processed sequentially. Note that the order in which they are processed is non-deterministic.

pSim (xP) With intra-kernel parallelism, a configurable number of threads is allocated to optimally divide the processing load of concurrent transitions. The parallel execution of transitions introduces some overhead in thread pooling and locking, and disallows some optimizations in optimally rescheduling models. The concurrency of events and the combined computational load of the transitions has to outweigh this overhead to obtain a significant speedup, as we will demonstrate in Section PERFORMANCE EVALUATION.

Synchronization Protocol Transparency

We define synchronization protocol transparency as having a single model, which can be executed on any kind of kernel, without any modifications. User should thus only provide one model, implemented in C++11, which can be simulated using any of the six synchronization protocols. The synchronization protocol to use is a simple configuration option. The exception is conservative synchronization (*Cx*), 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 should the user never want to use conservative synchronization. Defining the lookahead is difficult and can often be unpredictable. 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 *Nx*.

Defining a lookahead function is therefore recommended in combination with conservative synchronization, but it is not a necessity, as a default value ϵ (*i.e.*,

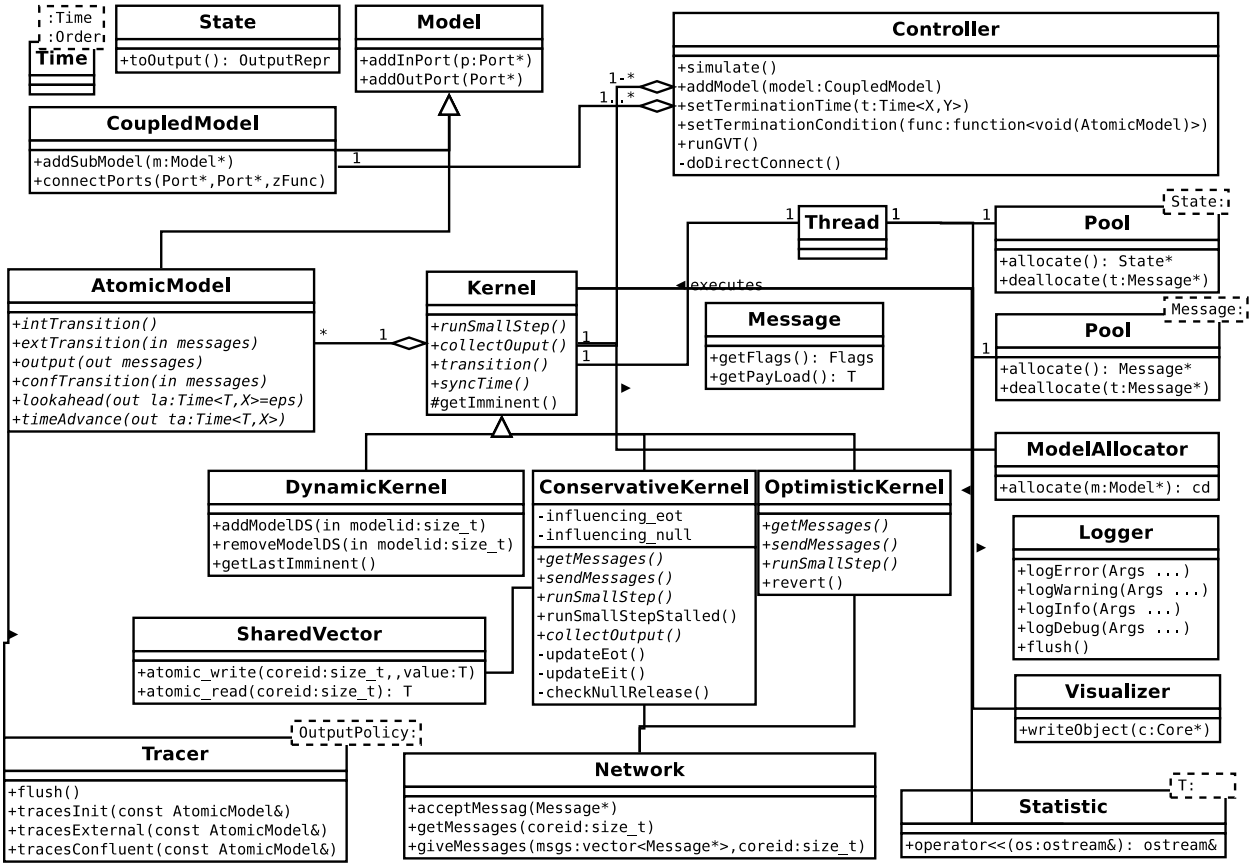

Figure 1. *Dxex* design.

the smallest representable timestep) is used otherwise. Providing this default implementation has no impact in sequential or optimistic simulation; as the function is never called, the compiler will optimize it out. By providing this default implementation in the model base class we ensure that a model can run with any synchronization protocol.

Increasing Parallelism

The goal of our contribution is to increase simulation performance as much as possible, leveraging parallel computation in the process. Parallelizing goes further, however, than merely implementing the different synchronization protocols.

We observed that after implementing all synchronization protocols, performance was still not within acceptable levels. Profiling revealed that most of the overhead was caused by two issues: memory management and random number generation. For both, it is already known that they can have significantly impact on parallelizability of code, since they introduce sequential blocks. Both were tackled using approaches that are in common use in the parallel programming world. We briefly mention how the application of these techniques influences our performance.

Memory Management Memory management is traditionally seen as one of the major bottlenecks in parallel computation [19], since memory bandwidth doesn't increase as fast as the number of CPU cores using it.

While this is always a problem, it is aggravated in *dxex* by providing automatic memory management for events and states. A model written for *Nx* synchronization will run correctly with conservative (*Cx*) or optimistic (*Ox*) synchronization without altering, from the point of view of the modeller, the (de)allocation semantics of events or states.

Furthermore, allocating and deallocating memory by making calls to the operating system, as is typically done by calls such as `malloc`, happens sequentially. To counter this, our memory allocators are backed by a thread-aware pooling library. With *Nx* synchronization, no allocated event persists beyond a single time advance, even allowing the use of an arena-style allocator. Conservative and optimistic simulation need to use generic pool allocators since events are shared across kernels and thus have a different lifetime.

Intra-kernel events can be (de)allocated without synchronization with the other kernels. They can be returned immediately to the memory allocator as the lifetime of these objects is known at creation. In contrast, inter-kernel events need a GVT algorithm to determine when safe deallocation can occur. Intra-kernel synchronization protocols therefore have a lower overhead than inter-kernel synchronization protocols. Simulations with many inter-kernel events suffer a performance hit, whereas the impact of many intra-kernel events can be minimized using arena allocators [20].

Dxex uses *Boost Pool* [21] allocators for *Cx* and *Ox* synchronization, and arena-style allocators for *Nx* synchronization. The latter can be faster, but at the cost of additional configuration. The allocators are supplemented by the library *tcmalloc* [22], reducing lock contention in `malloc` calls.

We primarily investigate this for optimistic simulation, as this is the most memory consuming mode of simulation [5]. Simulation execution times for all four combinations are shown in Figure 2. Optimistic simulation greatly benefits from the use of *tcmalloc*, regardless of the allocator. Nonetheless the pool allocator also reduces the allocation overhead, though only by a relatively small fraction. Both techniques are required to reduce the overhead of memory allocations in *dxex*, and are turned on by default.

Both pools and *tcmalloc* try to keep memory allocated instead of returning it to the Operating System (OS). As a result, the OS will usually report memory consumption that is higher than the actual amount of stored data.

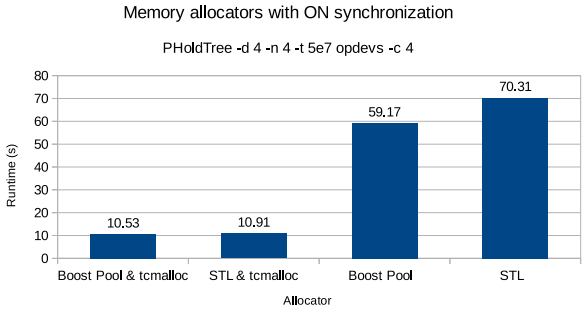

Figure 2. Effect of memory allocators with *ON* synchronization.

