PythonPDEVS: A distributed Parallel DEVS simulator

Yentl Van Tendeloo[†] **Hans Vangheluwe**[†,‡]
Yentl.VanTendeloo@uantwerpen.be hv@cs.mcgill.ca
[†]University of Antwerp, Belgium
[‡]School of Computer Science, McGill University, Canada

ABSTRACT
We extend PythonPDEVS, our modular simulator for the Parallel DEVS formalism, with distributed simulation using optimistic synchronization based on Time Warp. Modularity is maintained, with the addition of several new components useful for distributed simulation. The PythonPDEVS simulator supports, among others, model migration, modular allocation strategies, and distributed termination conditions. Python's introspection capabilities are used to provide default state saving and message copying. Domain-specific hints, encoded in a PythonPDEVS model, are exploited by the simulator to improve performance.

Author Keywords
Simulation; Python; Time Warp; Distribution; Parallel DEVS; Performance

ACM Classification Keywords
I.6.7 SIMULATION AND MODELING: Simulation support systems; I.6.8 SIMULATION AND MODELING: Types of simulation

INTRODUCTION
PythonPDEVS (a.k.a. PyPDEVS) belongs to the Parallel DEVS [4] family of simulation modelling languages and is grafted on the Python programming language. Python is a strongly typed, interpreted, object-oriented programming language.

With the growing demand for computing resources by modern simulation applications, the need for efficient simulators increases. For this reason, Parallel and Distributed Simulation becomes necessary. Apart from the performance improvements, the state space of current large-scale models becomes too large to represent in a single computational node. Distributed simulation allows us to split up this state space over multiple nodes.

This paper describes the addition of distributed simulation to PythonPDEVS [14]. We extend its modular design to accomodate for distributed simulation. PythonPDEVS can be downloaded from **`http://msdl.cs.mcgill.ca/projects/DEVS/PythonPDEVS`**.

Distributed simulation is performed using optimistic synchronization, based on the Time Warp [8] algorithm. We opted for optimistic synchronization, instead of conservative, as the user is not required to provide explicit lookahead information. State saving methods have to be provided, though a default is provided thanks to Python's introspection capabilities. Thanks to the use of garbage collection, there is also no need to override `new` and `delete` operations. A more elaborate comparison between both optimistic and conservative synchronization protocols can be found in [6].

Time Warp is an optimistic synchronization protocol, as it assumes that no causality errors will occur. As a consequence, no blocking will occur during simulation, in contrast to conservative simulation. Should a causality error occur, caused by a straggler message (a message arriving in the "past" of the local virtual time) [6], all computations up to that point in time are rolled back. Time Warp thus needs some way to roll back to an earlier point in time and invalidate all its actions after that point in time. Its fixed overhead consists of saving all states and all messages exchanged between two Time Warp executives, to allow for rollbacks. The variable overhead dependens on the number of rollbacks, which is dependent on the model (use of implicit lookahead), and the hardware being used (*e.g.,* latency and speed of computation).

It is possible to place an upper bound on the length of a rollback, by computing the *Global Virtual Time (GVT)*. The GVT is defined as the lowest timestamp currently present in the complete simulation. As such, no rollbacks to a point in simulated time before this timestamp will ever be required.

The remainder of this paper is organized as follows: Section DISTRIBUTED ARCHITECTURE presents the extended modular architecture of the simulator. Section TECHNIQUES elaborates on its functionality and performance improvements. In Section DOMAIN INFORMATION, domain information is used to enhance some of the most time-consuming algorithms. Section PERFORMANCE evaluates performance using synthetic benchmarks. Section Related work explores related work and Section CONCLUSION concludes the paper.

DISTRIBUTED ARCHITECTURE
This section briefly discusses the architecture of our major contribution: an implementation of Time Warp with support for (modular) model migration, and a modular model allocator.

Time Warp modifications
Some slight changes to the Time Warp algorithm were made, for better compliance with the DEVS formalism. These changes are mostly based on the ideas presented in [9]. All atomic DEVS models on a single node are grouped and work under the same Time Warp Executive. As a result, only a single Time Warp executive needs to run per simulation node.

This has several advantages:

1. Intra-node messages, which are quite frequent if a good allocation is chosen, can take a much faster path than inter-node messages. There is no need for message saving during sending and receiving, and no checking for straggler messages is required.
2. The core simulation algorithm remains unchanged. Every node simply performs a sequential Parallel DEVS simulation, but some logic is included to check for network messages and to handle stragglers.
3. There is no issue with multiple threads (one for every atomic model) that need to be scheduled, as this is done explicitly by the DEVS formalism.

