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

Work In Progress paper

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. 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[12] language grafted on the Python language, with a matching simulator. Python is a strongly typed, interpreted, object-oriented programming language. Recent changes to the original PyDEVS[1] enhance its simulation performance drastically. A variant, called PythonPDEVS (a.k.a. PyPDEVS), implements Parallel DEVS[4], allowing several additional performance improvements. Hence, most of our work is focused on improving PyPDEVS. Most of our discussion is relevant to both of them, in which case it is referenced as Py(P)DEVS. 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. PyPDEVS example

We will elaborate on the major features of Py(P)DEVS: a *modular architecture* (section 2.) and *performance* (section 3.). Related work is explored in section 4. Conclusions are given in section 5.

2. ARCHITECTURE

The design of the Py(P)DEVS simulator is modular, with as a prime example, the modular support for scaled-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

Algorithm 1 Basic simulation algorithm

```
clock ← 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 ← scheduler.readFirst()
end while
```

in that it only executes a single step and then waits for the required wall-clock time.

2.1. Modular realtime simulation

Apart from *as-fast-as-possible* simulation, Py(P)DEVS also supports *scaled realtime* simulation. The latter uses 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.
Three different platforms are supported:

1. **Raw threads** are the straightforward way to implement realtime simulation. Waiting for the correct wall-clock 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 and relies on the Python implementation for decent accuracy. Events can be unscheduled by manually setting the `Event` object, thus terminating the thread. This is necessary 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. The raw threads solution fails, as the UI's event management facilities must be used to interleave UI and simulation events. The major difficulty is that callbacks should be used to the simulator, meaning that we should hand over all control to another program. 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.

The used platform is completely transparant to the modeler. Adding additional platforms is simple and only requires the user to write a small interface. The main function `wait`, takes the function to run and its delay as arguments. Adding the game loop mechanism only took 40 lines of platform-specific code, demonstrating the elegance of the modular design.
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 can be interrupted using the `interrupt` function. The time of the event will be determined by the wall-clock time at the moment the message is injected (taking into account the scale factor). The interrupt will be processed immediately, except in the game loop platform, where it will only be processed when the *step* function is called.
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 advances. Reading in the whole file at the start of the simulation and prescheduling all event notices is more efficient for small files, but does not scale for large input event traces.

