The Modular Architecture of the Python(P)DEVS Simulation Kernel

Yentl Van Tendeloo[†] and Hans Vangheluwe[†,‡]
[†]University of Antwerp, Belgium
[‡]School of Computer Science, McGill University, Canada
yentl.vantendeloo@student.ua.ac.be, hans.vangheluwe@ua.ac.be

Abstract

We introduce two sequential simulation languages and supporting simulation tools: PythonDEVS for Classic DEVS, and PythonPDEVS, for Parallel DEVS. Python(P)DEVS is fully compliant with the standard definition of the DEVS formalism. Complex simulation initialization and termination conditions are supported. The main contribution is a modular architecture which allows the user to choose the scheduler, the realtime time management platform, the tracer(s), termination conditions, . . . Both as-fast-as possible and real-time simulation are supported. For real-time simulation (and deployment), three different platforms for time management are supported: thread-based, integration with UI event processing, and integration with the game loop of modern game environments. The simulation kernel is highly optimized and as its modularity allows for user-provided custom schedulers, model-specific knowledge can be taken into account, leading to high performance.

1. INTRODUCTION

PythonDEVS (a.k.a. PyDEVS) is a Classic DEVS[16] language grafted on the Python language, as well as a matching simulator. Python is a strongly typed, interpreted, object-oriented programming language. Recent changes to the original PyDEVS[3] enhance its simulation performance drastically. A variant, called PythonPDEVS, implements Parallel DEVS[6], allowing several additional performance improvements. Hence, most of our work is focused on improving PythonPDEVS.

The basic generator and queue model in Listing 1 serves as a simple example of Py(P)DEVS concrete textual syntax. More elaborate examples and how to use several options can be found in the documentation included in the package.

```
# A simple event Generator with IAT parameter
class Generator(AtomicDEVS):
   def __init__(self, IAT=1.0):
      AtomicDEVS.__init__(self, "Generator")
      self.IAT = IAT # Inter Arrival Time
      self.state = True
      self.outport = self.addOutPort("outport")

   def timeAdvance(self):
      if self.state:
         return self.IAT
      else:
         return INFINITY

   def outputFnc(self):
      # example output event content: [5,"a"]
      return {self.outport: [5,"a"]}

   def intTransition(self):
      self.state = False
      return self.state

# A simple Queue with processing_time parameter
class Queue(AtomicDEVS):
   def __init__(self, processing_time=1.0):
      AtomicDEVS.__init__(self, "Queue")
      self.state = None
      self.processing_time = processing_time
      self.inport = self.addInPort("input")
      self.outport = self.addOutPort("output")

   def timeAdvance(self):
      if self.state is None:
         return INFINITY
      else:
         return self.processing_time

   def outputFnc(self):
      return {self.outport: [self.state]}

   def intTransition(self):
      self.state = None
      return self.state

   def extTransition(self, inputs):
      # Only take the first element from the bag
      self.state = inputs[self.inport][0]
      return self.state

# A coupled model: Queue taking input from a Generator
class CQueue(CoupledDEVS):
   def __init__(self):
      CoupledDEVS.__init__(self, "CQueue")
      self.generator = self.addSubModel(Generator())
      self.queue = self.addSubModel(Queue())
      # connecting sub-models' output to input
      self.connectPorts(self.generator.outport,
          self.queue.inport)

model = CQueue() # create a model instance
sim = Simulator(model) # create a simulator
# can be configured with simulation end-time
sim.simulate() # run model simulation
```

Listing 1. PyDEVS example

We will elaborate on the major features of Py(P)DEVS, which include *compliance* (section 2.), *modular architecture* (section 3.) and *performance* (section 4.). Related work is explored in section 5.. Section 6. concludes the paper.

2. COMPLIANCE

Several criteria to define compliance with the Classic DEVS formalism were defined in [10]:

1. **Non-negativeness of the time advance function**: the time advance should not advance backwards.
2. **Correct output function**: the output function should be called exactly once before each internal transition function.
3. **Event passing instantaneity**: event passing between ports should take no simulated time.
4. **Precise time granularity**: the simulation should run at a fixed time granularity.
5. **Event time synchronization**: the time of occurence of events should be the same in all relevant (connected) models.
6. **Correct event sequence**: all events should be processed in the right temporal sequence.
7. **Correct tie-breaking**: simultaneous internal event occurrences should be handled correctly using the *select* function.
8. **Correct event dispatching**: events should be correctly dispatched from output port(s) to input port(s).
9. **Event independence**: all event instances should be unique (no references to each other nor to model states).