The following disadvantages apply:

1. Straggler messages cause larger rollbacks, as potentially unrelated models are rolled back too.
2. Some optimization algorithms presented in the literature (see [6] for an overview), become harder to implement as we deviate further from the base Time Warp algorithm.

Model Migration

To allow for load balancing (over computational nodes), PythonPDEVS supports the migration of atomic DEVS models during simulation. At this time, the user is required to manually specify the migrations to perform.

The following steps are taken when performing a migration from node A to node B:

1. **Lock** both nodes. To prevent both nodes from progressing any further in simulated time, both nodes have to be locked.
2. **Roll back** both nodes to the *GVT*. This is an optimization that minimizes the amount of data that has to be sent over the network. As each node stores the state history of all its models, transferring a model requires a copy of the complete state history to be sent. All remote messages, both sent and received, are stored and would have to be sent too. Such histories can be arbitrary large. Rolling back both nodes, the state history and sent message queue will be emptied.
3. **Transfer** the model's current state and message queue. Due to the rollback, only the current state has to be transfered. All received messages, including anti-messages, need to be forwarded too. The sent message history does not need to be send, as it will be empty. Afterwards, the model's state and its received messages can be removed from the sending node. The newly received model is taken into account in the receiving node's scheduler and removed from the sending node's scheduler.
4. **Notify** all other nodes of the change. This is required to update the routing tables on every node. Nodes only need to take the new location into account from now on. They can assume that all previously sent messages will arrive at the correct node.
5. **Unlock** both nodes. From now on, all nodes will send messages for the migrated model to node B instead of to node A. Node B will process the new model like any of its other models.
6. **Forward** messages to node B if necessary. Due to asynchronous messaging, some messages might still be in the network during the migration of the model. Therefore, A might receive a message destined for a migrated model. In this case, node A does not have to perform a rollback, but has to forward the message to node B. This is only required for transient messages, as future messages will have the correct destination set.

All nodes, except for nodes A and B, will continue simulation throughout the migration, without blocking.

The component selecting the models to migrate, is a modular component. Right after every GVT computation, the component will be invoked with the current simulation time. The component must return a dictionary, a kind of hashmap in Python, containing some models (as key) and their new destination (as value). The simulation kernel ensures that the requested migrations are performed.

By default, we have included a general purpose migration component that parses 3-tuples of the form $\langle time, model, location \rangle$. As soon as simulated $time$ is reached, the model $model$ will be migrated to node $location$. The user might want to extend this if the migrations have to for example be performed conditionally.

Model Allocation

The PythonPDEVS formalism requires the user to manually provide the coupled model with a compute node, which will be responsible for that submodel. For example, `self.addSubModel(MyModel(), 2)` will place the model created by `MyModel()` on node 2. As a result, related or unrelated atomic models can be assigned to the same node, depending on the user. Of course, whether or not this is optimal, is up to the user performing the allocation. At this time, the user is again required to manually specify the desired allocation.

The main disadvantage of this approach is that the location of all models must be specified in the code. This clutters the model and means that a sequential model has to be altered before it can be distributed. For simple models, this is unlikely to be a problem, but for more complex models, allocating might be much harder.

One solution to this problem is to use the full functionality of Python in the model's constructor, allowing for *e.g.* function calls or library imports. While it moves the problem to a central point in the model, it is not elegant. The model, describing behaviour, should not be concerned with its distribution. The ideal model distribution is often a simulator detail, as it depends on *e.g.*, rollback cost, forward simulation cost and cost of network messages. Detailed insight into simulation algorithms is required, so these choices do not belong in the model.

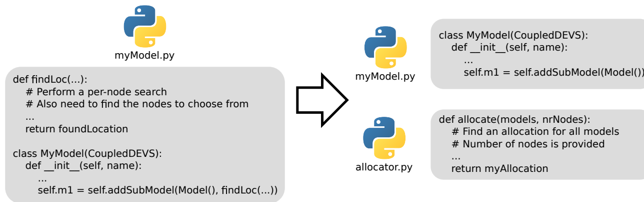

Figure 1. Allocator allows for simple distribution.