Random Number Generators Random Number Generators (RNG) are another aspect of the program that can limit parallelization. All accesses to the RNG will result in the modification of a global (*i.e.*, shared between threads) variable. This easily becomes a bottleneck in simulation, since random numbers are a common occurrence in simulation [23]. As such, a nontrivial amount of time in a simulation is often spent waiting for an RNG.

We still need to guarantee determinism and isolation between the calls to the RNG, as well as avoiding excessive synchronization. *Dxex* uses the Tina RNG collection (TRNG) [24] as an alternative random number generator with performance and multithreading in mind. Since the RNG is an implicit part of the state in the Parallel DEVS formalism, though often not implemented as such, we evaluated performance for both approaches: one global RNG per thread, and one RNG per atomic DEVS model.

We see in Figure 3 that storing the RNG in the state is very expensive for the default Standard Template Library (STL) random number generator with optimistic synchronization. This is primarily caused by the significant difference in size: 2504 bytes for the STL random number generator, and 24 bytes for the Tina random number generator. Only *Ox* synchronization seems affected, as it needs to copy more bytes in every transition due to state saving. No additional copies need to happen in *Nx* or *Cx* synchronization.

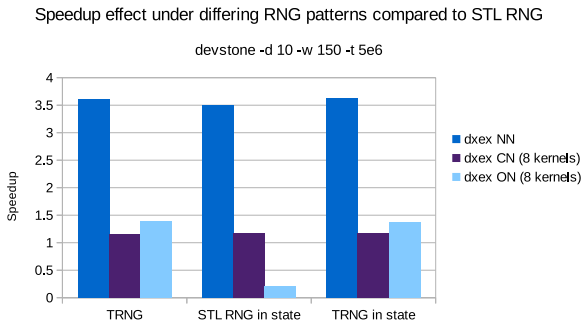

Figure 3. Speedup with different RNG usage patterns compared to STL random number generator.

Figure 3 shows that, for *NN* synchronization, any of the three alternatives is three times faster than STL RNG. For *Cx* and *Ox* synchronization, the synchronization overhead seems to be the main bottleneck, as seen by the big speedup gap with *Nx* synchronization. *Cx* synchronization is almost unaffected by changing the RNG.

Performance Evaluation

In this section, we evaluate the performance of different synchronization protocols in *dxex*. We also compare to *adevs* [10], currently one of the most efficient simulation tools [25, 26], to show that our modularity does not impede performance. CPU time and memory usage is compared for different synchronization protocols.

We start with a comparison of *NN* synchronization, to show how *adevs* and *dxex* relate in this simple case. Afterwards, we compare inter-kernel synchronization protocols, including a comparison with *adevs* again. Inter-kernel synchronization protocols are then compared in combination with intra-kernel synchronization protocols, in the context of recent theoretical work [27].

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 our repository.

The benchmarks are ran on a machine with 8 x AMD Opteron(TM) Processor 6274 with 8 cores per CPU (for a total of 64 cores) and 192 GB RAM.

Benchmark Models

We use three different benchmark models, covering different aspects of the simulation tool.

First, the *Queue* model, based on the *HI* model of DEVStone [28], creates a chain of hierarchically nested atomic DEVS models. A single generator pushes events into the queue, which get consumed by the processors after a fixed or random delay. It takes two parameters: the width and depth of the hierarchy. This benchmark reveals the complexity of the simulation kernel for an increasing amount of atomic models and an increasingly deep hierarchy. An example for a width and depth of 2 is shown in Figure 4.

Second, the *Interconnect* model, a merge of PHold [29] and the *HI* model of DEVStone [28], 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 atomic model receives each generated event (*i.e.*, the event is broadcasted). The model takes one parameter: the number of atomic models. This benchmark shows the complexity of event creation, event routing, and simultaneous event handling. An example for three atomic models is shown in Figure 5.

Third, the *PHold* model [29], creates n atomic models, where each atomic 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 within the same kernel, or outside of the kernel. Afterwards, a uniform selection is made between the possible atomic models. The model takes two parameters: the percentage of remote events (determining the percentage of events routed to other kernels), and the percentage of high-priority events. High-priority events are events generated in a very short time after the previous event. This benchmark shows how the simulation kernel behaves in the presence of

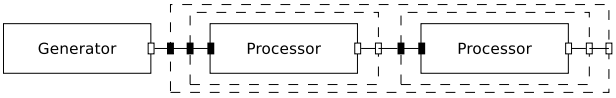

Figure 4. Queue model for depth and width 2.

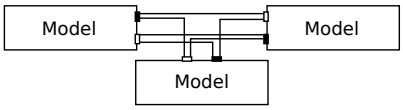

Figure 5. Interconnect model for three models.

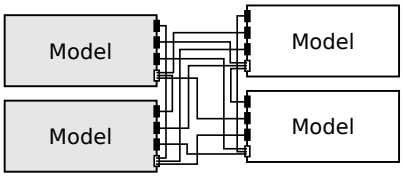

Figure 6. PHold model for four models. Color denotes the two nodes.

many local or remote events, in combination with a varying percentage of high-priority events. An example for four atomic models, split over two kernels, is shown in Figure 6.

Single kernel (NN Synchronization)

We start by evaluating *NN* synchronization performance, in order to obtain a baseline for our comparison of other synchronization protocols.

Queue For the first benchmark, we tested the effect of hierarchical complexity of the model in the performance of the simulator. A set of three tests was performed, where each test has the same number of atomic models but an increasing depth. The results can be seen in Figure 7. Since *dxex* performs direct connection [13] on the model, there is no performance hit when the depth is increased. Direct connection only needs to be done at initialization, making it a neglible one time cost for long running simulations. *Adevs*, on the other hand, suffers from the increased depth, even though some similar (but not identical) optimization to event passing was made [30]. With every new hierarchical layer, routing an event from one atomic model to the next becomes more expensive, resulting in an increase in runtime.

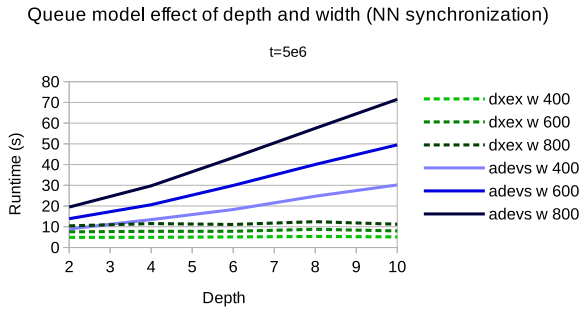

Figure 7. Queue benchmark results with *NN* synchronization.

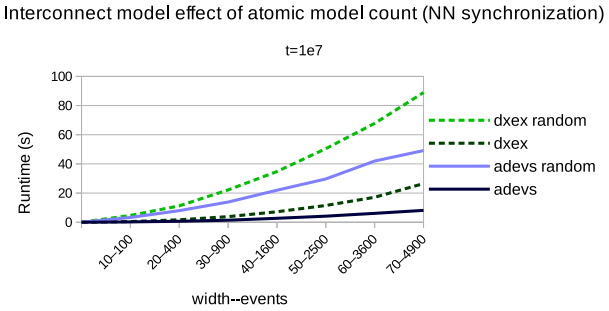

Figure 8. Interconnect benchmark results with *NN* synchronization.

Interconnect In the Interconnect model, we increase the number of atomic models, quadratically increasing the number of couplings and the number of external transitions. As shown in Figure 8, *adevs* now outperforms *dxex* by a fair margin. Analysis showed that this is caused by the high amount of events: event creation is much slower in *dxex* than it is in *adevs*, despite *dxex*'s use of memory pools. To shield the user from threading and deallocation concerns, *dxex* provides an event superclass from which the user can derive to create a specialized event type. Copying, deallocation, and tracing are done at runtime, adding significant overhead when events happen frequently. Profiling the benchmarks revealed the increased cost of output generation and deallocation as the determining factor.