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

```
__  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

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. Supporting arbitrary Python object would complicate the parsing of input.
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, which is done transparantly by the simulator.
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 ($R > 1$).

2.2. Modular tracing

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

1. **Verbose** tracing outputs all available information about the simulation. It displays the type of transition that occurs, on which model it occurs and the effect on the model. Additionally, the incoming and outgoing messages are shown for each port. This happens in a human readable form to allow simple debugging. A sample verbose trace is shown in Listing 2. The output of this tracer is difficult to process automatically. To address this issue three other tracers are provided.
2. **XML** tracing outputs the information to XML with the structure defined by Song [11]. Song also provides a tool to visualize these traces. The main advantage of this trace is that it is very versatile and is simple to parse. Such traces can become very large however, due to the verbosity of XML and the granularity of logging.

Adding a new tracer is very simple and only requires the addition of functions to be called when an internal, external or confluent transition happen. These functions take the model on which the transition happened as a parameter.
Since tracing incurs a large simulation overhead, it is possible to disable tracing completely. Final output of aggregate performance measures such as average queue lengths can still occur however. All tracers are completely independent. It is thus possible to run multiple tracers simultaneously.

2.3. Modular scheduler

One of the most time-critical (and complexity-defining) parts of discrete-event (and DEVS in particular) simulation is the scheduler used. The scheduler keeps track of event/model notices (i.e., time-value pairs). Nearly every DEVS simulator uses a different data structure as the basis of the scheduler. The simplest scheduler is an eventlist, as is used in the abstract simulator[12, 3]. More complex schedulers are based on a *heap*. Even in seemingly identical schedulers, slight variations are found. For example, `vle`[10] and `adevs`[8] both use a heap for their scheduler, though they use it differently. `vle` uses a general-purpose heap and uses invalidation to cope with reschedules. On the other hand, `adevs` reimplements a heap and makes reschedules a native operation.

In several situations, some user knowledge might be encoded to enhance the scheduler. It is for this reason that Py(P)DEVS supports a user-defined scheduler. This offers the user the possibility to define a custom scheduler, as long as the same interface is used. Such a scheduler does not even need to be DEVS-compliant, as long as it is compliant with the actual model being simulated. This allows for example for the efficient simulation of discrete-time models.
Of course, most users will not wish to write their own scheduler, though it offers an extra opportunity for when every bit of performance is required. Py(P)DEVS already includes a variety of schedulers.
The supported schedulers are:

1. A **sorted list** is an event list which is continuously kept sorted on the time attribute. 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 many inactive models or if very few models change in a simulation step.
2. The **minimal list** is an event list too, but is unsorted. It iterates over the complete list for every operation.
3. An **activity heap** is a scheduler that maintains a heap with all scheduled elements. It is updated by pushing and popping new elements using the `heapq` library in Python (a priority queue implementation). Reschedules are handled by invalidation, followed by a periodic

	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	$O(k \cdot log(n))$	$O(n \cdot log(n))$
No Age	$O(k \cdot log(n))$	$O(n \cdot log(n))$
Dirty heap	$O(k \cdot log(n))$	$O(\infty)$

Table 1. Complexity of the different schedulers. k is the number of reschedules and n is the total number of models in the simulation.

cleanup. Performance can be problematic in cases where many invalidations occur. This, as the size of the heap can actually become much larger than the number of models. If a cleanup is triggered, it will take some time to completely reconstruct the whole heap. This scheduler partially takes activity into account: models that are scheduled at time $+\infty$ are simply not taken into account and do therefore not influence the complexity.

4. The **dirty heap** is identical to the activity heap, but does not perform periodic cleanup. In the worst case, it is possible for the heap to grow larger at every timestep, making the time and space complexity unbounded.
5. The **heapset** scheduler still contains a heap, though it does not contain the elements themselves, but only the time at which they transition. This time can then be used to look up the actual models in a hashmap. The size of the heap is minimized by only including times, thus (time-)colliding models do not increase its size. Note that this scheduler takes advantage of the efficient dictionary (a kind of hashmap) implementation in CPython.
6. The **no age** scheduler is similar to the heapset scheduler, but without an age field, thus making it non-general. This allows slightly simpler comparisons internally.

Each of these schedulers has specific cases where it excells. For example the *Minimal list* when lots of reschedules occur, or the *Heapset* when only a small number of reschedules occur at each iteration.

In Figure 1, three different situations are tested and all (applicable) schedulers are compared. It becomes clear that noticable speedups can be achieved by choosing an appropriate scheduler. There are also variations in memory usage, though these are not shown here.

The idea of using a specific data structure for specific situations was also used in for example Meijin++[9], where the data structure 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 such 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 is worthwhile if the model has either variable behaviour, or if the user simply has no idea which scheduler to use.