Whereas the early versions of PyDEVS failed criteria 1, 4 and 9, the current version passes all criteria. Two other Classic DEVS simulators, CD++ and X-S-Y, were also checked, which revealed that they both failed for some criteria. An overview of compliance is shown in table 1.

Both failed criterion 1, as they do not check the time advance value and hence allow simulations to go back into the past if the model erroneously computes a negative time advance. Furthermore, criterion 7 posed problems for both simulators, as neither support a *select* function. Instead of calling a *select* function, the first model in the imminent model list will be selected to transition first. The order of models in this list is implementation dependent.

Addtionally, CD++ failed on criterion 4. This is mainly due to the use of a single precision floating point number, instead of double precision. It does pass criterion 9, but this is however caused by the fact that only integers are allowed as messages and not because the simulator taking care of such a situation. X-S-Y failed criterion 9 due to the possibility for messages to contain references to other models or states. This problem could be easily resolved in Python (the implementation language of X-S-Y) by using e.g., the *deepcopy* library function. In PyDEVS, this problem is solved by offering the modeller the possibility to define the degree of compliance. By default, Python's *pickle* is used, though the user can define custom serialization functions too. If the modeller is certain that these references will not violate the DEVS formalism, not making copies is a performance optimisation.

Criteria	PyDEVS	CD++	X-S-Y
1	pass	**FAIL**	**FAIL**
2	pass	pass	pass
3	pass	pass	pass
4	pass	**FAIL**	pass
5	pass	pass	pass
6	pass	pass	pass
7	pass	**FAIL**	**FAIL**
8	pass	pass	pass
9	pass	pass	**FAIL**

Table 1. Compliance overview

Note that we only compared Classic DEVS simulators, as the compliance criteria differ for Parallel DEVS. The latter does away with the *select* function, adds a *confluent transition* function, and uses *bags* of events.

3. ARCHITECTURE

Algorithm 1 Basic simulation algorithm

```
$clock \leftarrow scheduler.readFirst()$
while not terminationCheck() do
    for all $scheduler.getImminent(clock)$ do
        Mark model with $intTransition$
        Generate and route output
        Mark destinations with $extTransition$
    end for
    for all marked models do
        Perform marked transition
        Send info about the performed transition
        to subscribed tracer(s)
    end for
    $scheduler.massReschedule(transitioning)$
    Clean model transition marks
    $clock \leftarrow scheduler.readFirst()$
end while
```

The design of the Py(P)DEVS simulator is modular, with as a prime example, the modular support for realtime simulation. Other examples include modular tracers (for model validation), a user-selectable modular scheduler (for performance) and the support for termination condition(s) (for versatility). A simplified version of the simulation algorithm is shown in algorithm 1. The realtime version differs in that it only executes a single step and then waits for the required time.

3.1. Modular realtime simulation

Apart from *as-fast-as-possible* simulation, Py(P)DEVS also supports *realtime* simulation. The latter uses almost exactly the same algorithm as *as-fast-as-possible* simulation, as only the main simulation loop differs due to possible asynchronous *user-provided* input and the requirement to wait after every transition phase until the appropriate wall-clock time. Realtime simulation therefore has exactly the same set of features as as-fast-as-possible simulation. The only difference is the *termination function*, which is only evaluated at the time of processing a transition, for performance reasons). As a consequence, the realtime simulator may overshoot a termination condition which depends on the value of simulated time.

Py(P)DEVS supports three different platforms for realtime simulation:

1. **Raw threads** are the straightforward way to implement realtime simulation. Waiting for the correct time is done by means of a *Python Event* as provided by the standard library. A "wait" on this event will occur. This implementation is simple, but it starts many threads in cases where many inputs are scheduled. Events can be unscheduled by manually setting the `Event` object, thus terminating the thread. This is for example necessaary when an external input is received. The use of raw threads is appropriate when implementing for example network protocols.
2. **UI events** are useful when interaction between the simulator and a user interface (such as one based on the Tk library) is required. For such situations, the raw threads solution fails, and the UI's event management facilities must be used to correctly interleave UI and simulation events. The major difficulty is that the calls to schedule the event must be done in the main thread of the program, where the Tk main window was created. For this reason, we have the other threads call the main loop to do the actual scheduling into the Tk event list. Unscheduling also happens in the same way. The actual scheduling logic is provided by Tk itself, and only a wrapper around it needed to be written.
3. The **Game loop** mechanism allows the realtime simulation to be incorporated within a game loop. This loop, typically with a fixed frame rate, both updates the game state and renders the representation of that state. Within each frame, a single call is made to the simulator. The simulator does hence not have control over the advancement of time. It can only observe the time to which the game loop has advanced, and process all events (over)due by that time. In this approach, the accuracy is limited to the frame period. In order to notify the game loop whether the termination condition is satisfied, a callback method is provided for the game loop to query the simulator.

It is completely transparant to the modeler which of the three platforms is used. Adding additional platforms is simple and only requires the user to write a small interface. The main function *scheduleAfter*, takes as a single argument, the function to run after a (provided as argument) delay has passed. Adding the game loop mechanism only took 50 lines of platform-specific code, demonstrating the elegance of the modular design.

Events are not only *schedulable*, but also *cancelable* (or unschedule-able). On the one hand, an internal transition that was scheduled to occur, should be cancelled as soon as another event is provided by the user. On the other hand, all events that are read from a file are pre-scheduled at the start of the simulation, so these should not be unscheduled.

In addition to the threading platform, the external event senders are also written as modularly as possible. Two input methods are supported:

1. With **user input during the simulation**, the simulator will present a prompt to the user during the actual simulation. The time of the event will be determined by the simulation time at the moment the message is injected (that is, as soon as the *return* key is pressed). This method can also be used to halt the simulation prematurely, simply by injecting the empty string or any invalid input.
2. With **file input** from a file containing time-ordered event notices (time-event pairs). Entries from this file are parsed "on demand", as the simulation time advances. Reading in the whole file at the start of the simulation and pre-scheduling all event notices may be more efficient for small files, but does not scale for large input event traces. Note that such a simulation will run in batch mode, until the termination condition is satisfied.

For maximum flexibility, a combination of both methods is supported, making it possible to use a file as a generator, while the user still provides manual input. Input is always provided in the form `inputPort inputValue`, where the `inputPort` is a string that is mapped to a `Port` object. This mapping, for all ports in the model, is constructed at the start of the simulation. Only strings, and not arbitrary Python objects can be interactively injected during the simulation.

Events can be put on every possible input port of the model, even on those of models deeply nested in the model hierarchy. This partially breaks modularity. The much more intrusive alternative is however to change the model under study and create a series of ports and connections, from the topmost coupled model down to the desired nested model. When Parallel DEVS is used, the message has to be put in a bag before insertion into the simulation.

Simulations can be run in *scaled realtime* where the ratio R between simulated and wall clock time specifies *realtime* ($R < 1$), slower-than-realtime ($R < 1$), and faster-than-realtime($R > 1$).

3.2. Modular tracing

Py(P)DEVS supports the use of several different tracers, which can be useful for debugging. The supported tracers are:

1. **Verbose** tracing will output all available information about the simulation. It will display the type of transition that happens, on which model it happens and the effect on the model. Additionally, the incoming and outgoing messages are shown for each port. This happens in a human readable form, as to allow simple debugging. The output of this tracer is difficult to process automatically, for which reason two other tracers are present.
2. **XML** tracing will output the information to XML with the structure defined in [15], which also includes a tool to visualize these traces. The main advantage of this trace is that it is very versatile and is simple to parse. Note however, that such traces can become very big due to the verbosity of XML and the amount of data that is being logged.
3. **VCD** tracing outputs the information in Value Change Dump[1] format, which is mainly used by languages such as Verilog. The output files are relatively small, though only binary values, *floating* and *error* values are allowed as messages and states. These traces can be visualised using e.g. *GTKWave*.
4. **Cell** tracing is a specific tracer for models that can be visualised in a grid. Several options are offered, such as writing to multiple files for easy batch processing. It is similar to the tracer present in CD++[2], though some important variations exist. First of all, our tracer is limited to 2D models, though this limitation is not fundamental. On the other hand, this tracer is not limited to Cell-DEVS models, but allows arbitrary models to be assigned *x* and *y* attributes, which specify the location of the model. The state also has a method to retrieve the state in a single number, which is again user-definable. The resulting files then contain a matrix representation of the grid, which can be visualized with most plotting tools. It offers a more visual tracing environment in those specific cases where it is possible. An example of a *fire spread* model is shown in figure 1.