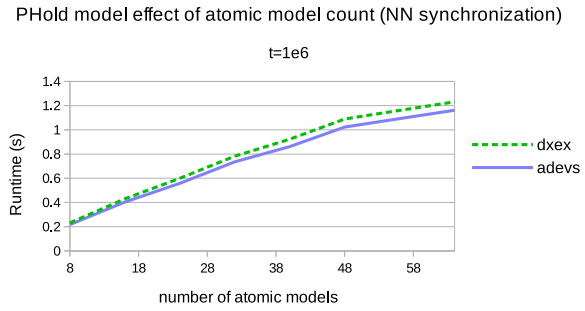

Figure 9. PHold benchmark results with *NN* synchronization.

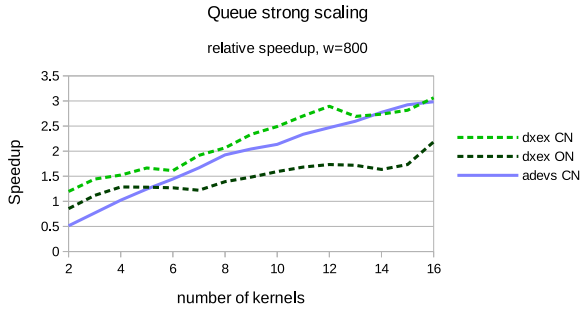

Figure 10. Queue model strong scaling speedup compared to *dxex NN* synchronization. Each kernel uses one thread, and has one physical CPU core allocated to it.

PHold The PHold model is very similar to the Interconnect model. The biggest difference is that the amount of messages sent is much lower. The number of events scales linear with the number of atomic models, not quadratic. Figure 9 shows that in terms of performance *dxex* and *adevs* are very close to each other, with *adevs* slightly outperforming *dxex*.

Inter-Kernel Parallelism (CN and ON synchronization)

We now continue by describing our inter-kernel parallelims performance for different synchronization protocols, compared to *adevs*. The speedup of *adevs* is computed with the corresponding *dxex NN* synchronization benchmark. This was done to take into account the performance difference observed in *NN* synchronization. As such, the highest speedup indicates the fastest results among all tools, independent of *NN* synchronization results for *adevs*. We only compare *xN* results, as we are now only interested in the differences between the various inter-kernel synchronization protocols, and have thus disabled all other forms of parallelism. These results are generalized to conservative and optimistic synchronization in general. The comparison between *xN* and *xP* is made later on.

Queue The Queue model is one single chain of atomic models, resembling a pipeline. This structure can be exploited to prevent cyclic dependencies in *CN* and *ON* synchronization.

Figure 10 shows the speedup compared to *NN* synchronization for a fixed problem size, with an increasing number of used CPU cores (*i.e.*, strong scaling). As the number of cores increases, *ON* quickly becomes the worst choice. This is mainly caused by the pipeline structure of the model: the last atomic models in the queue only respond to incoming messages and therefore have to be rolled back frequently. The difference between *dxex CN* and *adevs CN* becomes smaller when more and more cores are used. The same effect can be seen when the problem size is increased in tandem with the number of used CPU cores (*i.e.*, weak scaling) in Figure 11.

Interconnect In the Interconnect model, we determine how broadcast communication is supported across kernels. The number of atomic models is now kept constant at eight. Results are shown in Figure 12. When the number of kernels increases, performance decreases due to increasing contention in conservative simulation and the increasing number of rollbacks in optimistic simulation. All atomic models depend on each other and have no computational load whatsoever, negating any possible performance gain by splitting up the work over multiple kernels.

PHold In the PHold model, we first investigate the influence of the percentage of remote events on the speedup. A remote event in this context is an event that is sent from an atomic model on one kernel to an

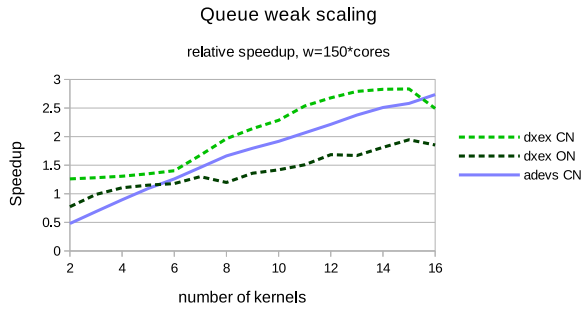

Figure 11. Queue model weak scaling speedup compared to *dxex NN* synchronization. Each kernel uses one thread, and has one physical CPU core allocated to it.

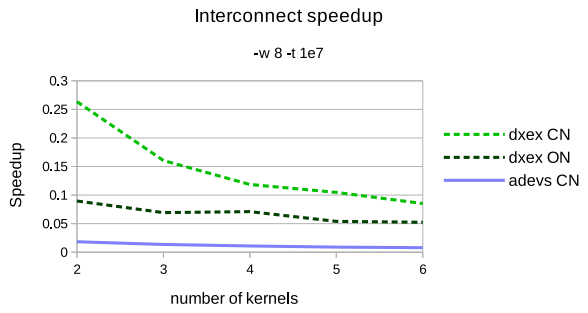

Figure 12. Interconnect benchmark results for *ON* and *CN* synchronization, compared to *dxex NN* synchronization. Each kernel uses one thread, and has one physical CPU core allocated to it.

atomic model on another kernel. When remote events are rare, optimistic synchronization rarely has to roll back, thus increasing performance. With more frequent 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 and wait. Even though a single synchronization protocol is ideal throughout the whole simulation run, it shows that different synchronization protocols respond very differently to a changing model.

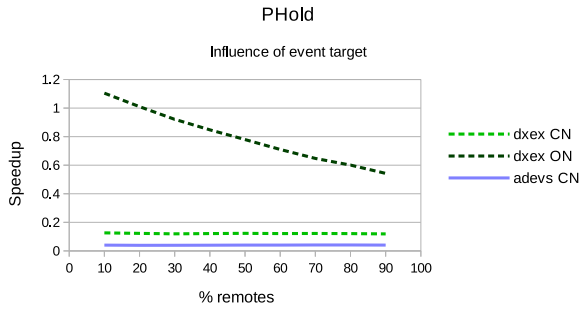

Figure 13. PHold benchmark results using four kernels, with varying percentage of remote events.

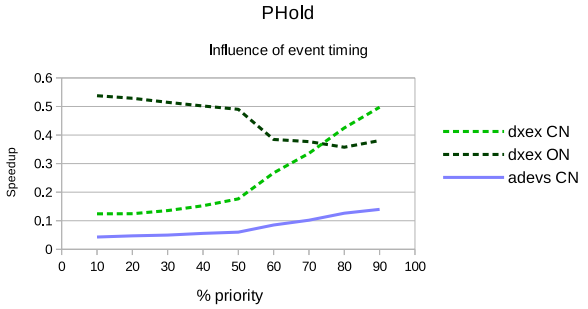

Figure 14. PHold benchmark results using four kernels, with varying amount of high-priority events.

Adevs is significantly slower for *CN* synchronization. Analysis of profiling callgraphs 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. These functions require the user to implement a state saving method. But in contrast to *PythonPDEVS* and *dxex*, which handle this inside the kernel, users need to manually define this. We feel this would lead to an unfair comparison as we would like to keep the models agnostic of the underlying protocols across all benchmarks.

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 forward in simulation time and rolls back when these high-priority events happen. This situation closely mimics the model typically used for comparing conservative and optimistic synchronization [5].

Figure 14 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 unconditional progression of simulation, which frequently needs to be rolled back. Results show that there is no single perfect synchronization algorithm for this kind of model: depending on configuration, either synchronization protocol might be better.

