Performance analysis of a parallel PDEVS simulator handling both conservative and optimistic 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 algorithms have been proposed, each with a specific kind of simulation model in mind. Due to the significant differences between these algorithms, current Parallel DEVS simulation tools restrict themself to only one such algorithm. In this paper, we present a Parallel DEVS simulator, grafted on C++11, which offers both conservative and optimistic simulation. We evaluate the performance gain that can be obtained by choosing the most appropriate synchronization protocol. Our implementation is compared to ADEVS using hardware-level profiling on a spectrum of benchmarks.

1. INTRODUCTION

1.1 DEVS

The family of DEVS [17] formalisms serve as a common basis for most other discrete event formalisms. Of interest in this paper are the 3 key formalisms: Classic [18], Dynamic Structured [1] and Parallel [4] and their implementation. This project uses the DirectConnect [3] algorithm, so from a kernel's perspective only Atomic Models exist in the simulation linked to each other by connected ports.

1.2 Parallel computing

Parallel execution of a PDEVS simulation can lower overall runtime and increase the bound on the state space, thereby enabling simulation of more complex systems in the same time-frame. While the shared memory parallelism offered by most modern hardware does not increase the state space bounds, it can reduce the runtime and offers more direct communication and control between entities involved in synchronization compared to distributed simulation.

1.3 Motivation

Adevs [13] offers a very fast conservative synchronized shared memory DEVS simulator, but no optimistic synchronized variant. The latter can be significantly faster, especially in simulations where the runtime behaviour of the simulation is hard to predict.

The matured parallelism features of C++11 were used in this project with the dual aim of writing standard-compliant (and thus portable) code and without losing access to powerful low-level threading primitives.

1.4 Solution

The usage of the DirectConnect[3] algorithm makes reusing the adevs kernel hard. The dxexmachina kernel follows the design of the PythonPDEVS [16] kernel where appropriate, but by the very nature of the implementation languages has to differ in key aspects (e.g. memory allocation strategy). The core aims of the project are to offer a deterministic simulation kernel where the simulation author is shielded as much as possible from the kernel implementation, without sacrificing performance. As in PyPDEVS, a model need be written only once for use in the different simulation kernels (with the exception of a non-trivial lookahead).

The tracing framework from PyPDEVS was ported to allow optional verification of simulations.

1.5 Time

The PDEVS formalism has $\mathbb{R}$ as time base, but any implementation has to decide on an enumerable representation of time. Dxexmachina kernels can operate on IEEE754 floating point time units or integral time. In principle any type with well defined operators can be used as template parameter, but from a performance point of view a type fitting in a machine word offers obvious advantages.An integral representation significantly reduces the possible range of the virtual time, but avoids approximation errors. Furthermore, the notion of ϵ as the absolute minimum between time points needs to be established, this is non-trivial for floating point.

The select() function, which imposes a sequential ordering between concurrent events, is implicit in this project by extending time representation with a causality field with a range at least as large as the maximum nr of models in the kernels (2^{24} by default). If A and B are imminent at time t then $t[1]_a < t[1]_b \oplus t[1]_b < t[1]_a$, while $t[0]_a == t[0]_b == t[0]$. This avoids evaluating the select() function on all imminent

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

models, while still maintaining the deterministic order of concurrent transitions.

2. BACKGROUND

In this section, we provide a brief introduction to two different synchronization protocols for parallel simulation, and the features offered by C++11 that aid in our implementation.

2.1 Conservative Synchronization

Conservative synchronization is defined by the invariant that no model will advance in time before it has received all input from any influencing model.

This requires the concepts of eot (earliest output time) and eit (earliest input time) which define the timespan within which a model can safely advance.

The eit of any model is the minimum of all eot values of (directly) influencing models. A model can simulate up to (but not including) eit, then waits until that value is increased. An important disadvantage here is that the influenced-by relation is always defined at model(link) creation, not at runtime. A model that can influence another, but never does, can severely slow down the protocol.

Deadlock between models that influence each other and end up waiting on each other can be broken/avoided by a variety of means, in this simulator the CMB [2] null-message protocol is used.

In our implementaton, null-time is the timestamp a model is guaranteed to have passed in simulation. More precisely, a null message of time t is a guarantee that any output with timestamp t-ϵ is already sent.

In general, the eot/eit/nulltime of a kernel is the mimimum of each of those values for all models in the kernel.

Conservative synchronization explicitly relies on information provided by the model creator in the form of lookahead, a relative timespan during which the model is insensitive to outside events. This can be non-trivial to calculate, a simulation writer will in general not be able to predict the exact lookahead of models involved in an experiment without having run the experiment.