Adding a new tracer is very simple and only requires the addition of functions to be called when an internal and external transition happen. These functions take the model on which the transition happened as a parameter.
Since tracing provides a big simulation overhead, it is possible to disable tracing completely, though the simulation will not provide any output whatsoever. All tracers are also completely independent, thus it is possible to have multiple tracers running simultaneously.

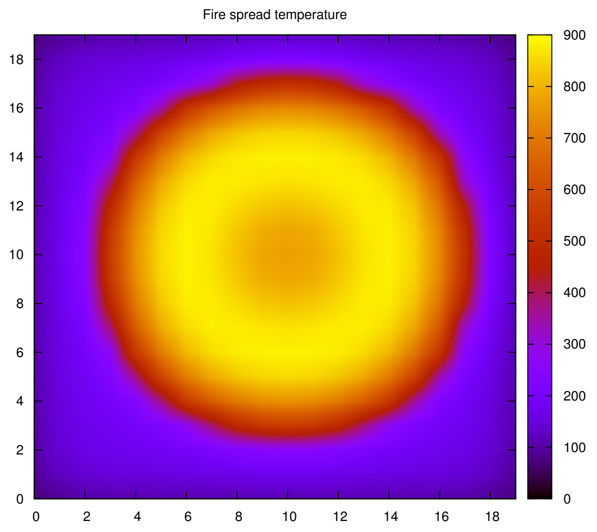

Figure 1. Cell trace of a fire spread model

```
__  Current Time:       1.00 _______________________

 EXTERNAL TRANSITION in model <Processor>
   Input Port Configuration:
     port <inport>:
        Event = 1
   New State: 0.66
   Next scheduled internal transition at time 1.66

 INTERNAL TRANSITION in model <Generator>
   New State: 1.0
   Output Port Configuration:
     port <outport>:
        Event = 1
   Next scheduled internal transition at time 2.00
```

Listing 2. Example verbose output

3.3. Modular scheduler

One of the most time-critical (and complexity-defining) parts of DEVS simulation is the scheduler being used. Nearly every different DEVS simulator uses another datastructure to use as the basis of the scheduler. The simplest schedulers being a simple eventlist (as specified in the abstract simulator[16, 5]). More complicated schedulers, which also yield a better efficiency, are based upon the idea of a heap. Even in these seemingly conceptually identical schedulers, slight variations are visible. Take for example vle[14] and adevs[12]. Both use a heap for their scheduler, though they use it differently. In vle, a model is rescheduled by invalidating the model in the heap and inserting a new (valid) copy in the heap. This way the heap invariant stays true and there is no need to pop random elements from the heap, making a standard heap implementation suffice. On the other hand, adevs extends the basic heap and adds some specific code to make the popping of random elements possible, though at a

relatively high cost. All these schedulers have specific situations in which they are fastest. In several situations, some user knowledge might be present to enhance the scheduler. For this reason, PyPDEVS supports a user-defined scheduler too. This offers the user the possibility to define a custom scheduler, as long as the same interface is used. Such a scheduler doesn't even need to provide complete DEVS-compliance, as long as it is compliant to the desires of the user and the actual model being simulated.
Of course, most users will not be tempted to write their own scheduler, though it offers an extra opportunity for those users that require every bit of performance. For these users, PyPDEVS already includes 7 different schedulers, of which 5 are suitable for general purpose use. These other 2 schedulers are specifically written for a subset of all possible DEVS models, where some shortcuts can be taken in the scheduling logic.
The supported schedulers are:

1. **Sorted list**: this implements the simplest scheduler of all: a basic sorted list. It is clearly the least efficient, at least complexity wise. It performs decently in several situations, though it has very inefficient scheduling if there are lots of inactive models or if very little models change in the simulation step.
2. **Minimal list**: similarly simple to the *Sorted List*, though it searches for the minimal element in each iteration.
3. **Activity heap**: this is the default scheduler of PyPDEVS. It maintains a heap with all scheduled elements, which is then simply updated by pushing and popping new elements using the *heapq* library in Python. Reschedules are handled by invalidation, followed by a periodic cleanup. This scheduler offers nice efficiency in most cases, though it could be problematic in cases where a lot of invalidations happen, as the size of the heap can actually become much larger than the number of models. If the required cleanup is triggered, it will take some time to completely reconstruct the whole heap. This scheduler partially takes activity into account, meaning that models that are scheduled for *infinity* are simply not taken into account and therefore do not influence the complexity.
4. **Dirty Heap**: a copy of the *Activity Heap* scheduler, but without periodic cleanup. It is general, though there is no bound on the amount of memory consumed, nor on the time complexity of operations on this heap. The main advantage is that the periodic cleanup is often unnecessary and can now be avoided. In the worst situation, it is possible for the heap to grow larger at every timestep, making the time and space complexity unbounded.
5. **Heapset**: the datastructure is still a heap, though it doesn't contain the elements themself, but only the time at which they should transition. This time can then be used to look up the actual models in a hashmap. The primary advantage is that it minimizes the size of the heap by only including times, thus colliding models do not increase its size. Thus the heap can be kept very small if lots of models are scheduled at the same time. Note that this scheduler takes advantage of the efficient dictionary (a kind of hashmap) implementation in CPython. It tries to be good at everything, at the cost of never being the best, making it a good scheduler for general situations.
6. **No Age**: a copy of the *Heapset* scheduler, but without an age field, thus making it non-general. This allows slightly more simple comparisons internally.
7. **Fixed time**: this scheduler will only have a list of 'scheduled' and 'not-scheduled' models. Such a simplification allows for many optimisations, though it is very specific. It is applicable in models where every model that must transition, transitions at exactly the same time.

	Average case	Worst case
Sorted list	$O(n \cdot log(n))$	$O(n \cdot log(n))$
Minimal list	$O(n)$	$O(n)$
Activity heap	$O(k \cdot log(n))$	$O(n \cdot log(n))$
Heapset	$O(k \cdot log(n))$	$O(n \cdot log(n))$
Fixed time	$O(k)$	$O(n)$
No Age	$O(k \cdot log(n))$	$O(n \cdot log(n))$
Dirty heap	$O(k \cdot log(n))$	$O(\infty)$

Table 2. Complexity of the different schedulers. k is the number of reschedules and n is the total number of models in the simulation. The major difference is often in the constant factor.

Each of these schedulers has its specific situations, for example the *Minimal list* in case lots of reschedules happen, or the *Heapset* in cases where only a small number of reschedules happen at each iteration. Three different kinds of situations are tested and all schedulers are compared.

The first one in figure 2 contains the *DEVStone* benchmark, where lots of reschedules happen due to many collisions. The second one in figure 3 contains the same benchmark, but with a random time advance, thus preventing collisions. The final one in figure 4 contains the *Fire Spread* model, but now shows that a slight advantage can still be gained by using a *specific* scheduler [1].

[1]This difference is even more significant when different interpreters are used, as the *sort()* method of a list is implemented rather efficiently in *CPython*, due to the implementation in C.

