An overview of PythonPDEVS

Yentl Van Tendeloo[1] Hans Vangheluwe[1,2]
[1] University of Antwerp, Belgium
[2] McGill University, Canada

Yentl.VanTendeloo@uantwerpen.be
Hans.Vangheluwe@uantwerpen.be

Abstract:
We present an overview of PythonPDEVS, a family of DEVS simulation kernels. While a plethora of DEVS simulation kernels exist nowadays, we believe that there is a gap between low-level, compiled simulation kernels, and high-level, interpreted simulation kernels. PythonPDEVS fills this gap, by providing users with a high-level, interpreted simulation tool that offers features similar to those found in other high-level tools, while offering comparable performance to the low-level tools. In this paper, we focus on the three main motivations for the use of PythonPDEVS: (1) the use of Python as a high-level, interpreted language, which is also used by modellers to create their models, (2) a rich feature set, comparable to other high-level tools, and (3) decent simulation performance, comparable to other low-level tools. PythonPDEVS therefore aims at users new to DEVS modelling and simulation, or programming, while still offering competitive performance.

Keywords:
Python, DEVS, tool, distributed simulation, performance

1 Introduction

We present an overview of PythonPDEVS, a family of DEVS simulation kernels. While a plethora of DEVS simulation kernels exist nowadays, we feel that there still exists a gap. Most DEVS simulation kernels, like adevs [1], vle [2], or CD++ [3], are implemented in low-level, compiled languages, such as C++. These languages are a logical choice for the implementation of a simulation kernel, due to the resulting efficiency. But while efficient during simulation, modellers are required to write low-level code as well, which often requires compilation. This results in slower turnaround than if an interpreted programming language were used [4]. More feature-rich and high-level simulation kernels certainly exist, such as MS4Me [5] and X-S-Y [6]. And while they have lots of features and are easy to use, performance is not one of their top priorities. This results in slow simulations compared to lightweight, compiled simulation kernels, making them unsuited for large-scale DEVS models. There are thus two groups of simulation kernels: lightweight, compiled, and efficient on one hand, and high-level, feature-rich, and inefficient on the other hand. We believe that PythonPDEVS can fill in this gap.

The core characteristics of PythonPDEVS are presented next. First, it is implemented in Python, a **high-level, interpreted language**. Models are written in the same language, resulting in completely interpreted execution, avoiding compilation. This makes it faster to prototype simulation models, as it reduces the turnaround time. Second, PythonPDEVS offers a wide set of **features**, similar to currently available simulation tools. While PythonPDEVS itself is only a simulation kernel, tools like DEVSimPy [7] and the AToMPM DEVSDebugger [8] extend on it to provide even more features. Most features are supported out-of-the-box, without any custom code. Third, PythonPDEVS offers **decent performance**. Performance is not at the level of the lightweight, compiled simulation kernels, but is significantly faster than other feature-rich, high-level tools.

We will take a deeper look at the latter two: supported features, and the performance tweaks to achieve this performance. Our previous work [9, 10] investigated some features and their performance in detail. Here, we present an overview of our complete feature set.

2 Features

PythonPDEVS supports a wide set of features, similar to those found in other high-level DEVS simulation tools. While there is no graphical front-end, some work has been done on this by [7, 8, 11]. Many of the features supported by other tools, and more, become available through the use of these extensions.

Table 1 presents an overview of the features, and the cases in which they are supported. PythonPDEVS supports three modes of execution: *Sequential As-fast-as-possible* (Seq. AFAP), *Sequential Realtime* (Seq. RT), and *Distributed As-fast-as-possible* (Dist. AFAP). Note that distributed realtime is not supported at all. Due to the different characteristics of each mode of execution, some features are selectively available. For the not implemented features, we could not find a useful enough application to warrant the significant effort required.