2.4. Modular termination condition

Another specific feature of Py(P)DEVS is the possibility to use a complex termination condition instead of a a simple termination time. Most other simulators only allow the simulation to halt at 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. Actual simulation speed is not improved, but it offers the possibility of halting 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.

3. PERFORMANCE

Even though Py(P)DEVS is written in the interpreted language Python, one of its new focal points is performance. The first version of PyDEVS was among the slowest DEVS simulators available, mainly due to the simple simulation procotols and the naieve implementation of the abstract simulator, without optimizations.
The complete simulation algorithm was revised and the complexity was drastically reduced. The most invasive optimisations are the use of different schedulers for different types of models and the use of direct connection[2]. Several other optimisations were adopted from [7].
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, though it can alter the complexity by a constant factor.
To put the choice of implementation language in perspective, we use the DEVStone[5] benchmark with many collisions. We compare our performance to `adevs`[8], which is currently one of the fastest available DEVS simulators and is written in C++. Since `adevs` implements the Parallel DEVS[4][1] formalism, we compared it to our parallel version of PyPDEVS.
The DEVStone model is rather artificial, though it shows the complexity in the number of models in situations with many collisions. The results shown in Figure 2 were obtained using an *Intel i5-2500, 3.30GHz with 4GB main memory*.
Considering that `adevs` uses a compiled programming language, PyPDEVS compares very favourably when the `PyPy` interpreter, which has JIT capabilities, is used.
The normal `adevs` benchmark was performed without any

[1] Technically, it is the DynDEVS formalism, though the difference does not matter here

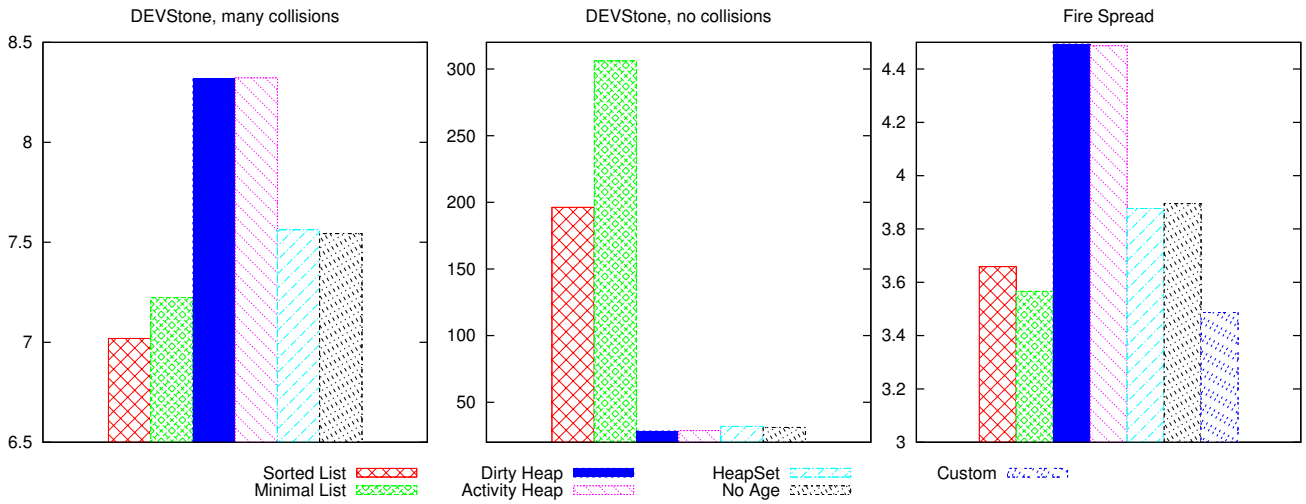

Figure 1. Scheduler comparison

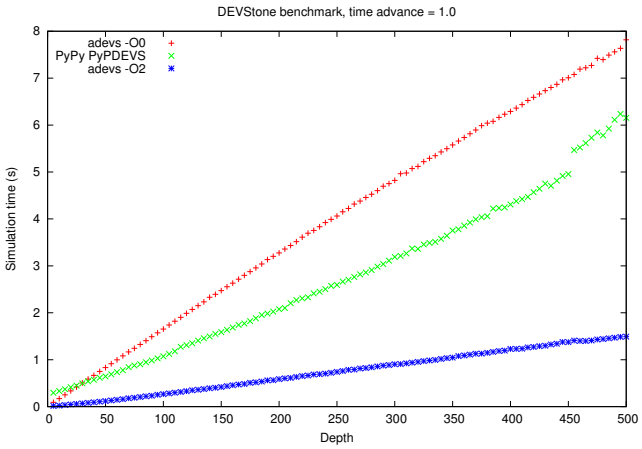

Figure 2. DEVStone comparison between PyPDEVS and `adevs`; PyPDEVS using the *minimal list* scheduler

compilation flags, thus disabling any compiler-induced optimisations. When `adevs` was compiled with optimisations (`gcc -O2`), `adevs` was again fastest. However, the `PyPy` JIT is still work in progress, so it has 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 happens at runtime), and several parts of code were never translated, resulting in normal *interpreted* execution which is slower than CPython interpration.

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 *minimal list* scheduler is the ideal scheduler due to its low time complexity that is independent of the number of collisions. *Adevs* always uses the same scheduler, with the result that it performs relatively bad in this situation. Of course, models that are a better fit for the *adevs* scheduler will again run faster on *adevs*, even if no optimization is performed.

Note that there is some slight jitter in the *PyPy* timings, even though the results are averaged over five simulation runs. This is caused by the JIT that compiles the Python code at run time only after it is executed a few times. Furthermore, the JIT causes some slight deviations in short simulations due to the warmup time. Notably, after 450 models, the PyPy garbage collector starts interrupting our simulation, causing massive slowdowns.

PyPDEVS has several features which `adevs` does not have, such as the possibility for scaled realtime simulation and several tracers. 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. In contrast, the `adevs` simulator and the model are compiled together into a single executable. Another advantage is dynamic typing: in PyPDEVS it does not matter what kind of messages are passed, whereas in `adevs`, all messages have to be of the same type (though inheritance can be used).

4. RELATED WORK

The idea of modular design is also present in JAMES II[6]. Unlike JAMES II, 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[1]), minimal list (`CD++`) and a dirty heap (`vle`). 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` is another DEVS simulator written in Python that supports realtime simulation, though it only supports threads.

5. CONCLUSION AND FUTURE WORK

We presented a new version of PyDEVS, a DEVS simulator written in Python, compliant with the Classic DEVS specification, and PyPDEVS, a Parallel DEVS simulator. 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 in several situations. Python(P)DEVS uses Python as its implementation language, allowing for highly readable and maintainable code in both the simulator and the models.

Future work will further develop PyPDEVS, by providing a distributed 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 computation nodes using Time Warp optimistic synchronization. This implementation will again focus on performance whilst maintaining compliance with the specification.

Additional functionality is also planned for the Py(P)DEVS simulator, such as reinitialisation, support for dynamic structures and nesting of multiple simulations. Due to the importance of the scheduler in defining the complexity of the simulation, the polymorphic scheduler will be benchmarked and tweaked further.

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

[2] 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.

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

[4] 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.

[5] Ezequiel Glinsky and Gabriel Wainer. DEVStone: a benchmarking technique for studying performance of DEVS modeling and simulation environments. In *Proceedings of the 9th IEEE International Symposium on Distributed Simulation and Real-Time Applications*, DS-RT '05, pages 265–272, Washington, DC, USA, 2005. IEEE Computer Society.

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

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

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

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

[10] 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.

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

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