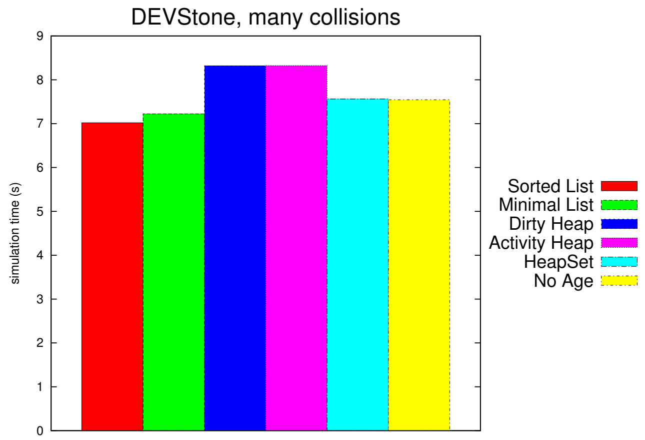

Figure 2. Scheduler comparison for DEVStone with many collisions

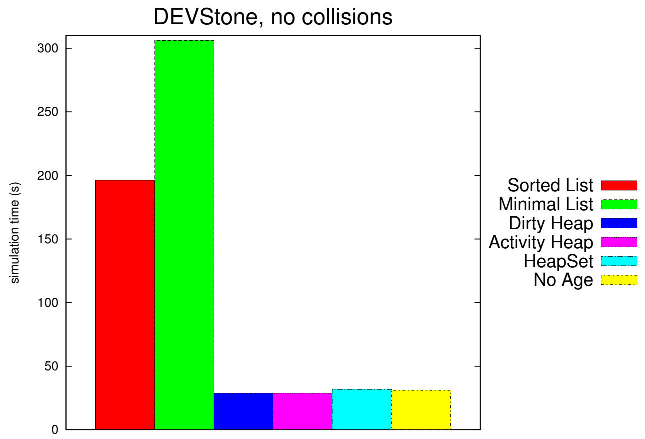

Figure 3. Scheduler comparison for DEVStone without collisions

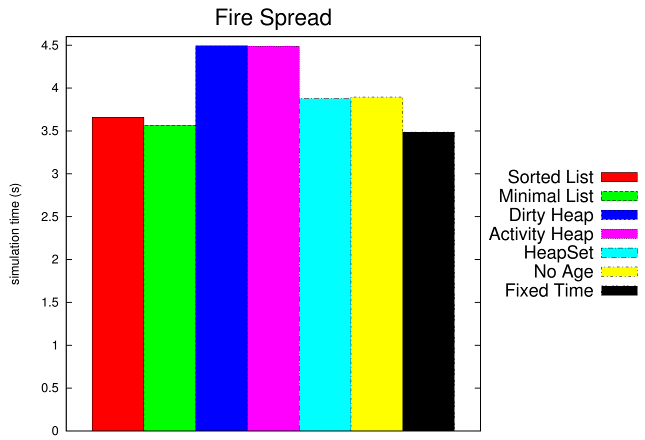

Figure 4. Scheduler comparison for Fire Spread

From these different simulations, it becomes clear that noticable speedups can be achieved in such a way. Varieties in memory usage are also to be expected, though they are not shown here.

The idea of using a specific datastructure for specific situations was also used in e.g. Meijin++[13], where the datastructure could even be changed at runtime if it was detected that a speedup could be gained in that way. PyPDEVS currently offers basic support for a polymorphic scheduler, which chooses at run time between either the *heapset* or *minimal list* scheduler, based on the patterns seen in the last simulation steps. The additional cost of statistics gathering and swapping the scheduler can often be made up for if the model has either variable behaviour, or if the user simply has no idea which scheduler to use. Of course, manually selecting the best scheduler is still slightly faster, but is impossible for fluctuating scheduling patterns.

3.4. Modular termination condition

Another specific feature of Py(P)DEVS is the possibility to use a termination condition instead of a termination time. Other simulators only allow the simulation to halt after a specific simulation time, whereas we allow the modeller to define a condition that should be checked at every simulation step. Such a condition could check for an unacceptable situation and immediately halt simulation if such a situation is detected. Whereas the actual simulation speed is not improved, it offers the possibility of stopping the simulation earlier.
The main disadvantage of this approach is that it incurs a slight performance overhead in situations where only a termination time is desired, as function calls have a high overhead in Python. Other implementation languages might support inlining to avoid this overhead.

4. PERFORMANCE

Even though Py(P)DEVS is written in Python, one of its new focal points is performance. The first version was among the slowest DEVS simulators available, mainly due to the simple simulation procotols and the lack of severe optimisations to the abstract simulator.
The complete simulation algorithm was revised and the complexity was severely reduced. The most invasive optimisations were the use of different schedulers for different situations and the use of direct connection[4], several other optimisations were adopted from [11].
Additional speedup can be observed when using the PyPy Python interpreter instead of the default CPython interpreter. Using another interpreter does not lower the complexity[2],

[2]It could be possible that some library functions are implemented differently, potentially offering a difference in complexity depending on the situation.