For distributed simulation, Time Warp [12] is used to provide optimistic synchronization. With Time Warp, each distributed node will simulate ahead in time, in the assumption that no causality errors will occur. Should such an error occur (*i.e.*, a message from the past arrives), the simulation state is rolled back to the point in time where the message was supposed to arrive. This allows for parallelism, as all nodes can simulate independently, but incurs two kinds of overhead: a fixed one, to store each intermediate simulation state, and a variable one, to roll back the simulation state when a causality error is detected.

In realtime simulation, the simulated time is coupled with the wall-clock time, as shown in Figure 1. This means that in a single second of wall-clock time, the simulation will also progress exactly a single second. Scaled variants of this are possible, for example that each second in wall-clock time causes a progression of four seconds in simulated time. This is in contrast to as-fast-as-possible simulation, where the simulation simply progresses as fast as the system allows. Realtime simulation is mainly used when coupling the model with other realtime components, whereas as-fast-as-possible simulation is used when only the simulation results matter.

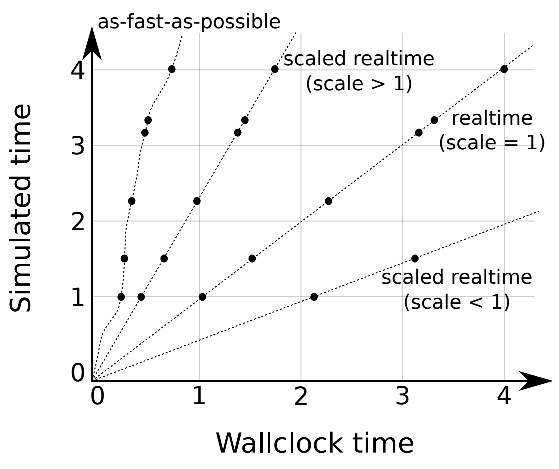

Figure 1: Realtime versus as-fast-as-possible simulation.

Table 1: Supported features in each situations.

	Seq. AFAP	Seq. RT	Dist. AFAP
Classic DEVS	Y	Y	N
Parallel DEVS	Y	Y	Y
Dynamic Structure DEVS	Y	Y	N
Tracing	Y	Y	Y
Checkpointing	Y	N	Y
Nested simulation	Y	Y	N
Termination condition	Y	Y	Y
Livelock detection	Y	Y	Y
Transfer functions	Y	Y	Y

The remainder of this section will briefly describe each feature, as well as why it is relevant for users to have this feature.

Classic DEVS Classic DEVS refers to the original DEVS formalism defined by [13]. It is still a viable formalism, and is still preferred by some people due to its simplicity. There is also not that much opportunity for parallelism and efficient execution. For performance reasons, most tools no longer support Classic DEVS, however. Due to difficulty with global synchronization, there is also no distributed variant of Classic DEVS implemented in PythonPDEVS.

Parallel DEVS Parallel DEVS [14] is an extension of Classic DEVS, which, as the name implies, offers more options for parallel execution. Apart from the prospect of parallel execution, there are also some additional optimizations in place that allow for faster execution, even during sequential execution. It is now becoming the default DEVS formalism implemented by tools, and is also the default in PythonPDEVS.

Dynamic Structure DEVS Dynamic Structure DEVS [15] is another extension of DEVS (both Classic DEVS and Parallel DEVS), where the model can be reconfigured at runtime. Some models, such as those where entities are created and destroyed during simulation, lend themselves much better to these cases, and are therefore best expressed using Dynamic Structure DEVS. PythonPDEVS does not support Dynamic Structure DEVS during distributed simulation, since this opens up the possibility for connections to change as well, which can have catastrophic results in optimistic synchronization if not handled well.

Tracing One of the most important artifacts of simulation is the simulation trace, which contains the simulation results. Different users need different traces: some prefer textual traces, whereas others want visual traces (*e.g.*, plots of the evolution of a value). In PythonPDEVS, multiple tracers can be plugged in, which are invoked at every transition that needs to be traced. Several are provided out-of-the-box, with the most helpful one being a normal textual trace. Visual tracers are provided as well. Users can easily define their own tracers, which are invoked at every simulation step.