Conservative kernels can operate if there exists a cyclic dependency between them, but at a quite severe performance penalty, as seen in section 4.

2.2 Optimistic Synchronization

Optimistic synchronization allows causality errors to occur but recovers from those errors using a roll-back mechanism, the most common of which is Timewarp [10]. Whenever a kernel receives an event with a timestamp in the kernel's past, the state of the kernel (and all models) is reverted to that time. The gain in runtime this provides is offset by the increase in memory required to keep saved states and (sent) messages. Optimistic does not rely on any domain specific information, in contrast to conservative. It is only sensitive to runtime use of connections, not the probability that they might occur.

If the (runtime) dependency graph contains a cycle, optimistic can suffer a series of cascading reverts. Without domain specific information the kernel assumes that any event will influence at least one model, but this can lead to an infinite loop of reverts in the worst case. This effect can be lessened by lazy cancellation and/or lazy re-evaluation [7].

2.3 Global Virtual Time

To avoid exhausting memory in state/event saving, optimistic synchronization relies on the concept of global virtual time[10]. In optimistic simulations, GVT is defined as the lowest timestamp of any unprocessed event.

Intuitively this is the simulation timepoint that is certain to be preserved, corresponding exactly with the simulation up to that time in a non-parallel implementation.

In conservative, the minimum simulation time of all kernels is the GVT, or in terms of null messages: the least timestamp of any null message in transit. The GVT calculation is vital to safely commit unrecoverable transactions such as IO (e.g. tracing), releasing memory, ...

2.4 C++11 Parallelism Features

C++11 offers a wide range of portable synchronization primitives in the Standard Library, whereas in earlier versions one had to resort to non-portable (C) implementations. More importantly, C++11 is the first version of the standard that actually defines a multi-threaded abstract machine memory model in the language. Our kernels use a wide range of threading primitives and atomic operations. As an example, eot/eit/nulltime are exchanged not as messages but reads/writes to atomic fields shared by all kernels. This avoids the otherwise unavoidable latency penalty by mixing simulation messages with synchronization messages, for an in-depth study see [5]. Most modern compilers support the full standard, allowing the kernels to be portable by default on any standard compliant platform.

3. FEATURES

3.1 Based on PythonPDEVS

The simulator is based on PythonPDEVS [16] , and provides the following features:

1. Direct Connect [3]
2. Dynamic Structured DEVS
3. Termination function. If specified, a termination function is applied every simulation round to each model to test whether the simulation can terminate. Only available in single-threaded simulation.
4. State/Message can have any payload type. Different message types can be used together within the same simulation.
5. Tracing An asynchronous, thread safe and versatile tracing mechanism allows exact verification of the simulation.
6. Optimistic and Conservative synchronization of PDEVS.

The implementation tries to adhere to the C++ principle that you don't pay performance-wise for what you don't use. For this reason, the support for a termination function for the multi-threaded kernel was abandoned, as it is non-trivial to implement and had a non-negligible impact on the runtime, even when not in use.

The tracing is not comparable to adevs's listener interface.

To be usable in optimistic simulation, the tracing of the simulation has to be reversible and only be committed at GVT points. Furthermore, the framework itself has to be threadsafe and deterministic so that a simulation will always produce the exact same output. The following features from PythonPDEVS are not present

1. Activity tracking and relocation
2. Serialization
3. Interactive control.
4. Distributed simulation

Serialization in this context is the ability to save/load a complete simulation to disk, not the state saving mechanisms required for TimeWarp.
State saving has no impact in a single threaded or conservative kernel.
Model allocation is done by a derivable allocator object which the user can implement to arrange a more ideal (domain-specific) allocation. If this is omitted, a default (non-activity-aware) allocation stripes the models over the simulation kernels.
Debugging tools such as a logger and a graph visualizer are included which can track activity with respect to allocation for later study, but not online/dynamic as is possible in PyPDEVS.
While PythonPDEVS can be controlled from within a Python script and adevs has a Java interface, our implementation does not have any bindings to other languages.

3.2 Different Synchronization protocols

Conservative

A conservative kernel will determine which kernels it is influenced by. This information is constructed from the incoming connections on all hosted models. The process is only 1 link deep, since an influenced kernel will in turn be blocked by others deeper in the graph.
A model should provide a lookahead function which returns, relative from the current time, the timespan during which the model cannot change state due to an external event. This information is collected for all models hosted on the kernel, and the minimum is set as the lookahead of that kernel.
The kernel will calculate its earliest output time and write this value in shared memory. The eit of the kernel is then set as the minimal eot of all influencing kernels.
For garbage collection (of sent messages) the LBTS/GVT is calculated as $\min_{\forall i \in \text{influencors}}(\text{nulltime}[i]) - \epsilon$.