Our solution is to introduce a separate, modular allocator component. During model construction, the node on which models will be simulated can be ignored. Eventually, the allocator is called to perform the allocation using knowledge about the entire model. The component can do whatever it wants, from a few simple Python statements, up to a full-blown other DEVS simulation. The component can be specialized for the domain and simulation algorithms being used. The allocation *strategy* is likely domain-dependent, instead of model-dependent. Therefore, similar models can use the same allocator, without any change to the model. As the model is unchanged from that used for sequential simulation, it is guaranteed that no errors are introduced when making the transition from sequential to distributed simulation. Should multiple allocation strategies exist, trying them out becomes simpler as only the allocator module needs to be replaced.

Figure 1 shows how distribution code is stripped from the model. When a different model within the same domain is simulated, the `optimisiticAllocator.py` file can be re-used. If the same model needs to be simulated with different allocations, the file `optimisticAllocator.py` can easily be replaced.

TECHNIQUES

With the introduction of distributed simulation, several parts of the simulator had to be altered. Other parts are new, as these are only necessary in distributed simulation. The main challenge is to deliver identical simulation traces for sequential and distributed simulation to the user.

Direct connection

Direct connection [2] (between atomic models) removes all coupled models from the routing path, as direct connections between source and destination of a multi-level connection are created. The main advantage of direct connection is that messages do not pass through coupled models. Direct connection was already implemented in the sequential version of Python(P)DEVS. In a distributed simulation however, direct connection will have to cope with connections passing through multiple computation nodes. Moreover, a connection might just be passing through a coupled model, which can be allocated to a different node than the source and destination of the connection. As a result, both the number of network messages and the latency between source and destination is reduced. With direct connection, all atomic models become direct children of a new root coupled DEVS model. There is thus no distinction between subtrees, making very fine-grained allocation and migration possible, as mentioned previously. The normal situation, hosting a subtree, is simply a special situation of our more general support.

Message grouping

For distributed simulation, it makes sense to group related messages into a single message. This does not only reduce overhead and latency, but may also reduce the number of rollbacks, when sending multiple related messages. Grouping of messages is used in 4 situations:

1. **Simulation messages** from different models, destined for the same model, are grouped. Grouping only happens at the model-level, instead of at node-level, as model migration would otherwise become more involved. As one grouped message has only a single ID, anti-messages for these messages are implicitly grouped too. Message overhead in the incomming message queue and outgoing message history is also reduced.

2. **Anti-messages** for the same node are grouped. Thanks to direct connection, no information about the source, destination, content, ... is required. Anti-messages are reduced to a single message, containing a list of IDs to invalidate. Anti-messages were already implicitly grouped, though now messages from different simulation times are also grouped.

3. When **migrating** multiple models between the same nodes, all model states are grouped into a single call. This does not make much of a difference in terms of overhead or latency, as model states are generally large. However, it might be possible to use compression on the states when transferring them across the network. Compression profits from similar data being passed, which can significantly reduce the amount of network traffic required. Whether or not there is any potential gain in doing so depends on the hardware being used.

4. **Irreversible operations**, such as I/O, require some special attention in Time Warp as they cannot be rolled back. These operations are sent to the controller as a string, which is executed as soon as the GVT has progressed sufficiently far. Sending these messages to the controller can take some time, as many may occur simultaneously. An example is a tracer, which outputs some data (possibly to a file) for every transition that is being performed. Grouping all such calls in a single batch can have a significant impact, certainly when latency is fairly high. As an added advantage, invalidations of these I/O operations are automatically grouped.

Termination condition

Complex termination conditions were also previously supported in Python(P)DEVS [14]. With distributed simulation, distributed termination conditions are supported, up to a user-customizable degree. The tradeoff is between accuracy (of the exact time at which the simulation is terminated) and performance.

Two possibilities, or a combination thereof, are supported:

1. **Force** models being read to run at the controller. This is of course not a real distributed termination condition, as it makes sure that no distributed states are accessed. If a migration is performed that would move the model away from the controller, it is silently ignored. This is by far the simplest way and is recommended if the state is checked frequently. However, frequently used intra-node connections might be reduced to inter-node connections. This results in excessive rollbacks and significant message passing overhead, with lower performance as a result.
2. **Pull** the state of remote models. When reading the state of remote models, the simplest way is to request the state of the model. Due to the use of Time Warp, the distributed state is by no means guaranteed to be consistent. When a model state is requested, the request should contain the time for which the state is required. The model processing the request will traverse its state history and retrieve the state at that point in time. If this time is not yet reached, the call will block until the requested state becomes available. When the model is rolled back, the previously sent state becomes invalid too. A rollback will propagate to the controller, which has to check the termination condition again. Note that termination conditions are only executed at the controller, right before a simulation step. A state change of a remote model will go undetected, as this does not cause a simulation step at the controller. It is possible for a distributed simulation to run longer than a sequential one, as termination is only detected at the next simulation step at the controller. This conservative approach guarantees that all behaviour is computed upto at least the time of termination (but possibly beyond it).