Checkpointing Simulation of big models can take a long time, even using a fast simulation kernel. The longer a simulation takes, the higher the chance that it will be abruptly terminated due to hardware failure. Distributed simulations are even more prone to hardware failure, as they also introduce network communication. Especially for such big simulations, it is important to be able to continue a simulation after it was abruptly terminated. This is possible through checkpointing, where the current simulation state is periodically stored. Should simulation terminate abruptly, it is possible to continue simulation, without any loss of information, by restoring that snapshot. This feature is not supported in realtime simulation, as storing the current simulation state introduces significant and unpredictable delays in the simulation. Furthermore, it is difficult to conceive the semantics of restoring a crashed realtime simulation: the linear relation between wall-clock time and simulation time is broken.

Nested simulation In case the execution of the current model depends on the simulation of another model (possibly an abstracted model of the current model), it is useful to allow for nested simulation. The currently running simulation is suspended, and the nested simulation is started. After simulation, the result of the nested simulation is used to define some aspects of the currently running simulation. PythonPDEVS supports this feature thanks to a clear design, which avoids the use of global and static variables. For distributed simulation, it is possible to nest a sequential simulation inside a distributed simulation, but it is not possible to nest a distributed simulation in another simulation. While not impossible to implement, synchronization overhead is significant, and would require seperate synchronization algorithms.

Termination condition While most simulation kernels nowadays support the use of a termination time, PythonPDEVS also supports the use of a termination condition. In this case, a special function is invoked before each simulation step, which determines whether simulation should terminate or not. This termination function has access to the current simulation state and the current simulation time (to allow for time-based termination). This causes a significant overhead in the general case, but offers more possibilities to the user. For example in design space exploration, we want to quickly prune models that don't fulfill some basic requirements, which is only possible by checking these conditions at simulation time.

Livelock detection One of the problems that remain in the DEVS formalism, is the possibility of a time advance equal to zero. While this is not a fundamental problem, and is necessary for some situations, it is possible that they form a loop. In such a loop, simulation livelocks as the simulation never progresses in simulated time. Since simulation time no longer progresses, simulation of the model will never terminate, even though models keep being executed. Static analysis is difficult, if not impossible, since the time advance depends on the current simulation state, which is unknown beforehand. PythonPDEVS monitors model execution, and aborts execution if the simulation time has not increased after a sufficient number of transitions. This method sometimes marks fine models as erroneous (*i.e.*, allows for false positives), but will certainly terminate a locked simulation (*i.e.*, no false negatives).

Transfer functions Despite the fact that all variants of the DEVS formalism define the possibility for transfer functions, only several simulation tools actually implement it. The use of transfer functions eases the reuse of models in a different context, where a conversion between different types needs to happen. Due to the inefficiencies caused by a naive implementation, and the rare use by the community, many simulation kernels neglect to implement this aspect of the formalism.

3 Performance

As we try to differentiate ourself from other feature-rich and high-level DEVS tools, we also focus on improving simulation performance. PythonPDEVS therefore contains many internal optimizations, mainly focused on the use of "leaky abstraction". DEVS models are optionally augmented with domain-specific hints, which offer domain-specific knowledge about the model to the simulation kernel. Normally, a DEVS simulation kernel is unaware of the domain of the model it is simulating, and can therefore not use more efficient, but less general, algorithms.

Due to space restrictions, we did not include performance results here, but we refer to previous work [9, 10]. Using intrusive features (*e.g.*, tracing, checkpointing) affects performance.

Table 2 shows an overview of our applied optimizations, and when they are applicable. Some optimizations are not applicable in all modes of execution, as some focus on the problems arising from distributed simulation. Again, distributed realtime simulation is not supported and thus not shown.

Some of these optimizations are also implemented in other DEVS simulation tools, such as direct connection [16].

Table 2: Performance optimizations applied in each situations.