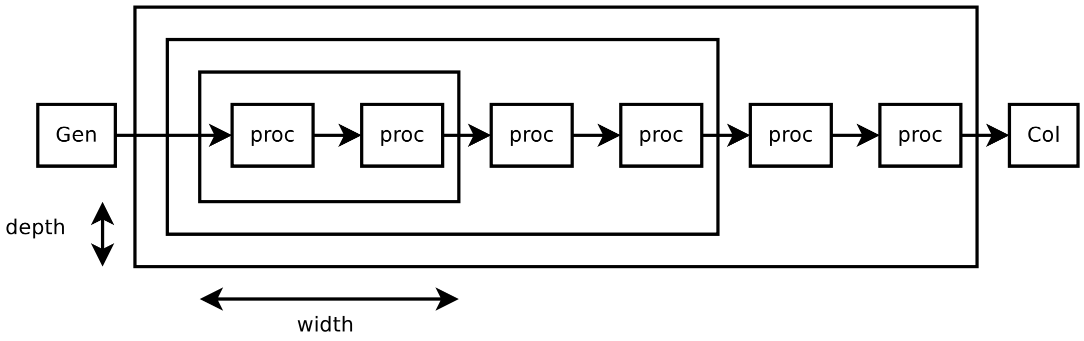

Figure 5. DEVStone model, here the width is 2 and depth is 3

though it can alter the complexity by a constant factor.
Another option would be to use Cython to create a compiled version of the simulator. Simply compiling the model without any additional information provided nearly no speedup. To obtain more significant speedups, static typing should be introduced for the performance critical functions. The main problem with this is that it requires intrusive changes, preventing the model from being run using a normal Python interpreter, as it was no longer valid Python code.
To show a small comparison, we performed a comparison using DEVStone[7] with many collisions. To put the performance in perspective, we compare our performance to adevs[12], which is currently one of the fastest available DEVS simulators and is written in C++. Since adevs implements the Parallel DEVS[6][3] formalism, we compared it to PyPDEVS.
Our DEVStone model is rather artificial, though it clearly shows the complexity in the number of models in situations with many collisions. These results were obtained using an *Intel i5-2500, 3.30GHz with 4GB main memory*.
Knowing that adevs uses a compiled programming language, PyPDEVS compares very favorably when the *PyPy* interpreter is used, which has JIT capabilities.
The normal adevs benchmark was done without any compilation flag, thus disabling any compiler-induced optimisations. If adevs was compiled with optimisations (*gcc -O2*), adevs was the fastest again. However, the PyPy JIT is still work in progress, providing the potential for even higher performance in the future without any further optimisations to PyPDEVS itself. Furthermore, this simulation time included the JIT code generation (which clearly happens at run-time) and several parts of code were never translated, resulting in normal *interpreted* execution which is said to be even slower than usual in the PyPy documentation.
In PyPDEVS, we choose to use the *minimal list* scheduler, as the DEVStone benchmark causes a number of collisions that is dependent on the number of models. In such situations, the

[3]Technically, it is the DynDEVS formalism, though the difference doesn't matter here

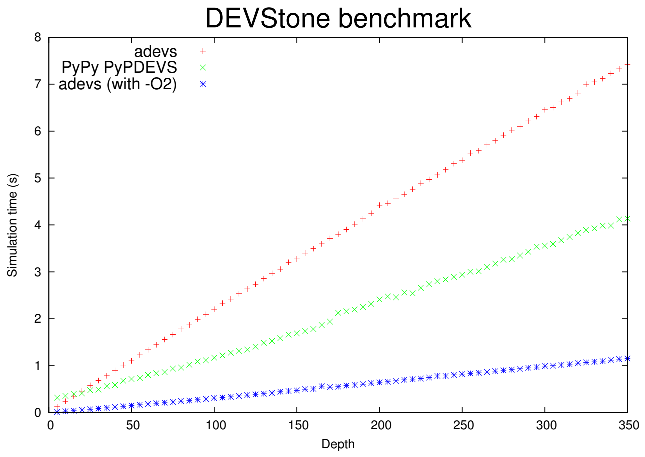

Figure 6. DEVStone comparison between PyPDEVS and adevs; PyPDEVS using the *minimal list* scheduler

minimal list scheduler is the ideal scheduler due to its low time complexity that is independent of the number of collisions. Note that there is some slight jitter in the *PyPy* timings, even though the results are the average of 5 simulation runs. This is caused by the garbage collector that gets called after some threshold in memory usage is reached. Furthermore, the JIT causes some slight deviations in short simulations due to the warmup time.
PyPDEVS also includes several features over adevs, such as the possibility for realtime simulation, several tracers and the use of a termination condition. Additionaly, PyPDEVS models are written in Python, offering all the advantages of Python to the modeller. This allows for example to change the model and rerun it without recompilation. On the other hand, the adevs simulator and the model are compiled together into a single executable. Another advantage is dynamic typing: in PyPDEVS it doesn't matter what kind of messages are passed, whereas in adevs, the same type has to be used (though inheritance can be used).