Push-based signalling, where a state change is signaled to the controller, is not implemented. Its main disadvantage is the potential flooding of the network and the controller in the presence of frequent state changes.

So the ideal approach is based on the model being simulated: if states are read out frequently, it is advised to run that model on the controller. On the other hand, if the state is read out infrequently, or if moving the model to the controller causes too many rollbacks, a pull-based solution is prefered.

Pseudo-random number streams

Pseudorandom number streams are of great importance in discrete-event simulation. While pseudorandom numbers can easily be generated using, *e.g.*, the `random` module in Python, these are not guaranteed to be deterministic in a distributed simulation [13]. The primary problem is that a rollback does not roll back the internal state of the random number generator. Actually, the Parallel DEVS formalism does not even guarantee that colliding transition functions are processed in a deterministic order. While this offers possibilities for simulator optimization, it makes a deterministic pseudo-random number stream hard to achieve.

The PythonPDEVS simulator solves this problem by offering a random number generator class to the user. This class has its own state and an instance of it should be saved *in the state* of every model that requires it. Random numbers are then generated using a random number generator that is in the scope of the model only.

As the class is not part of the simulator, it should only be considered a utility class. The user is free to use other methods, as long as they can generate deterministic random number streams in the presence of a non-deterministic order of transition functions and rollbacks. The class is only a wrapper around the `random` module, but it configures its seed every time a random value is required. This new seed is determined by another random number and is saved in its state. As a global random number generator is used, possibilities for non-determinism are still present. However, PythonPDEVS does not execute transition functions in parallel, nor does it use the global random number generator itself. No changes to the random number generator's internal state will occur between the setting of the seed and the retrieval of both the random value and the new seed.

Per-node scheduler

The most important performance feature of the sequential Python(P)DEVS simulator was its modular scheduler [14]. Its rationale was that different domains have different scheduler access patterns. As the scheduler is a resource-heavy component, chosing an appropriate scheduler may have a significant impact on performance.

With distribution, different nodes host different models, which might have different access patterns too. Every Time Warp executive has its own scheduler, which is independent of the scheduler of other executives. As such, each node can have a specific scheduler, optimized for its own access pattern.
This further extends the possibilities for domain-specific schedulers. Previously, if a single model's behaviour did not fit into the domain-specific scheduler's domain, a general purpose scheduler had to be used. Now, it becomes possible to run such models on another node. All other nodes run a domain-specific scheduler, while only a single node runs a general purpose scheduler.

DOMAIN INFORMATION

As was the case in Python(P)DEVS, domain-specific hints are an option in distributed PythonPDEVS. Thanks to Python's introspection, general purpose algorithms can be used as a default for many domain-dependent operations. These general purpose algorithms are quite slow and, for maximum performance, the user might want to override them.

State saving

As rollbacks are possible with Time Warp, the simulation kernel has to keep a copy of each and every state. Several options are possible: either Python's introspection is used by default, or the user defines a custom copy method. Writing a decent custom copy method for a state is relatively difficult as unreliable state saving can cause non-determinism problems. State saving cannot be ignored, as the user likely reuses state objects, or parts of them.

The three main state saving methods that are supported are:

1. **deepcopy** uses Python's `deepcopy` library to make a deep copy of the data using introspection. Due to its low performance, this should only be used if no other option is possible (*e.g.,* unserializable state).

2. **pickle** uses Python's `pickle` library to serialize and deserialize an object. Despite using strings internally, pickling turns out to be more efficient than creating a deepcopy. This is explained by the fewer use cases (*e.g.*, no locks or threads) and an implementation in C (the `cPickle` module). As unserializable objects are unlikely to be part of the state, this is the default in PythonPDEVS.

3. The user can manually define a **custom** `copy` method for every kind of state. This has the highest potential for high performance, though defining an incorrect copy method will likely result in non-deterministic failure of the simulation. It is recommended that a custom state saving method is only defined after the state's structure has stabilized. Further optimizations are possible, such as saving only parts of the state.