	Seq. AFAP	Seq. RT	Dist. AFAP
Direct connect	Y	Y	Y
Single loop	Y	Y	Y
Termination time	Y	Y	Y
No transfers	Y	Y	Y
Scheduler	Y	Y	Y
Migration	N	N	Y
Allocation	N	N	Y
Memoization	Y	Y	Y
State copy hints	N	N	Y
Event copy hints	Y	Y	Y

3.1 General optimizations

We first start with optimizations to the simulation algorithm itself, which are applicable in all situations. These optimizations are hardcoded and there is no way for the modeller to influence them.

Direct connect At the start of simulation, all connections are resolved to direct connections between two atomic models. Instead of having intermediate Coupled DEVS models, all of them are removed and links are made direct, thus reducing all intermediate algorithms to pass around the events. This process is called direct connection [17] and effectively reduces the hierarchy to a single Coupled DEVS model, with all Atomic DEVS models being its direct children. While this process might seem simple, special care must be taken when executed in combination with several of our features. With Classic DEVS, the select function reasons about only a single level, and is defined in terms of coupled DEVS models. This optimization is an implementation detail, and should therefore not be visible to the select function, which should still reason about the (no longer existing) coupled DEVS models. With Dynamic Structure DEVS, similar problems occur because runtime modifications should be applied to the original structure, and not to the direct connected structure. Therefore, after modifications are made, the direct connect algorithm is ran again to refresh all connections. In a distributed simulation, care should also be taken to make links correct, even between different simulation nodes. Furthermore, each node should still have a single Coupled DEVS model, without there being a single "global" Coupled DEVS model. Finally, transfer functions should also chain all calls that would normally be made through the path the event follows from its source to its destination.

Single loop Instead of closely following the abstract simulator [18], PythonPDEVS takes a significantly different approach. In our approach, we avoid all hierarchy (through the use of direct connection) but also all simulation messages, which are instead exchanged for method calls. During initialization, all models are gathered, and a single event list is constructed. From this event list, imminent models are popped, which are directly invoked with the commands to execute, instead of having to be retrieved from the hierarchy. In combination with direct connection, this removes hierarchy completely, as models are directly instructed from the main simulation loop. This causes a significant decrease of simulation overhead, as the loop becomes much tighter and messages are avoided altogether. Apart from removing overhead, it simplifies the simulation algorithm, as there is no need for the processing of simulation messages, nor Coupled DEVS models. Instead of a set of decoupled simulation message processing algorithms, PythonPDEVS uses a single loop as shown in Listing 1.

Listing 1: Single loop simulation algorithm

```
direct connection
initialize event list
while not done:
   pop imminent models from event list
   for each imminent model:
      mark model for internal transition
      collect output
      for each output port in output:
         for each connected input port:
            append event to model input
            mark model for external transition
   for each marked model:
      if marked for internal and external:
         invoke confluent transition
      else if marked for internal:
         invoke internal transition
      else if marked for external:
         invoke external transition
      compute time advance
      push new time on event list
```

Termination time While one of the features of PythonPDEVS is the use of a termination condition, this is not an efficient solution. In fact, most of the time, a point in simulated time suffices to determine if a simulation should terminate. Executing a function is rather slow in Python, so it is more efficient to explicitly inline the comparison of the simulation time with the termination time. Termination conditions are still supported, but if a termination time is defined instead, the inlined code is used exclusively. Certainly in distributed simulation, using a termination time provides vast improvements in simulation time.

No transfers Yet another feature of PythonPDEVS, which impacts performance, are transfer functions. Again, however, they are seldom used in practice, despite their implementation causing a significant overhead due to the frequent invocation of a function, which is most of the time the identity function. To avoid this slow path, the connection resolution phase of direct connection optimizes out identity functions. If a transfer function is detected on the path, only the explicitly defined functions are chained, as the implicit ones are again the identity function.

3.2 Simulation hints

The remainder of the optimizations are focused on the modeller providing additional information to the simulation kernel. DEVS models are augmented with additional information, but still remain valid DEVS models nonetheless.