Optimistic

The optimistic kernel requires from the hosted model only that copying the state is well-defined, which is provided in the base State class for the user. The kernels use Mattern's [11] GVT algorithm with a maximum of 2 rounds per iteration to determine a GVT. This process runs asynchronously from the simulation itself. Once found, the controlling thread informs all kernels of the new value, which they can use to execute garbage collection of old states/(anti)messages.

The user need only provide one implementation of a model for use with both synchronization protocols. A lookahead function is desired to accelerate conservative, but is not required. In the absence of a user supplied lookahead, the kernel assumes it cannot predict beyond its current time + ϵ, creating a lockstep simulation.
The implementation details such as defining the copy semantics of a State are provided (but can be overridden).
From the user's perspective, the multi-threaded aspect of the kernel is not exposed.

3.3 Performance Improvements

Continuous profiling of the kernels in several benchmarks highlighted the following key bottlenecks:

Heap

A kernel never sends a complete object to another kernel, only a pointer to the object. This avoids a possibly expensive copy of the payload, but at the cost of allocation overhead.
This cost becomes prohibitively expensive in highly connected models, so to reduce that overhead we use thread_local memory pools for states and messages, and optionally replace the system malloc with calls to tcmalloc[8]. In this way allocating threads do not block each other, and in a single threaded kernel we can leverage arena-style pools if desired.
This changed the ownership semantics of several objects in a non-trivial way, since the thread that creates an object has to destroy it (if it can prove it is no longer used). Experiments with synchronized pools proved slower than malloc/free.
Initially the kernels used strings as identifiers, as is done in PyPDEVS, profiling quickly indicated this to be a performance bottleneck. C++ strings are heap allocated variable sized objects with an atomic reference count. Access of that reference count across threads is expensive, as are the calls to malloc/free the string implementation makes to create/destroy new object, or copy existing.
Strings are more intuitive to work with from a user's standpoint. So as a compromise the user can reference models/ports by string name (usually when constructing the model). Once simulation starts all objects use integral identifiers for performance. This also increased usage of the constexpr feature of C++11 in, amongst others, timestamps and message headers.

Raw pointers

While an important C++11 feature in general, our initial usage of smart pointers for some types of objects was misplaced. Used across threads the reference counting becomes prohibitively expensive, and the (de)allocating caused significant contention between threads. Models are still held by a smart pointer, as is a kernel, but a message is a raw pointer to compacted memory.

Locking

Locking between kernels uses mostly atomic operations, where we can occasionally leverage memory orderings to only pay for synchronization when we need it. Messages are exchanged via a shared set of queues each with a dedicated lock.
On a higher level, we avoid the sending of synchronization messages entirely by writing the timestamp directly into shared memory.

Sending of antimessages is fairly cheap in our implementation, since only the modified pointer to the original message is sent to the receiving kernel.

Schedulers

PythonPDEVS has a wide range of schedulers for the user to choose from, with performance of each depending on the simulation type. Profiling showed in our case that, for a C++ implementation, the heap implementation used in adevs was faster than any of the schedulers we had tested before. Unlike most node based heaps, this scheduler uses a fixed size array where a heap is rebuilt or modified in place depending on the nr of items to update. Items are only updated, never removed.

4. PERFORMANCE

4.1 Sequential Simulation

CPU Usage

Devstone

The Devstone [9] benchmark is highly hierarchical, using directconnect the dxexmachina kernels can exploit this whereas adevs needs to walk the structure of the model to pass events.

PHold

PHold [6] is a parallel oriented benchmark, sequential runtime is measured only as a baseline. The usage of random nr generators takes up a significant amount of runtime in this model.

Interconnect

Interconnect [14] is a benchmark where all models broadcast, creating a complete graph in terms of dependencies between models. It highlights in sequential simulation the cost of heap allocation (messages) in dxexmachina. As the model count increases, we see the expected quadratic increase in runtime in both kernels, but an increasing penalty for dxexmachina w.r.t. adevs. This effect is due to the heap allocation of the messages by dxexmachina, even though this is minimized using memory pools it remains significant.