Event copy

Copying of events, exchanged by different models, is one of the additional features of PythonPDEVS and was already supported [14]. The user had the choice to copy events in 3 different ways: the `cPickle` library, a user-defined copy method, or no copying at all.

As not all objects are serializable by the `cPickle` library, the user could circumvent this problem by defining a custom copy method, or not performing any copying. This is no longer possible in a distributed simulation, as all events that pass through the network have to be serialized. It is still possible to use the other options for performance reasons, though the user should be aware that all events passing through the network, need to be serializable too.

Memoization

PythonPDEVS groups a number of atomic models into a single Time Warp executive. Due to this, the scope of rollbacks is extended from a single atomic model to a complete node. This is fairly pessimistic, as a single straggler is unlikely to have such a serious impact on all models.

When performing migrations, huge rollbacks occur to ensure that the transfered state is as small as possible. These rollbacks are not caused by causality errors, so there is absolutely no impact on any other model.

Many models are therefore rolled back, without a causality error for them specifically. To counter these problems, a kind of memoization is used for the states. When a rollback is performed, the state history is not cleared, but saved elsewhere. Before a model transitions, the memoized state history is checked and its result reused if possible. The algorithm is shown in Algorithm 1. This only applies to the transition functions and not for *e.g.*, the output function. As only internode messages are saved, message copying would be required otherwise.

Algorithm 1 Transition algorithm with memoization.

$newstate \leftarrow NULL$
if memoization history not empty **then**
  if current state matches first memoized state **then**
    if internal transition **then**
      $newstate \leftarrow memoizedstate$
    else if external input values matches memoized input values **then**
      if confluent transition **then**
        $newstate \leftarrow memoizedstate$
      else if elapsed time matches elapsed time **then**
        if external transition **then**
          $newstate \leftarrow memoizedstate$
        end if
      end if
    end if
  end if
end if
if $newstate = NULL$ **then**
  clear memoization history
  execute normal transition code
else
  remove first memoized value
end if

This is not a general kind of memoization, as it is based on rollbacks and the state history. As such, it does not speed up sequential simulation, to avoid state saving overhead. Additionally, both Time Warp and memoization of every transition use a lot of resources for state saving. When both are present, we would need to distribute memory resources between both of them. Some form of management for the memoized states would have to happen too.

Domain-specific information is required to decide whether or not two states are exactly equal. This information has to be provided by the user in the form of a comparison method between different states. If no comparison method is provided, they are assumed to be inequal. Turning memoization off completely, does not cause a significant performance improvement, as states were copies for Time Warp anyway.

PERFORMANCE

PythonPDEVS's performance is evaluated for a variety of synthetic models. The impact of three kinds of domain-specific hints is evaluated using different synthetic models.

All simulations were performed on a shared cluster of *Intel Core2 6700 dual-core @ 2.66GHz* machines with *8GB* main memory, running *Fedora Core 13, Linux kernel 2.6.34*. The machines are connected through a $1Gbps$ link, with an average RTT of $0.1ms$. The number of nodes used in experiments is always indicated. These machines have *CPython 2.7.5* and use *MPICH 3.0.4* as MPI implementation with the *MPI4py 1.3* wrapper. Local simulations only use a single core of a single machine. Results are based on 5 simulation runs, with boxplots shown where appropriate, to illustrate the variability in the results.

We provide a general benchmark (PHOLD) of distributed simulation in PythonPDEVS. Afterwards, we evaluate the effect of domain-specific information for distributed simulation.

PHOLD

The PHOLD model is a Parallel Discrete Event Simulation (PDES) benchmark for Time Warp implementations [5]. As PythonPDEVS is a Parallel DEVS formalism, a Parallel DEVS equivalent of the PDES benchmark was constructed. A variety of parameters can be configured in the model, such as the fraction of remote versus local messages and computational load in transition functions.

Figure 2 shows the structure of a PHOLD model for 2 nodes. This model has a total of 4 atomic models, with 2 atomic models per node. If 10% remote messages is used, this means that model A has a 90% chance to send its message to model B, 5% chance for model C and 5% for model D. Every model starts with a single event, which is processed and forwarded to a randomly chosen other node. If a model has multiple events, events are queued until previous messages are processed. Should all models of a node run out of events to process, the node blocks until a message is received.

Figure 3 shows that speedup keeps increasing when more nodes are added. 10% of events are remote, with a computational load of $25ms$ in the transition function. A total of 2 atomic models are present on every node.