Scheduler Many simulation kernels nowadays hardcode their event queue implementation, or scheduler, which determines which models are imminent at a specific point in time. While hardcoding this component provides some performance benefits, the ideal data structure depends on the model being simulated [9]. PythonPDEVS uses modular schedulers, which allows users to choose the most appropriate data structure for their model. In distributed simulation, it is even possible to use different data structures at different nodes. This feature is a hint to the simulation kernel, telling it how to efficiently manage its data. There is an **activity extensions** for this [19], which provides a *polymorphic data structure*: the data structure monitors accesses to the data structure, and optimize itself for the detected pattern.

Migration In distributed simulation, performance depends on the distribution of the computational load. While the modeller might have a good idea of how load is distributed, it might change throughout simulation. PythonPDEVS allows users to specify migration strategies as hints to the simulation kernel. The simulation kernel uses this strategy to migrate parts of the model between different nodes, if necessary. This mechanism is used to equalize the load over the nodes as simulation progresses. There are again **activity extensions** to this [20], which provide information on the measured load to the migration strategy. The simulation kernel optimizes for the evolution of the simulation state (*i.e.*, the future), instead of the past state. Apart from load measured in CPU time, users can provide further hints to the kernel on what to measure, which could even be a domain-specific notion (*e.g.*, amount of cars on a road).

Allocation While migration solves the problem of load distribution during simulation, it might not even be possible to find a good initial allocation, or it might be too difficult to set it up during model initialization. PythonPDEVS allows users to define an allocation strategy [10], which is invoked on the completely initialized model. Users can then specify the initial allocation based on the initialized model, instead of on a partially constructed model. The allocation strategy can also be extended using **activity extensions** [20], which allows for monitoring of an initial run of the simulation. This initial run is kind of like a sample run of the first few simulation steps. Load distribution and exchanged messages are monitored, and activity values are passed on to the allocation strategy, which uses this data to optimize the model distribution for the remainder of the simulation.

Memoization Optimistic synchronization using Time Warp incurred a variable overhead, due to the need to rollback simulation state in case of causality errors. The main cost is not restoring the simulation state, but repeating the same (or similar) simulation as before. As only several models are influenced by the arrival of an event, there is no need to recompute all models the node that was rolled back. With memoization, the state of atomic models is stored before and after execution of the transition function. When a rollback occurs, these states are not discarded, but placed in a queue. If the model is executed again, we can potentially reuse these saved states to avoid the execution of the transition function. This requires some hints to the simulation kernel, as the simulation kernel has no way to know whether or not two simulation states are equal. The modeler is required to provide a comparison function between two states.

State copy hints The fixed overhead of optimistic synchronization is caused by saving all simulation states, in order to be able to restore them if necessary. General ways of copying states, such as serialization or built-in deepcopy functions, have the disadvantage that they have to handle many corner cases as well, and are therefore not that efficient. By providing a more specific copy algorithm, telling the simulation kernel how to make a copy of the current simulation state, this overhead can be partially mitigated.

Event copy hints When an atomic model creates events, care should be taken not to break modularity, as these events potentially contain references or pointers to the state of other models. The host language (*e.g.*, Python or C++) has no way of knowing that this is not allowed in DEVS, and will therefore allow these operations. Such tricks, however, are in violation with the DEVS formalism, and should be considered abuse of the host language. In PythonPDEVS, events are therefore copied by default, such that each model receives a seperate copy of the event. While this is nice to have for people new to DEVS, it causes performance problems in both time and space. Therefore, there are three main options in PythonPDEVS: either the messages are naively copied (default), either they are not copied at all (for performance), or a user-specified function is invoked to perform the copy. This user-defined function is again a kind of hint to the simulation kernel, which allows it to stay conform to the DEVS formalism while being relatively efficient.

4 Conclusions and future work

We presented a brief overview of the main features of PythonPDEVS, as well as the performance improvements which make simulation sufficiently fast. Through the combination of a high-level, interpreted programming language, a feature-rich simulation tool, and decent performance, we believe that PythonPDEVS fills the gap which currently separates efficient, but low-level, simulation tools from the high-level, but inefficient, simulation tools. This makes PythonPDEVS ideal for people who want a lightweight and efficient DEVS simulation kernel, with low turnaround times, while still having access to functionality commonly only found in heavyweight tools. We achieve this by having an implementation in Python, which decreases development time of both the simulation kernel and the models, but also through the addition of domain-specific hints, which boosts simulation performance.