Memory Usage

Platform and tools

Both dxexmachina and adevs use tcmalloc as memory allocator, in addition dxexmachina uses memory pools to further reduce the frequency of expensive syscalls (malloc/free/sbrk/mmap/...). Tcmalloc will only gradually release memory back to the OS, whereas our pools will not do so at all. If memory has been allocated once, it is from a performance point of view better to keep that memory in the pool. For this reason a memory utilization can be best measured using peak allocation. Profiling is done using Valgrind's massif tool [12] Platform used was a i5-3317U Intel cpu, 8GB RAM, Fedora 22 (kernel 4.2.6), a page is 4,096KiB.

Measure

Adevs passes messages by value (in container by reference), we pass by pointer. The runtime effects of this choice are already demonstrated in the interconnect benchmark, so in this section we measure memory usage in number of allocated pages. This combines text, stack and heap memory for the program profiled, from the point of view of the OS or user, this is the actual memory in use. It is important to note that, especially in the case of optimistic, not all this memory is in use by the kernel, since the pools will in general not return memory once it is allocated but keep it for later reuse.

Results

Devstone

Table 1. Devstone 40x40 t5e5, unit MiB, 4 kernels (if parallel)

Adevs	AdevsCon	DX	DXCon	DXOpt
44	70	42	75	363

Since conservative passes messages by pointer, it needs a GVT/LBTS implementation to organise garbage collection, this inevitable delay explains the higher memory usage w.r.t adevs.
Optimistic needs a more complex GVT/garbage collection algorithm plus the differences in LP virtual times are far larger, which explains the heavier memory usage. Devstone (flattened) is allocated in a chain, this means that the leafs in the dependency graph will do a lot of unnecessary simulation before having a revert, leading to quite severe memory pressure. Unlike conservative and sequential execution, memory usage in optimistic varies greatly depending on scheduling of kernel threads and drifting between kernels.

Interconnect

In section 4.2 the parallel performance of this benchmark is further explained. It is of interest for sequential to contrast with the runtime penalty that dxexmachina suffers w.r.t. adevs. Optimistic fails this benchmark, see for detailed analysis section 4.2.3. The discrepancy between both conservative implementations is detailed in 4.1.4.

Table 2. Interconnect w 40 t5e5, unit MiB, 2 kernels (if parallel)

Adevs	AdevsCon	DX	DXCon
39	39	35	52

Priority Network

The priority network model is detailed in section 4.4.

Table 3. Priority model n 16, m 9, p 10, t5e5, unit MiB, 2 kernels

DX	DXCon	DXOpt
35	58	69

4.2 Parallel Simulation

Devstone

The flattened models are allocated to kernels by giving each kernel a distinct section of the chain, resulting in a low ratio of inter-kernel to intra-kernel messages. For optimistic this can cause more reverts since the kernels will start to drift faster as the modelcount increases. Furthermore, optimistic is quite sensitive to an increase in kernels, since the delay before a revert propagates increases. Of note as well is the warm-up time this benchmark requires, for $n = d\text{x}w$ models, it takes timeadvance()*n transitions to activate the last model in the chain. For parallel this can be reduced to $\frac{ta()*n}{kernels}$ before the last kernel becomes active.

PHold

In Phold [6] allocation is specified in the benchmark itself, each kernel manages a single node with a constant set of subnodes. The parameter R determines the percentage of remote destination models.

The dynamic dependency graph is a very sparse version of the static dependency graph, penalizing conservative. Lookahead is ϵ, so conservative spends most of its time crawling in steps of ϵ. Since the dependency graph between kernels is a complete graph, this is not a simulation that scales in our implementation. For N kernels, each kernel has to query the null-time of N-1 kernels, resulting in O(N^2) polling behaviour. This benchmark therefore highlights the price that we pay for sharing those values instead of sending an actual null message. In a non-cyclic simulation with a non-trivial lookahead (e.g. devstone), that choice however does pay off.

Optimistic suffers little from the above problems, however due to the high interconnectivity a cascading revert is still possible. More seriously, a revert is very expensive in PHold due to our usage of C++11's random nr generators. The cost of a revert is dominated by the recalculation of destination models, not in allocating/deallocating states/messages. Again, this could be significantly reduced using lazy-cancellation. Once a revert happens the drift between the kernels increases fast, increasing the likelihood of more reverts.

Interconnect

In Interconnect the set of atomic models form a complete graph (w.r.t. connections), each model broadcasts messages to the entire set.