In Figure 4, the effect of the number of remote messages is shown for a 20-node simulation. With an increasing number of remote events, more messages will have to pass over the network and are possible stragglers. As the number of rollbacks becomes higher, performance is significantly reduced.

We have chosen the computational load to be $25ms$, as this signifies a model with serious potential for distribution. Of course, if the computational load for the model is relatively small compared to the Time Warp overhead, no speedup can be achieved.

Impact of memoization

The synthetic benchmark places around 800 atomic models on the same node. Only one of these models is connected to another model on another node. Each model has computationally heavy transition functions, which are rolled back when a straggler is received. Rolling back all models is clearly wasteful, as the receiving model is not connected to the others in any way. Figure 5 shows the results of memoization on 2 nodes. Only the model that received the message will have to re-do all its computations. All other models can use their memoized values, without invoking the transition function.

Impact of state saving methods

To better isolate the influence of the different state saving methods, this benchmark only times the copying of the states. We compare 3 different state saving methods: *deepcopy*, *pickle* (default), and a *custom* hand-written one. Even though it is named *custom*, it consists of a series of assignments of

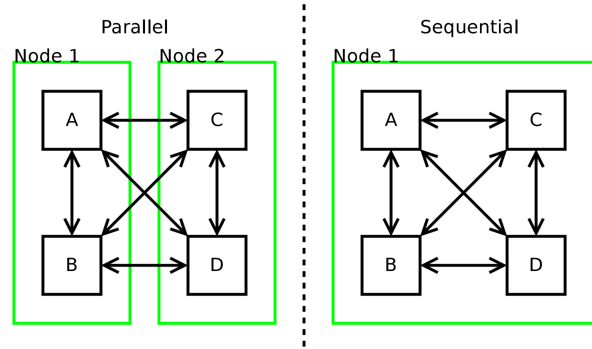

Figure 2. PHOLD model for 2 nodes with 2 models per node.

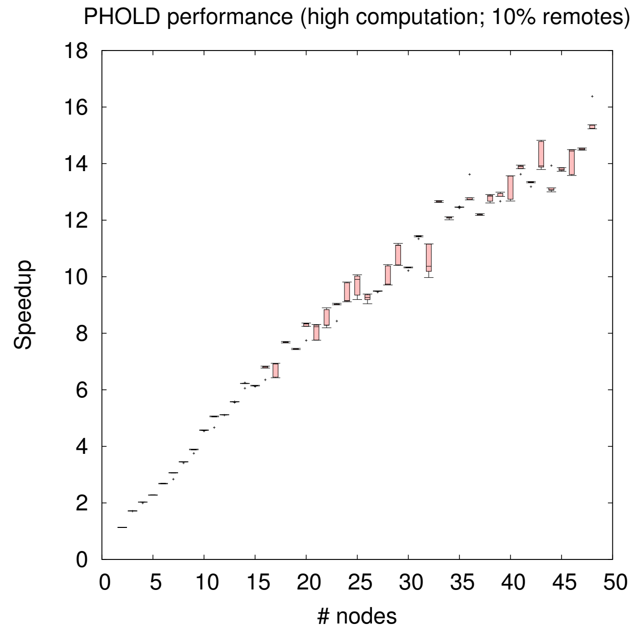

Figure 3. Effect of number of nodes with 10% remote messages and $25ms$ computational load on PHOLD.

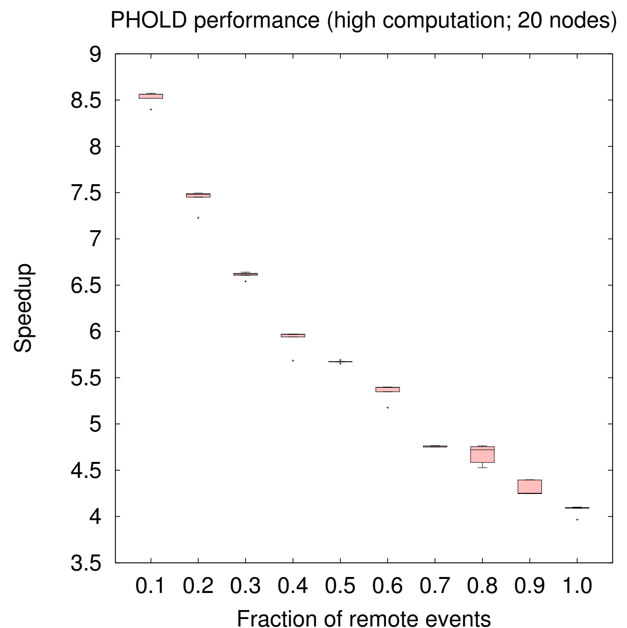

Figure 4. Effect of number of remote events for 20 node simulation and $25ms$ computational load on PHOLD.