Intra-Kernel Parallelism (xP synchronization)

The abstract simulator of Parallel DEVS included its own notion of parallelism, which we refered to as pSim, and have implemented as *xP* synchronization. The *xP* synchronization protocol is trivial, as it merely executes a set of independent functions concurrently. As indicated before, intra-kernel synchronization is independent of inter-kernel synchronization in *dxex*. That is, all six combinations are possible. Following the theoretical analysis published in [27], a comparison is warranted between *xN* and *xP* synchronization.

Model We have opted to use the *Queue* model for this comparison, as it allows for many interesting configurations. We have three different configurations, each using 16 threads in total, to be ran on a CPU with 16 CPU cores: *NP* synchronization uses one kernel with 16 threads each; *OP* and *CP* synchronization use 4 kernels with 4 threads each; and *ON* and *CN* synchronization use 16 kernels with 1 thread each. Additionaly, *NN* synchronization only uses one kernel and one thread, as it is completely sequential. All speedup results are shown in comparison to *NN* synchronization. Atomic models are always equally distributed over the available kernels.

This allows us to observe which is more efficient in obtaining a speedup, each with the same number of CPU cores available. We simulate a computational load by a sleep of 5 milliseconds. The model is configured with depth 4 and width 300 if the transition function has no load (*i.e.*, no sleep), and width 50 if the computational load is active (*i.e.*, sleeps for 5 milliseconds). In our configuration, an imminent atomic model always generates output, resulting in the receiving atomic model becoming imminent. The probability that an internal transition in an atomic model generates output and is connected to a receiving atomic model is 1 ($q = 1$ [27]). The probability that an atomic model becomes immediately imminent depends on whether fixed or random time advance is used (p [27]). In the case of a fixed time advance, a model will always become immediately imminent ($p = 1$). When random time advance is used, this probability is $\frac{1}{n}$, with n denoting the total amount of atomic models, as only one atomic model becomes active ($p = \frac{1}{n}$).

The benchmarks are run sufficiently long enough to guarantee that the frequency of internal and external transitions is equal within a benchmark, regardless of the randomness of the time advance. In our model each atomic model executes internal and external transitions, creating an ideal use case to evaluate the speedup *xP* synchronization can obtain. The key difference is that although the event frequency is the same, their concurrency is not. The coefficient of variation of our results is less than 1%. The communication overhead is hard to estimate, but given our coefficient of variation, we can expect this overhead to be constant.

We consider two cases: one where all transition functions happen simultaneously, and one where transition functions never happen simultaneously. We defer the discussion on which of these two is the most realistic, as this depends on the problem domain. For example, in a simulation with a fixed timestep (*e.g.*, cell-based models, discretization of continuous model), transition functions often occur simultaneously. Conversely, simulations with an arbitrary timestep (*e.g.*, many independent systems communicating together) have very few simultaneous events.

Concurrent events ($q = 1; p = 1$) First we create a model where all transitions happen simultaneously, with a significant computational load in the transition functions. In Figure 15, we observe that all synchronization protocols result in a speedup of about 10. In this scenario

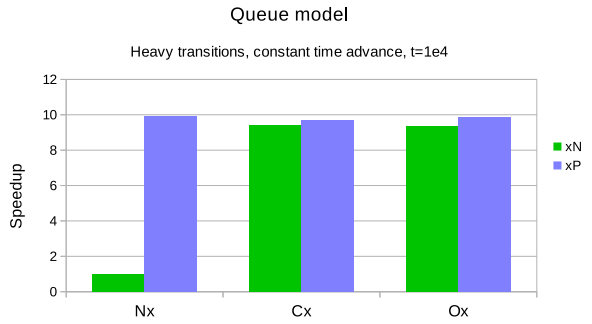

Figure 15. Queue speedup benchmark with $p = 1$, $q = 1$, and significant computational load

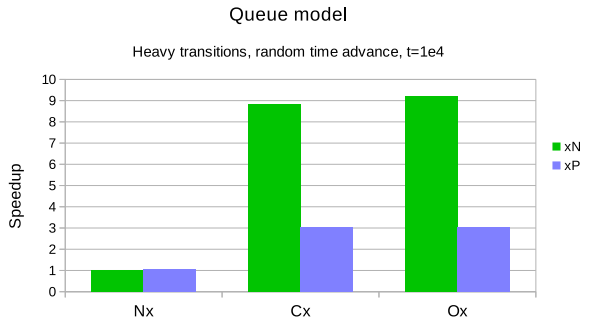

Figure 16. Queue speedup benchmark with $p = \frac{1}{n}$, $q = 1$, and significant computational load

there is no real advantage between the different parallel configurations. Note, however, that *NP* synchronization is trivial to implement, whereas *Ox* and *Cx* synchronization are much more difficult to implement.

Random events ($q = 1; p = \frac{1}{n}$) Now we randomize the time advance in the atomic models, resulting in very few transition functions that happen simultaneously. Even when two transitions are only minimally apart in simulated time, they cannot be executed in parallel, as there might otherwise be a causality error. The transition function has the same computational load as in the previous configuration. Results are shown in Figure 16. We observe that *NP* synchronization adds little overhead, but doesn't achieve any significant speedup either. With *ON* and *CN* synchronization, we again achieve high speedups. This is not the case with *OP* and *CP* synchronization, as we only have four kernels available for inter-kernel synchronization. The four threads per kernel, in this case, are not used fully as only two transitions occur simultaneously at all times.

Computational load Finally, we remove the computation load from the transition function, in combination with many concurrent events. Figure 17 shows that for *NP* synchronization, the overhead of thread management and shared memory communication is crippling for performance. Even though many events occur concurrently, the computation in the transition function does not outweigh the overhead. This results in much slower simulation than *NN* synchronization. Even *OP* and *CP* synchronization are unable to achieve any performance

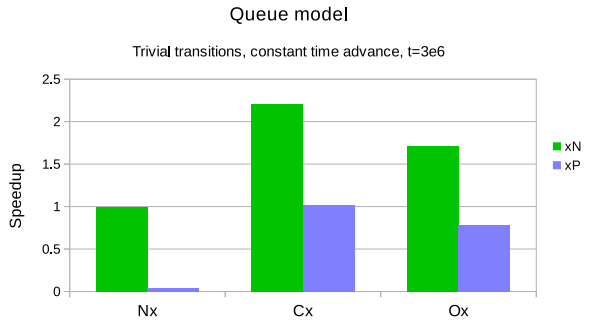

Figure 17. Queue speedup benchmark with $p = 1$, $q = 1$, and trivial computational load

increase, whereas *ON* and *CN* synchronization increase performance marginally. This is because the overhead of thread management is avoided, similar to the results obtained by Himmelspach *et al.* [4]

Discussion In *dxex*, any inter-kernel synchronization protocol can be combined with any intra-kernel synchronization protocol. While *xP* synchronization can offer a significant speedup at a trivial implementation cost, p must be high throughout the whole simulation. Additionally, the computational load of the transition functions should be high enough to warrant the thread management overhead. We conclude that each synchronization protocol has its distinct advantages and disadvantages: inter-kernel synchronization protocols depend on the coupling

of models and are difficult to implement, whereas intra-kernel synchronization protocols depend on the number of concurrent transitions and are trivial to implement.

Memory Usage

Apart from simulation execution time, memory usage during simulation is also of great importance. Having insufficient memory may cause sudden deterioration in performance, even to the point of making the simulation infeasible. We therefore also investigate the memory usage of different synchronization protocols, again comparing 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 distributing the state over multiple machines, which neither *dxex* nor *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 in maximum resident set size as reported by the OS. We only show results for *xN* synchronization, as *xP* synchronization has no significant additional memory requirements for a reasonable number of threads.

Results Figure 18 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). Note the logarithmic scale due to the high memory consumption of optimistic synchronization.

Unsurprisingly, *Ox* 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.

Cx synchronization always uses more memory than *Nx* synchronization, as is to be expected. Additional memory is required for the multiple kernels, but also to store all events that are processed simultaneously.