In future work, we will consider usability of our tool, going further in the direction of the AToMPM DEVSDebugger [8]. We will focus our efforts on five aspects: (1) modelling and simulation environment, allowing for easy creation and simulation of models, (2) library of DEVS models, which allows the sharing and reuse of DEVS models, (3) debugging environment, which transposes most of the features of code debugging to the world of model debugging, (4) experiment design environment, allowing the graphical definition of experiments as well, and (5) efficient simulation, making the tool applicable in more situations.

Acknowledgments :
This work was partly funded by a PhD fellowship 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] J. J. Nutaro, "adevs," http://www.ornl.gov/ 1qn/adevs/, 2015.

[2] G. Quesnel, R. Duboz, E. Ramat, and M. K. Traoré, "VLE: a multimodeling and simulation environment," in *Proceedings of the 2007 summer computer simulation conference*, 2007, pp. 367–374.

[3] G. Wainer, "CD++: a toolkit to develop DEVS models," *Software: Practice and Experience*, vol. 32, no. 13, pp. 1261–1306, 2002.

[4] J. K. Ousterhout, "Scripting: Higher-Level Programming for the 21st Century," *Computer*, vol. 31, no. 3, pp. 23–30, 1998.

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

[6] M. H. Hwang, "X-S-Y," https://code.google.com/p/x-s-y/, 2012.

[7] L. Capocchi, J. F. Santucci, B. Poggi, and C. Nicolai, "DEVSimPy: A collaborative python software for modeling and simulation of DEVS systems," in *Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, 2011, pp. 170–175.

[8] S. Van Mierlo, Y. Van Tendeloo, B. Barroca, S. Mustafiz, and H. Vangheluwe, "Explicit Modelling of a Parallel DEVS Experimentation Environment," in *Proceedings of the 2015 Spring Simulation Multiconference*, ser. SpringSim '15. Society for Computer Simulation International, 2015, pp. 860–867.

[9] Y. Van Tendeloo and H. Vangheluwe, "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] Y. Van Tendeloo and H. Vangheluwe, "PythonPDEVS: a distributed Parallel DEVS simulator," in *Proceedings of the 2015 Spring Simulation Multiconference*, ser. SpringSim '15. Society for Computer Simulation International, 2015, pp. 844–851.

[11] B. Barroca, S. Mustafiz, S. Van Mierlo, and H. Vangheluwe, "Integrating a Neutral Action Language in a DEVS Modelling Environment," in *Proceedings of the 8th International ICST Conference on Simulation Tools and Techniques*, ser. SIMUTools '15, 2015.

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

[13] B. P. Zeigler, H. Praehofer, and T. G. Kim, *Theory of Modeling and Simulation*, 2nd ed. Academic Press, 2000.

[14] A. C. H. Chow and B. P. Zeigler, "Parallel DEVS: a parallel, hierarchical, modular, modeling formalism," in *Proceedings of the 26th conference on Winter simulation*, 1994, pp. 716–722.

[15] F. J. Barros, "Modeling formalisms for dynamic structure systems," *ACM Transactions on Modeling and Computer Simulation*, vol. 7, pp. 501–515, 1997.

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

[17] B. Chen and H. Vangheluwe, "Symbolic flattening of DEVS models," in *Summer Simulation Multiconference*, 2010, pp. 209–218.

[18] A. C. H. Chow, B. P. Zeigler, and D. H. Kim, "Abstract simulator for the parallel DEVS formalism," in *AI, Simulation, and Planning in High Autonomy Systems*, 1994, pp. 157–163.

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

[20] Y. Van Tendeloo and H. Vangheluwe, "Activity in PythonPDEVS," in *Proceedings of ACTIMS 2014*, 2014.