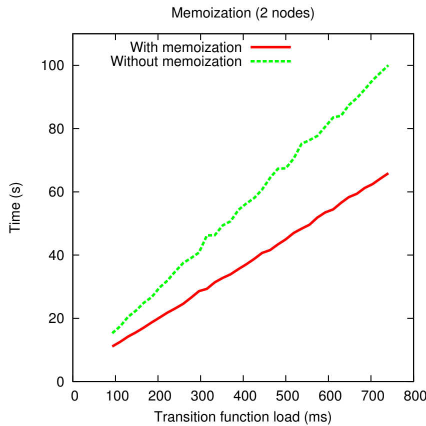

Figure 5. Impact of memoization.

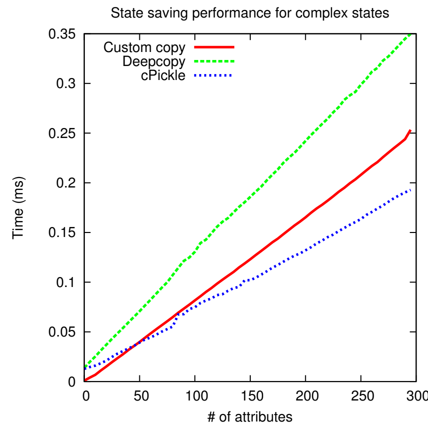

Figure 6. Impact of state complexity.

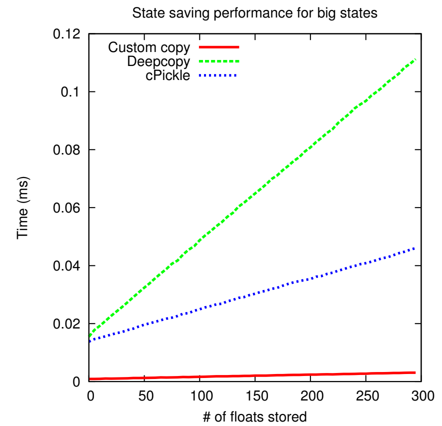

Figure 7. Impact of state size.

every attribute of the state. No further domain information is used.

The first benchmark, shown in Figure 6, increases the number of attributes contained in the state. As the number of attributes increases, all methods have to copy over more information, resulting in slower state saving. Clearly, the deepcopy method is the slowest due to its generality. Remarkably, a custom copy method is slower than the serialization (and subsequent deserialization) of the state. This is caused by the custom copy method requiring many operations to actually copy over the attributes: attributes are copied one-by-one. As this method is written in Python, this adds some overhead. The pickling method on the other hand, is written in C and only a single Python call is therefore made. This significantly reduces the overhead, even making up for generality if the state becomes complex enough. For states with less than 50 attributes, the custom copy method is still the fastest. Note though, that if it was known that only parts of the state had to be stored (*e.g.,* only a few attributes), the custom method could have taken this into account.

Figure 7 shows results for the second benchmark, where the state consists of only a single attribute. The attribute is a list, varying in size, and thus influencing the size of the state without making it more complex. These results indicate that deepcopy is again the slowest, as was to be expected. However, the custom copy method is a lot faster than pickling in this case. As the only attribute is a list, the custom copy method simply needs to make a copy of the list, for which a Python primitive function exists. Thus the serialization to string and parsing is avoided completely, just like the overhead of using Python. As a result, the custom copy method takes nearly no time in comparison to the other approaches.

Results show that the ideal state saving method is dependent on the kind of state being saved. If the state contains objects that cannot be pickled, the deepcopy method needs to be chosen manually. On the other hand, given sufficient knowledge about the structure of the state, higher performance can be obtained using a custom copy method.

RELATED WORK

PythonPDEVS is based on the PythonDEVS formalism and simulator [1], which supported only sequential, Classic DEVS [15] simulation. While we support Parallel DEVS [4], transition functions are always processed sequentially instead of in parallel, as suggested by the abstract simulator [3]. It has namely been shown that the sequential processing of transition functions is not necessarily slower than naive parallelisation [7]. Our distributed synchronization protocol is based on Time Warp [8], with the addition of some slight optimizations [6] and global Time Warp [9].