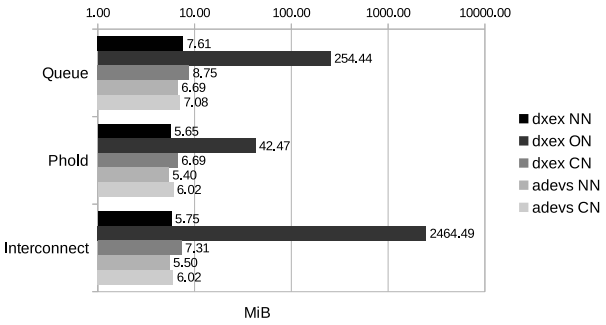

Figure 18. Memory usage results.

Conclusions on Performance Evaluation

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. Each synchronization protocol has its own specific kind of models for which it is the best one. But even with the right synchronization protocol, we have seen that two problems remain.

First, although a synchronization protocols might be ideally suited for specific model behaviour, nothing guarantees that the model will exhibit the same behaviour throughout the simulation. Similarly to the polymorphic scheduler [12], we wish to make it possible for the ideal synchronization protocol to be switched during simulation. When changes to the model behaviour are noticed, the used synchronization protocol can be switched as well.

Second, the allocation of models is nontrivial and has a significant impact on performance. While speedup for the Queue model, for example, was rather high in most cases, this is mostly due to characteristics of the model: the dependency graph does not contain any cycles. When cycles were introduced, as in the Interconnect model, performance became disastrous.

In the next two sections, we elaborate on these two problems.

Runtime Switching

Simply because a synchronization protocol is ideal at the start of the simulation, does not mean that it stays ideal during the simulation. It is well known, and repeated in the previous section, that model behaviour significantly influences the ideal synchronization protocol. Contrary to many modelling formalisms, the DEVS formalisms makes it possible to model basically any kind of discrete event model. As such, it is possible for the model to significantly change its behaviour throughout the simulation.

Defining the ideal synchronization protocol at the start of the simulation, when information about future model behaviour is scarce, might therefore not offer the best possible performance. In *dxex*, we not only make it possible to define the synchronization protocol to use, but also to change this decision throughout simulation. To switch between intra-kernel synchronization protocols, we only have to execute all concurrent transitions sequentially. This case is trivial, as it just flips a boolean, and is thus not further considered in this section. To switch between inter-kernel synchronization protocols, all kernels are notified of the switch and they are forced to stop simulation. When stopped, each kernel instantiates a new kernel, with the new synchronization protocol, that is provided with the simulation state of the previous kernel. Simulation is then resumed with the new kernels after the previous ones are destroyed.

As usual, switching imposes an overhead and should thus only be done if the benefits outweigh the induced overhead. This overhead depends on the size of the model and the number of kernels. For a simple model and a few kernels, the overhead is less than a second. Creating new kernels and moving the simulation models has an overhead linear in the amount of kernels and atomic models. The time required to synchronize and halt the existing kernels is variable, especially if the old synchronization protocol is optimistic since there is no real limit on the virtual time difference between kernels. Given that the existing kernels are equally loaded, this time difference will in practice scale linearly with the number of kernels.

Although we currently only support manual switches between different synchronization protocols, this is not necessarily the case. Ideally, a new component is added to the kernel, which monitors model behaviour and simulation performance, and toggles between them automatically. Our interface is augmented with the necessary bindings for such a decision component. Also, our interface is augmented with an interface for statistics gathering and model behaviour analysis. With all interfaces implemented and tested, we only leave open the actual switching logic. Machine learning is a possible direction for future implementations of this decision component.

Statistics Gathering

Traditionally, models are not exposed to simulation kernel details as they work at a different level of abstraction: modellers only care about simulation results, and not about how these results are obtained (*e.g.*, through parallel or distributed simulation). This is different for a new kernel component that has to monitor the behaviour of not only the model, but the simulator as well.

We add performance metrics in the kernel, which logs relevant performance metrics and processes them for use in other components. These metrics include the number of events created and destroyed, the number of inter- and intra-kernel events, the number of rollbacks, the measured lookahead, details of the GVT and EOT calculations, and information on the fairness between kernels. With all these metrics, the decision component can get a global view on both model and kernel behaviour.

For example, if the actually seen lookahead is significantly higher than the defined lookahead, it might be interesting to switch to optimistic synchronization. When the number of rollbacks is excessively high, switching to conservative synchronization might be considered.

Visualization of Communication To provide more insight in our benchmark models, we created a simple visualization of their simulation trace. This trace visualizes the allocation of the model and all defined connections. For each connection, the number of events transferred is annotated. Examples are shown for the three benchmark models used before: Figure 19, Figure 20, and Figure 21 show traces for the Queue, Interconnect, and PHold models, respectively. Using this information, we notice that the Interconnect benchmark indeed has a lot of inter-kernel events. Despite the similar structure, the

PHold model does not have as many inter-kernel events. These results are relevant information that can be used by the hotswapping component.

Model Allocation

Although the synchronization protocol is one of the defining factors in simulation performance, model allocation has a significant impact on which protocol is ideal. Depending on the model structure, and how it is mapped to the different kernels, it might not even make sense to parallelize at all. Indeed, if the model is allocated such that frequent communication is necessary between kernels, parallelism is naturally reduced. This brings us to the topic of model allocation, as also implemented in *PythonPDEVS* [31]. In this section we focus on the inter kernel synchronization protocols, as intra kernel synchronization is unaffected by the inter kernel topology.

The modeller can specify which kernel a model should be allocated to, should such manual intervention be required. This is handled by the default model allocator. If no preference is given, a simple striping scheme is used but this is often insufficient. By overriding the default allocator, a modeller tunes the allocation scheme for a specific model. This interface can be linked to graph algorithms for automatic allocation scheme generation, for example to avoid cycles in the dependency graph.

Performance Evaluation

To evaluate the influence of model allocation, we define a new model, based on PHold [29]. The model structure resembles a tree: an atomic model can have a set of children, with children being connected to each other recursively.

Unlike the Queue model, the width of the hierarchy is still present in the topology of the atomic models after direct connection. The PHoldTree model allows us to investigate speedup in terms of model allocation, by modifying the depth and width (fanout) model parameters.

The PHoldTree model is similar in structure to models of gossiping in social networks [32]. The lookahead of an atomic node is the minimally allowed ϵ, as is often the case in realistic models (*e.g.*, if it is unknown what the lookahead might be). We demonstrate the importance of allocation by comparing performance of a breadth-first versus a depth-first scheme. Both options are automated ways of allocation that are independent of the model.

PHoldtree, like Queue, is highly hierarchical, but its flattened structure cannot be partitioned into a chain, as was the case in the Queue model. This topology is interesting since it highlights the effects of allocation. First, we evaluate the model with *NN* synchronization to provide a baseline for further results.

No Inter-Kernel Synchronization (NN) Since *adevs* does not use direct connection, we expect a noticeable performance difference between *dxex* and *adevs*. This is shown in Figure 23, where the fanout (n) determines the performance penalty *adevs* suffers compared to *dxex*. Profiling indeed indicates that an increase in width per subtree (n) leads to higher overheads in *adevs* due to the lack of direct connection. *Dxex* uses direct connection, making it independent of fanout. Performance of *dxex* is, for this model, only dependent on the number of models. Slight deviations can still be seen, though, caused by the initialization overhead of direct connection. Both *adevs* and *dxex* scale linearly in the number of atomic models.

Inter-Kernel Synchronization (CN and ON) Next, we run the model using two different allocation schemes, based on Breadth-First Search (BFS) and Depth-First Search (DFS). First, we explain both allocation schemes.

Breadth-first allocation traverses the tree in a breadth-first way, allocating subsequently visited atomic models to the same node. Intuitively, atomic models at the same level in the tree, but not necessarily siblings, are frequently allocated to the same node. Since there is only infrequently some communication between siblings, and even never between different subtrees, this does not sound an intuitive allocation. This allocation strategy is shown in Figure 24.