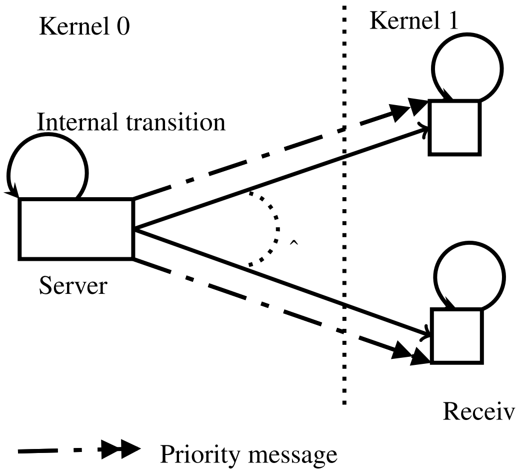
Allocation is irrelevant, the resulting dependency graph between kernels remains a complete graph. The runtime dependency graph is almost immediately equal to the static dependency graph. Conservative still faces the same issues as in PHold, with the key difference that for a fixed time advance lookahead is equal to the timespan between transitions. The scaling issue is identical as in PHold.

Our optimistic implementation does not complete an instance of this benchmark. The kernels get stuck in an infinite cascade of reverts. If kernel A reverts, it will send anti-messages to all others who in turn revert and send anti-messages to all others (and A itself again). Support for lazy cancellation could potentially undo this anti-pattern.

Priority network model

The priority benchmark is composed of a single server generating a stream of $0 <= m <= n$ messages at fixed time intervals, interleaved with a probability p for a priority message, to n receivers.

This defaults lookahead for the receivers to ϵ but this time there is no scaling effect, nor are there cycles in the dependency graph. This model therefore highlights the basic strengths/weaknesses of both synchronization protocols. Receiving models are allocated on another kernel than the server, and have a internal transition so will not wait for the incoming messages.



A key difference here with the other benchmarks is that a state (in the Receiver instances) is very cheap to copy/create, once the random nr generation is done the kernel holding the server will never revert since it is a source in the dependency graph. Optimistic will therefore not suffer the same performance hit in recreating states as it does in PHold.

5. RELATED WORK

5.1 PythonPDEVS

Dxexmachina is closely related to PythonPDEVS in design and philosophy. PythonPDEVS allows anyone who grasps the DEVS formalisms to immediately simulate his/her model without having to consider the kernel implementation. C++ implementations cannot hope to match the fast prototype/edit/run cycles provided by PythonPDEVS, although this can be minimized by building the kernels as libraries. Advanced features such as activity based relocation and the performance gains this results in, are still unique to PythonPDEVS.

5.2 Adevs

Adevs's source code is still under active development, allowing for an exact comparison in performance and features. It remains in most aspects the fastest simulation engine for the DEVS formalism, but it lacks an optimistic synchronization implementation. By virtue of not flattening Coupled Models, performance suffers in increasingly hierarchical models.

5.3 CD++

Different projects on CD++ offer conservative (CCD++) as well as optimistic (PCD++) parallel simulation. In contrast to our single program, with a non-fragmented code-base, neither projects offer both synchronization protocols. CD++ relies on the WARPED kernel. It is a middleware that provides memory, event, file, time and communication scheduling. WARPED is not used here since we operate explicitly on a shared memory system and since we wanted to design our kernels using the least amount of overhead possible.

6. CONCLUSIONS

Both synchronization algorithms offer good performance in differing simulations, in some simulation our kernels outperform adevs whereas in others we can still improve. The optimistic implementation needs to be extended with lazy evaluation/cancellation to function in cyclic simulations.

6.1 Future work

Activity

As shown in [15] activity and allocation of models across kernels is a key aspect in achieving high performance in any parallel implementation. Allocating models so that there are no dependency cycles between their containing kernels is a first step, but not always possible. For optimistic one can use reallocation to break (runtime dependency) cycles or perform load balancing. If kernels are unevenly balanced they will begin to drift fast, causing increasingly more reverts.

Hybrid

The optimistic implementation could use (null/eot/eit) from conservative to detect and/or reduce the cost of reverts without completely stalling on influencing kernels. Conservative kernels could be extended with runtime information about influencing models, if one can guarantee a static dependency is not used for a fixed time-span, this dependency can be removed for that period of (virtual) time.
Ultimately the simulation could switch at runtime between protocols based on the information provided by activity tracking.

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. Chandy, K. M., and Misra, J. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM 24*, 4 (Apr. 1981), 198–206.

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

4. 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.

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

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

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

8. Ghemawat, S., and Menage, P. TCMalloc : Thread-Caching Malloc. `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`, Nov. 2005.

9. 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.

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

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

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

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

14. Van Tendeloo, Y. Research internship i: Efficient devs simulation.

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

16. 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.

17. 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.

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