Some Parallel DEVS simulators exist that use parallel or distributed simulation. Adevs [11] for example implements a parallel simulator, using conservative synchronization. It does not include distributed simulation, thus does not solve the state-space problem, nor can it be used on a cluster. VLE [12] implements a distributed simulator (the bundled *mvle* application), but does so without any synchronization. It only simulates several models (or one model with different configurations) at once. As a result, it does not help with the simulation of a single model in a single configuration.

CONCLUSION

We added distributed simulation, modularity, and increased performance to our sequential PythonPDEVS simulator. Distributed simulation offered new opportunities for modularity, such as model migrations and model allocation.

As in the sequential version, the distributed simulator utilizes domain-specific hints when available. Distribution yields additional speedup opportunities based on such user-provided hints. Several optimizations are evaluated, showing that they can indeed have a significant impact, on synthetic models.

We are now building a library of re-usable PythonPDEVS models. This will form the basis for extensive performance evaluation on non-synthetic models in different application domains (most notably, production systems and transportation, as well as –discretized models of– physical systems).

Future work will evolve in several directions. First, migration and allocation modules require the user to manually specify the rules that should be applied. Using activity [10], the simulator may provide information on load distribution in the

model. Second, a wide variety of optimizations to the Time Warp algorithm exist. Implementing some of these optimizations might further increase simulator performance. Third, CPython uses a Global Interpreter Lock (GIL), rendering parallellism using threads impossible. A design change of the simulator is possible, which removes the use of threads and lets the simulator perform all scheduling. Alternatively, if the GIL were to be removed, as when re-coding in C++, the problem would disappear. Fourth, we plan to run our performance analysis on more realistic models.

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. Bolduc, J.-S., and Vangheluwe, H. The modelling and simulation package PythonDEVS for classical hierarchical DEVS. Tech. rep., McGill University, 2001.
2. Chen, B., and Vangheluwe, H. Symbolic flattening of DEVS models. In *Summer Simulation Multiconference* (2010), 209–218.
3. Chow, A. C. Parallel DEVS: A parallel, hierarchical, modular modelling formalism and its distributed simulator. *Transactions of the SCS 13*, 2 (1996), 55–67.
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. Fujimoto, R. M. Performance of Time Warp under synthetic workkloads. In *Proceedings of the SCS Multiconference on Distributed Simulation* (1990).
6. Fujimoto, R. M. *Parallel and Distribution Simulation Systems*, 1st ed. John Wiley & Sons, Inc., New York, NY, USA, 1999.
7. Himmelspach, J., and Uhrmacher, A. M. Sequential processing of PDEVS models. In *Proceedings of the 3rd European Modeling & Simulation Symposium* (2006), 239–244.
8. Jefferson, D. R. Virtual time. *ACM Transactions on Programming Languages and Systems 7*, 3 (1985), 404–425.
9. Kim, K. H., Seong, Y. R., Kim, T. G., and Park, K. H. Distributed simulation of hierarchical DEVS models: Hierarchical scheduling locally and time warp globally. *Transactions of the SCS 13*, 3 (1996), 135–154.
10. Muzy, A., Varenne, F., Zeigler, B. P., Caux, J., Coquillard, P., Touraille, L., Prunetti, D., Caillou, P., Michel, O., and Hill, D. R. C. Refounding of the activity concept? towards a federative paradigm for modeling and simulation. *Simulation 89*, 2 (2013), 156–177.
11. Nutaro, J. J. ADEVS. `http://www.ornl.gov/~1qn/adevs/`, 2014.
12. Quesnel, G., Duboz, R., Ramat, E., and Traoré, M. K. VLE: a multimodeling and simulation environment. In *Proceedings of the 2007 Summer Simulation Conference*, SCS (2007), 367–374.
13. Reuillon, R., Traoré, M. K., Passerat-Palmbach, J., and Hill, D. R. C. Parallel stochastic simulations with rigorous distribution of pseudo-random numbers with DistMe: Application to life science simulations. *Concurrency and Computation: Practice and Experience 24*, 7 (2012), 723–738.
14. 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.
15. Zeigler, B. P., Praehofer, H., and Kim, T. G. *Theory of Modeling and Simulation*, second ed. Academic Press, 2000.