Depth-first allocation traverses the tree in a depth-first way, allocating subsequently visited atomic models to the same node. Intuitively, subtrees are frequently allocated to the same node, as shown in Figure 25.

Both allocators will try to divide models evenly over kernels. The effects of varying the number of atomic models per kernel are already evaluated in the previous section on scaling. Here we want to highlight the overhead of communication and inter-kernel dependence.

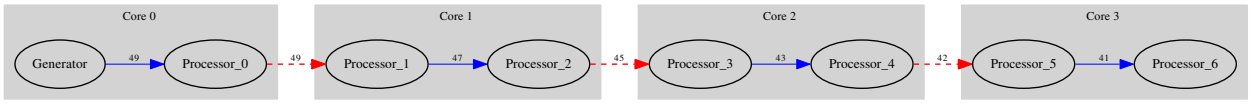

Figure 19. Queue model simulation trace across 4 kernels.

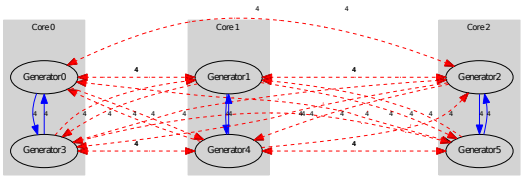

Figure 20. Interconnect simulation trace for 6 atomic models on 3 kernels.

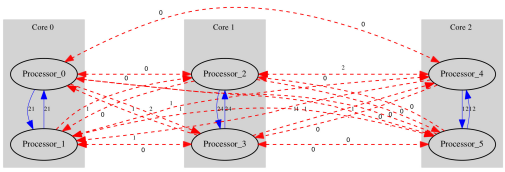

Figure 21. PHold simulation trace for 6 atomic models on 3 kernels.

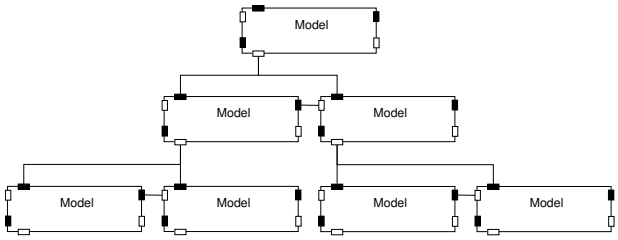

Figure 22. PHoldTree model for depth 3 and width 2.

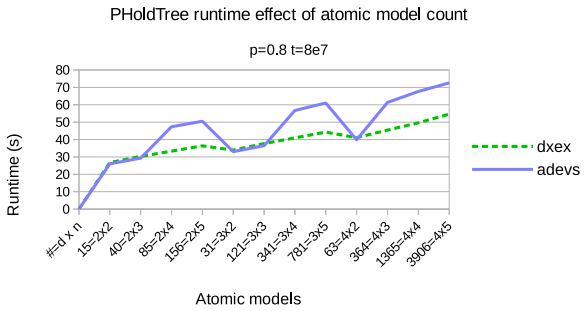

Figure 23. Effect of hierarchy with *NN* synchronization.

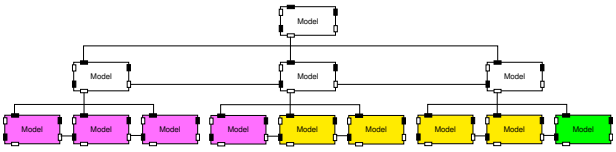

Figure 24. PHoldTree model breadth first allocation with 4 kernels.

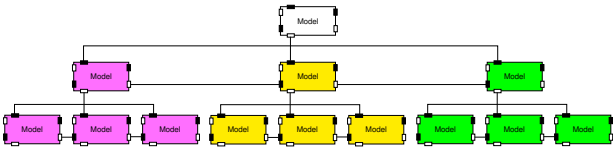

Figure 25. PHoldTree model depth first allocation with 4 kernels.

The breadth first allocation scheme results in a dependency chain with multiple branches, much like in the Queue model. Such a linear dependency chain can result in a higher speedup, as demonstrated with the Queue model. This is not always true though: a single kernel with an unbalanced computational load slows down the remainder of the chain. This effect is also apparent if the thread a kernel runs on is not fairly scheduled by the operating system. With conservative synchronization this leads to excessive polling of the EOT of the other kernels. With optimistic synchronization this leads to a cascade of rollbacks, since dependent kernels will simulate ahead of the slower kernel.

After simulation the traces can be visualized for both breadth-first and depth-first allocation. Using a breadth-first allocation scheme, as shown in Figure 26, we notice that many events get exchanged between kernels. This is caused by the high number of inter-kernel connections and the high number of events exchanged over these connections. The number of connections between atomic models at the same simulation kernel is also rather low. Using a depth-first allocation scheme, however, as shown

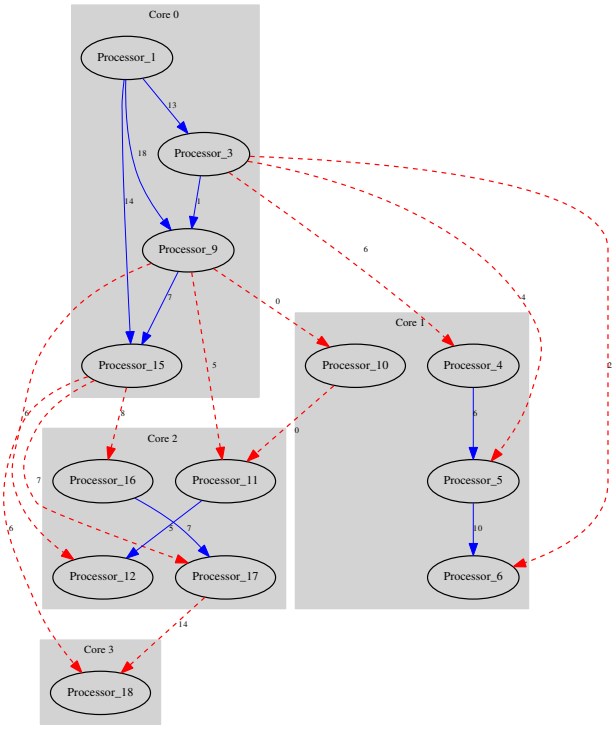

Figure 26. Visualization of a simulation of the model in Figure 24.

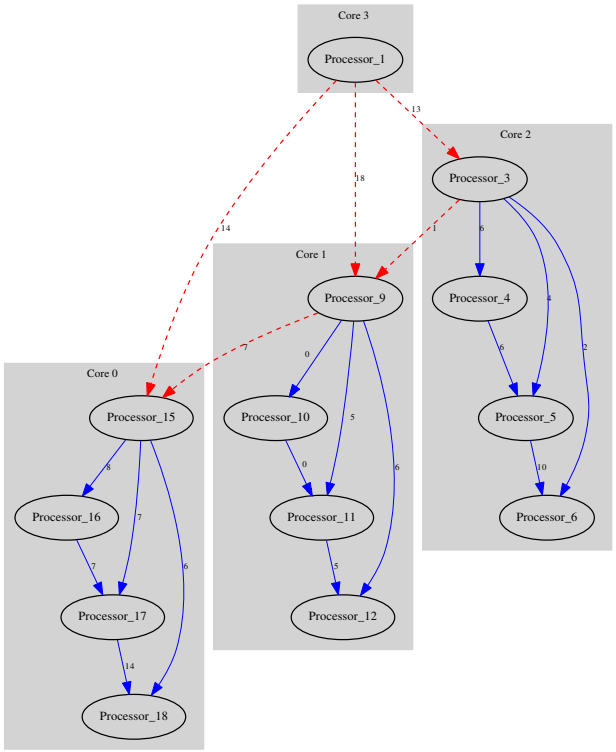

Figure 27. Visualization of a simulation of the model in Figure 25.

in Figure 27, minimizes inter-kernel connections while maximizing intra-kernel connections.