5. RELATED WORK

The idea of modular design is also present in JAMES II[8], though our work is focused solely on DEVS.
Nearly every different simulator uses its own kind of scheduler, which can be highly efficient in the problem domain for which the simulator was designed. Such examples include a sorted list (original PyDEVS[3]), minimal list (CD++[2]) and a dirty heap (vle[14]). The type of scheduler used is nearly never documented, forcing the user to delve into the source code (if available at all).
X-S-Y[9] is another DEVS simulator written in Python that supports realtime simulation, though it only supports the use of raw threads.

6. CONCLUSION AND FUTURE WORK

We presented a new version of PyDEVS, a DEVS simulator written in Python, compliant with the Classic DEVS specification. It supports both as-fast-as-possible and realtime simulation using different threading platforms. It offers many of its features in a modular way, without compromising simulation efficiency and, in combination with PyPy is even one of the fastest DEVS simulators. It uses Python as its implementation language, allowing for very readable code in both the simulator and the models.
Future work is focused on the development of PyPDEVS, by providing a distributed and parallellised version of the current PyPDEVS implementation. The same kind of tracers and termination conditions will be supported. Realtime simulation will still be supported, though only when no distribution is used. The main difference is the possibility to use multiple nodes using Time Warp optimistic synchronization. This implementation will again focus on performance.

ACKNOWLEDGMENTS

Jean-Sébastien Bolduc is gratefully acknowledged for his work on the original prototype of the PyDEVS simulator as well as Ernesto Posse for his later enhancements.

REFERENCES

[1] IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, 2006.

[2] Javier Ameghino and Gabriel Wainer. Application of the Cell-DEVS paradigm using N-CD++. In *In Proceedings of the 32nd SCS Summer Computer Simulation Conference*, 2000.

[3] Jean-Sébastien Bolduc and Hans Vangheluwe. The modelling and simulation package PythonDEVS for Classical hierarchical DEVS. Technical report, McGill Univ., 2001.

[4] Bin Chen and Hans Vangheluwe. Symbolic flattening of DEVS models. In *2010 Summer Simulation Multiconference*, SummerSim '10, pages 209–218, San Diego, CA, USA, 2010. SCS.

[5] A.C. Chow. Abstract simulator for the Parallel DEVS formalism. *AI. Simulation, and Planning in High Autonomy Systems*, 1994.

[6] Alex Chung Hen Chow and Bernard P. Zeigler. Parallel DEVS: a parallel, hierarchical, modular, modeling formalism. In *Proceedings of the 26th conference on Winter simulation*, WSC '94, pages 716–722, San Diego, CA, USA, 1994. SCS.

[7] Ezequiel Glinsky and Gabriel Wainer. DEVStone: a benchmarking technique for studying performance of DEVS modeling and simulation environments.

[8] J. Himmelspach and A.M. Uhrmacher. Plug'n simulate. In *Simulation Symposium, 2007. ANSS '07. 40th Annual*, pages 137–143, 2007.

[9] Moon Ho Hwang. X-s-y. `https://code.google.com/p/x-s-y/`, 2012.

[10] Xiaobo Li, Hans Vangheluwe, Yonglin Lei, Hongyan Song, and Weiping Wang. A testing framework for DEVS formalism implementations. In *Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, TMS-DEVS '11, pages 183–188, San Diego, CA, USA, 2011. SCS.

[11] Alexandre Muzy and James J. Nutaro. Algorithms for efficient implementations of the DEVS & DSDEVS abstract simulators. 2005.

[12] James J. Nutaro. adevs. `http://www.ornl.gov/~1qn/adevs/`, 2013.

[13] Nicolas Patrick. Meijin++, reference manual, 1991.

[14] Gauthier Quesnel, Raphaël Duboz, Éric Ramat, and Mamadou K. Traoré. Vle: a multimodeling and simulation environment. In *Proceedings of the 2007 Summer Computer Simulation Conference*, SCSC, pages 367–374, San Diego, CA, USA, 2007. SCS.

[15] Hongyan (Bill) Song. Infrastructure for DEVS modelling and experimentation. Master's thesis, School of Computer Science, McGill University, 2006.

[16] Bernard P. Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of Modeling and Simulation*. Academic Press, second edition, 2000.