Simulation results are shown in Figure 28 for both allocation schemes in combination with both synchronization protocols. We see that for both synchronization protocols, the depth-first allocation is significantly better than breadth-first allocation. This is what we expected for this model: depth-first allocation preserves locality better than breadth-first allocation. Whereas this is the case in this example, this is not true in general, as the ideal allocation depends on the model being simulated.

The most prominent aspect of these results is the low performance for conservative depth-first allocation for two kernels. This is mostly caused by the introduction of synchronization protocols: suddenly we need to take into account other kernels and passing around of lookahead values. And since the number of kernels is low, the overhead dominates. Optimistic is less sensitive to the number of atomic models per kernel as it does not need to poll each model for a lookahead, this explains the lower runtime penalty observed for optimistic.

Interestingly, we see that optimistic synchronization is less influenced by the allocation than conservative synchronization. This is likely caused by the lower number of connections to take into account in conservative synchronization. Whereas conservative synchronization needs to take into account even scarcely used connections, optimistic synchronization does not. The same is true in the opposite direction, though, where optimistic synchronization is slower when a good allocation is chosen. Conservative synchronization will then be able to make better estimations, whereas optimistic synchronization does not make estimations.

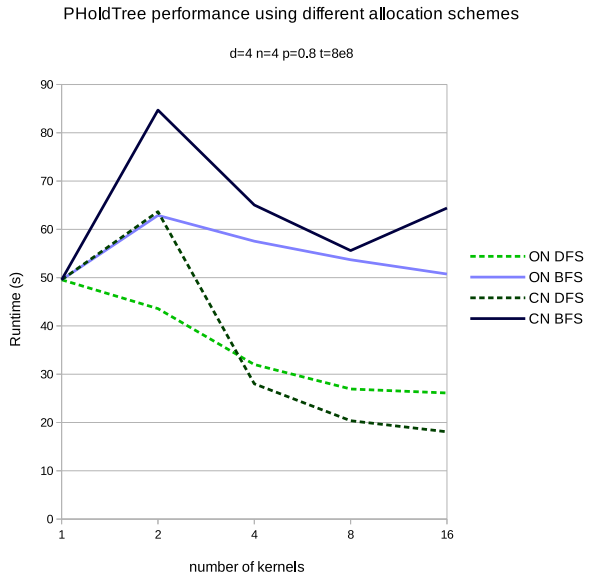

Figure 28. PHoldTree model performance using different allocation schemes.

Related Work

Several similar DEVS simulation tools have already been implemented, though they differ in several 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* [9]. Performance of *PythonPDEVS* was decent through the use of "hints" from the modeler. 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 distributed simulation [15], model migrations [31], or activity hints [16].

Although *PythonPDEVS* offers very fast turnaround cycles, simulation performance was easily outdone by compiled simulation kernels. In terms of performance, *adevs* [10] offered much faster simulation, at the cost of compilation time. While this is beneficial for long running simulations, small simulations are therefore negatively impacted. The turnaround cycle in *adevs* is much slower, 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. Similarly to *vle* [33] and *PowerDEVS* [34], *dxex* compromises 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 shared library. This significantly shortens the turnaround cycle, while still offering good performance. In terms of performance, *dxex* is shown to be competitive with *adevs*. Despite its high performance, *adevs* does not support optimistic synchronization, which we have shown to be highly relevant for certain kinds of models.

Previous DEVS simulation tools have already implemented multiple synchronization protocols, though none have done so in a strictly modular way that allows straightforward protocol switching for a single given model. For example, *adevs* only supports conservative synchronization, and *vle* only supports experiment-level parallelism (*i.e.*, run independent experiments in parallel). Closest to our support for multiple synchronization protocols is *CD++* [35]. For *CD++*, both a conservative (*CCD++* [36]) and optimistic (*PCD++* [37]) variant exist. Despite the implementation of both protocols, they are entirely different projects. Some features might therefore be implemented in *CCD++* and not in *PCD++*, or vice versa. And while this might not be a problem at this time, the problem will only get worse when each project follows its own course. *Dxex*, on the other hand, is a single project, where the choice of synchronization protocol is a simple configuration option. *CD++*, however, implements both conservative and optimistic synchronization for distributed simulation, whereas we limit ourselves to parallel simulation. A new architecture for sequential PDEVS simulation has been introduced in [38] with promising performance results. This algorithm achieves a speedup by using *xP* synchronization. By limiting our approach to pure parallel simulation (*i.e.*, no distributed simulation), we are able to achieve higher speedups through the use of shared memory communication. Recent work on parallel speedup in DEVS investigates theoretical limits of PDEVS [39, 27]

In the PDES community, the problem of choosing between synchronization protocols is well known and documented [40]. The challenges of implementing such runtime switching have previously been explored

already [41], and implemented [42]. Our contribution entails bringing this same feature to the Parallel DEVS community, further expanding upon our support for multiple synchronization protocols.

Model allocation and its impact on parallel performance has previously also been studied in the PDES community [43]. Referenced as partitioning of the simulation model, most studies distinguish between communication and computation as the two dimensions to partition over. Partitioning a model is identified as an issue to achieve scalability [44]. Some research in this context has been done for Parallel DEVS [45, 46]. Our contribution studies the effect of partitioning with emphasis on the effect of communication between kernels and in the presence of a flattened hierarchy. We focus on static partitioning since this is a limiting factor for our conservative synchronization implementation which does not support model migration. Model migration, as implemented by *PythonPDEVS*, might be an interesting addition to model allocation.

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++*. To further profit from our multiple synchronization protocols in a single tool, we further added runtime switching between synchronization protocols and model allocation support.

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 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 any of the six supported synchronization protocols. We distinguish between inter-kernel (none, conservative, and optimistic synchronization) and intra-kernel (sequential or parallel concurrent transitions) synchronization protocols, which are orthogonal to one another. Depending on observed model behaviour and simulation performance, runtime switching between synchronization protocols can be used.

Notwithstanding our modularity, *dxex* achieves performance competitive to *adevs*, another very efficient DEVS simulation tool. Performance is measured both in elapsed time, and memory usage. Our empirical analysis shows that allocation of models over kernels is critical to enable a parallel speedup. Furthermore we have shown when and why optimistic synchronization can outperform conservative and vice versa. We have also shown the influence of using the parallelism inherent in the Parallel DEVS formalism, and its interaction with inter-kernel synchronization algorithms. Finally we investigated the effect of memory (de)allocation on parallel simulation.

Future work is possible in several directions. First, 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 [5]. Faster GVT implementations [47, 48] might further help to minimize this problem. Second, the runtime switching between synchronization protocols can be driven using machine learning techniques. The simulation engine is already capable of collecting data to inform such a process, and is designed to listen for commands from an external component. Third, automatic allocation might be possible by analysis of the connections between models. This information is already used in *dxex* to construct the dependency graph in conservative synchronization. A graph algorithm that distributes models, while avoiding cycles, could be used to offer a parallel speedup in either optimistic or conservative synchronization. Similarly, it could serve as a default parallel allocation scheme that can be improved by the user. Finally, the use of transactional memory can offer several advantages in this project. If it becomes part of the new C++17 standard it would be of great interest to see if it can help reduce the performance effects of memory allocation and synchronization.

ACKNOWLEDGMENTS

This work was partly funded with a PhD fellowship grant from the Research Foundation - Flanders (FWO). Partial support by the Flanders Make strategic research centre for the manufacturing industry is also gratefully acknowledged.

References

[1] Zeigler B, Praehofer H and Kim TG. *Theory of Modeling and Simulation*. 2nd ed. Academic Press, 2000.

[2] Vangheluwe H. DEVS as a common denominator for multi-formalism hybrid systems modelling. In *IEEE International Symposium on Computer-Aided Control System Design*. 2000. pp. 129–134.

[3] Chow ACH and Zeigler B. Parallel DEVS: a parallel, hierarchical, modular, modeling formalism. In *Proceedings of the 1994 Winter Simulation Multiconference*. 1994. pp. 716–722.

[4] Himmelspach J and Uhrmacher A. Sequential processing of PDEVS models. In *Proceedings of the 3rd European Modeling & Simulation Symposium*. 2006. pp. 239–244.

[5] Fujimoto R. *Parallel and Distributed Simulation Systems*. 1st ed. New York, NY, USA: John Wiley & Sons, Inc., 1999.

[6] Kim KH, Seong YR, Kim TG et al. Distributed simulation of hierarchical DEVS models: Hierarchical scheduling locally and time warp globally. *Transactions of the SCS* 1996; 13(3): 135–154.

[7] Jafer S and Wainer G. Conservative vs. optimistic parallel simulation of DEVS and Cell-DEVS: A comparative study. In *Proceedings of the 2010 Summer Simulation Multiconference*. 2010. pp. 342–349.

[8] Cardoen B, Manhaeve S, Tuijn T et al. Performance analysis of a PDEVS simulator supporting multiple synchronization protocols. In *Proceedings of the 2016 Symposium on Theory of Modeling and Simulation - DEVS*. 2016. pp. 614–621.

[9] Van Tendeloo Y and Vangheluwe H. The modular architecture of the Python(P)DEVS simulation kernel. In *Proceedings of the 2014 Symposium on Theory of Modeling and Simulation - DEVS*. 2014. pp. 387–392.

[10] Nutaro J. adevs. **`http://www.ornl.gov/~1qn/adevs/`**, 2015.

[11] Jefferson D. Virtual time. *ACM Transactions on Programming Languages and Systems* 1985; 7(3): 404–425.

[12] Van Tendeloo Y. *Activity-aware DEVS simulation*. Master's Thesis, University of Antwerp, Antwerp, Belgium, 2014.

[13] Chen B and Vangheluwe H. Symbolic flattening of DEVS models. In *Proceedings of the 2010 Summer Simulation Multiconference*. 2010. pp. 209–218.

[14] Barros F. Modeling formalisms for dynamic structure systems. *ACM Transactions on Modeling and Computer Simulation* 1997; 7: 501–515.

[15] Van Tendeloo Y and Vangheluwe H. An overview of PythonPDEVS. In RED CW (ed.) *JDF 2016 – Les Journées DEVS Francophones – Théorie et Applications*. 2016. pp. 59–66.

[16] Van Tendeloo Y and Vangheluwe H. Activity in PythonPDEVS. In *Proceedings of the workshop on Activity-based Modeling and Simulation*. 2014. pp. 2:1–2:10.

[17] OpenMP Architecture Review Board. OpenMP application program interface version 4.5. **`http://www.openmp.org/mp-documents/openmp-4.5.pdf`**, 2015.

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

[19] Drepper U. What every programmer should know about memory. https://lwn.net/Articles/250967/, 2007.

[20] Hanson D. Fast allocation and deallocation of memory based on object lifetimes. *Software: Practice and Experience* 1990; 20(1): 5–12.

[21] Cleary S and Bristow P. Boost Pool: Fast memory pool allocation. **`http://www.boost.org/doc/libs/1_61_0/libs/pool/doc/html/`**, 2011.

[22] Ghemawat S and Menage P. TCMalloc: Thread-caching malloc. **http://goog-perftools.sourceforge.net/doc/tcmalloc.html**, 2005.

[23] L'Ecuyer P. Random numbers for simulation. *Communications of the ACM* 1990; 33(10): 85–97.

[24] Bauke H and Mertens S. Random numbers for large-scale distributed Monte Carlo simulations. *Physical Review E* 2007; 75(6): 066701:1–066701:14.

[25] Wainer G, Glinsky E and Gutierrez-Alcaraz M. tudying performance of devs modeling and simulation environments using the devstone benchmark. *SIMULATION* 2011; 87(7): 555–580.

[26] Van Tendeloo Y and Vangheluwe H. An evaluation of DEVS simulation tools. *SIMULATION* 2016; DOI:10.1177/0037549716678330.

[27] Zeigler B, Nutaro J and Seo C. What's the best possible speedup achievable in distributed simulation: Amdahl's law reconstructed. In *Proceedings of the 2015 Symposium on Theory of Modeling and Simulation - DEVS*. 2015. pp. 189–196.

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

[29] Fujimoto R. Performance of Time Warp under synthetic workloads. In *Proceedings of the SCS Multiconference on Distributed Simulation*. 1990. pp. 23–28.

[30] Muzy A and Nutaro J. Algorithms for efficient implementations of the DEVS & DSDEVS abstract simulators. In *1st Open International Conference on Modeling and Simulation (OICMS)*. 2005. pp. 273–279.

[31] Van Tendeloo Y and Vangheluwe H. Python-PDEVS: a distributed Parallel DEVS simulator. In *Proceedings of the 2015 Symposium on Theory of Modeling and Simulation - DEVS*. 2015. pp. 844–851.

[32] Jelasity M. *Gossip*. Springer Berlin Heidelberg, 2011. pp. 139–162.

[33] Quesnel G, Duboz R, Ramat E et al. VLE: a multimodeling and simulation environment. In *Proceedings of the 2007 Summer Simulation Multiconference*. 2007. pp. 367–374.

[34] Bergero F and Kofman E. PowerDEVS: a tool for hybrid system modeling and real-time simulation. *Simulation* 2011; 87(1-2): 113–132.

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

[36] 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. pp. 443–448.

[37] Troccoli A and Wainer G. Implementing Parallel Cell-DEVS. In *Proceedings of the 36th Annual Symposium on Simulation*. 2003. pp. 273–280.

[38] Vicino D, Niyonkuru D, Wainer G et al. Sequential PDEVS architecture. In *Proceedings of the 2015 Symposium on Theory of Modeling and Simulation - DEVS*. 2015. pp. 165–172.

[39] Capocchi L, Santucci JF and Zeigler B. PDEVS protocol performance prediction using activity patterns with Finite Probabilistic DEVS. In *Proceedings of the 2016 Symposium on Theory of Modeling and Simulation - DEVS*. 2016. pp. 605–613.

[40] Jha V and Bagrodia R. A unified framework for conservative and optimistic distributed simulation. In *Proceedings of the 8th workshop on Parallel and distributed simulation*. 1994. pp. 12–19.

[41] Das S. Adaptive protocols for parallel discrete event simulation. In *Proceedings of the 1996 Winter Simulation Conference*. 1996. pp. 186–193.

[42] Perumalla K. μsik-a micro-kernel for parallel/distributed simulation systems. In *Workshop on*

Principles of Advanced and Distributed Simulation. 2005. pp. 59–68.

[43] Bahulkar K, Wang J, Abu-Ghazaleh N et al. Partitioning on dynamic behavior for parallel discrete event simulation. In *Proceedings of the ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*. 2012. pp. 221–230.

[44] Nicol D. Scalability, locality, partitioning and synchronization PDES. In *Proceedings of the 12th Workshop on Parallel and Distributed Simulation*. 1998. pp. 5–11.

[45] Himmelspach J, Ewald R, Leye S et al. Parallel and distributed simulation of Parallel DEVS models. In *Proceedings of the 2007 Spring Simulation Multiconference - Volume 2*. 2007. pp. 249–256.

[46] Ewald R, Himmelspach J and Uhrmacher A. A non-fragmenting partitioning algorithm for hierarchical models. In *Proceedings of the 2006 Winter Simulation Conference*. 2006. pp. 848–855.

[47] Fujimoto R and Hybinette M. Computing Global Virtual Time in shared-memory multiprocessors. *ACM Transactions on Modeling and Computer Simulation* 1997; 7(4): 425–446.

[48] Bauer D, Yaun G, Carothers C et al. Seven-O'Clock: A new distributed GVT algorithm using network atomic operations. In *Proceedings of the Workshop on Principles of Advanced and Distributed Simulation*. 2005. pp. 39–48.