ITM Web of Conferences **3**, 01002 (2014)
DOI: 10.1051/itmconf/20140301002


# Activity in PythonPDEVS

Yentl Van Tendeloo[1,a] and Hans Vangheluwe[1,2,b]

[1] *University of Antwerp, Belgium*
[2] *McGill University, Canada*

**Abstract.** We introduce an activity-enhanced version of PythonPDEVS, a Parallel DEVS simulator. PythonPDEVS supports both sequential and distributed simulation. Both sequential and distributed variants of PythonPDEVS exploit information about computational activity to reduce simulation time. DEVS models can be augmented by the user with domain-specific information about computational load. This information is used by the simulator to improve performance.

## 1 Introduction

PythonPDEVS is a Parallel DEVS[1] simulator[1] written in the interpreted, high-level, object-oriented programming language Python. The simulator supports both sequential and distributed simulation, the latter using Time Warp[3] for optimistic synchronization. Both variants of PythonPDEVS are aware of the notion of computational *activity*[4]. Due to the many differences between sequential and distributed simulation –and mainly due to the way they can exploit activity– they will be discussed separately. The rationale behind and quantitative speedup results of our approach will be given for both variants. Our main contribution lies in the distributed simulation, which allows the user to specify when a model is *active*.

The remainder of this paper is organized as follows. Section 2 presents two different views on activity as well as the rationale for why we chose one over the other. Section 3 presents our use of activity in sequential simulation. Section 4 presents the additional opportunities for speedup offered by PythonPDEVS, using activity in distributed simulation. Section 5 explores related work and Section 6 concludes the paper.

## 2 Multiple activity definitions

Our goal is to provide a modeller with the means to encode knowledge about *predicted activity* [5]. This prediction will be exploited by the simulation/execution engine. Activity can be interpreted in many ways, depending on the particular resource use one wishes to focus on. Energy, time, and memory are all resources that can be at the basis of activity definitions. We will focus on the time resource. From the *communication load* perspective, a model has a high activity if it generates many

[a] e-mail: yentl.vantendeloo@student.uantwerpen.be
[b] e-mail: hv@cs.mcgill.ca
[1] PythonPDEVS supports different DEVS variants such as Classic DEVS[2]. This aspect is not elaborated here.

events in a fixed time window (i.e., has a high event rate). From the *computational load* perspective, a model has high activity if its *transition functions* take a long time to process.

Figure 1 illustrates the orthogonality of these two definitions: with the first definition, models 2 and 4 are highly active, whereas models 3 and 4 are highly active with the second definition. The distinction arises as models may send many output messages without incurring a heavy computational load. Our work will focus on the second definition for several reasons.

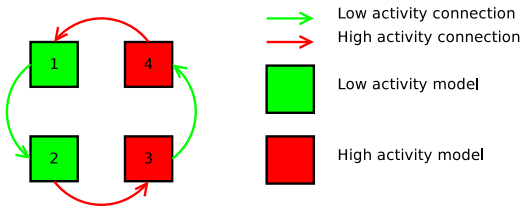


**Figure 1.** Small model illustrating both definitions

The main reason is that the event rate is less interesting in a distributed simulation due to the use of Time Warp. Using the *communication load* activity perspective in Time Warp distributed simulation would incur a significant simulation overhead. Since activity is being used to increase performance, such an overhead is unacceptable.

Apart from these two *general* methods, the modeller may also specify a custom concept of activity, based on for example the state of the model, or the changes thereof. To allow for user-provided definitions, possibly inspired by domain-specific knowledge, the computational load perspective can be generalized to use such definitions if they are available. Note that, if desired, the computational load perspective may encode *event rate* as an approximation of computational load. Such an approximation is appropriate in the case of high event rates and low computation in transition functions.

We chose the *computational load* perspective for the following reasons:

1. Load tracking is faster, as it does not depend on the number of events, but only on the number of transitions.

2. Event tracking is complex in a distributed simulation, due to events being rolled back. If such rolled back events were incorporated in the results, this would skew the results significantly depending on the amount of rollbacks. If they are not incorporated, one gets artificial results as the events did actually occur.

3. Distributed simulation is best split up based on load, certainly in a Time Warp implementation, where significant problems can occur if the load is not evenly spread over the nodes. These problems include long rollbacks, running out of memory and excessive inter-node communication.

4. It can simply be extended to allow user-defined, domain-specific definitions of activity.

## 3 Activity in Sequential simulation

Sequential simulation will be discussed first, as it presents some of the same opportunities for activity-based optimization as distributed simulation. Both sequential and distributed activity-based approaches are the same, though the latter adds substantial implementation complexity. Distributed simulation offers some additional opportunities for optimization, as will be discussed in Section 4.

### 3.1 Rationale

In sequential simulation, activity may be used to ignore models that are *inactive*. Such inactive models could for example be those whose `timeNext` is equal to $+\infty$. The use of quantization of continuous models [2] is pertinent in this context as it prevents excessive communication (and thus invocations of the external transition functions), at the cost of reduced accuracy. An example of this is a fire spread model, where cells that do not have any changes in their temperature are computationally inactive. The simulator may hence ignore these inactive models, in the limit reducing the complexity of the simulation by reducing it to computation of active models only. An example can be seen in Figure 2, where a grid of $n \times n$ cells is used. A traditional simulator has to evaluate $O(n^2)$ models. With activity, this can be reduced to $O(r)$, with $r$ the radius of the fire front, as due to the nature of the forest fire domain, only the fire front requires computation.

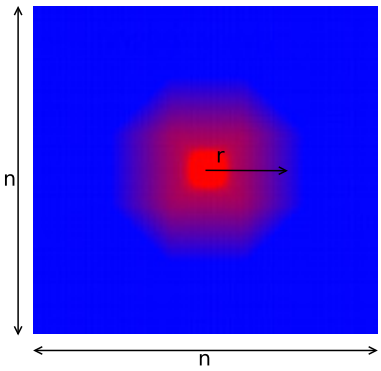


**Figure 2.** An example of the use of activity in a fire spread model.

### 3.2 Implementation

Ignoring models with a `timeNext` equal to $+\infty$ is already implemented in several DEVS simulators, such as *vle*[6] and *adevs*[7]. A dedicated activity-based simulator implementation for this type of (in-)activity is thus not necessary. An appropriate *scheduler* suffices to filter out inactive models. The scheduler is at the heart of any discrete-event simulator's main simulation loop and determines the next (sub-)model(s) of which the internal transitions should be processed.

Since PyPDEVS offers a *modular scheduler*, it is possible for domain-specific information to be incorporated in such a scheduler. In the fire spread example, we know that if a cell was completely burned down, it will *never* become active again. Such an inactive cell can thus be completely removed from the simulation, for example by removing it from the scheduler. The modular scheduler can also collect all required (activity) information from the model, right before scheduling, and use it to reduce its own complexity. This allows the scheduler to use its own (possibly domain-specific) concept of activity, which may deviate from our standard interpretation of activity, if needed.

### 3.3 Performance

To investigate scheduler-based performance optimizations, the fire spread model is simulated using different schedulers:

1. **Minimal list**: simply iterate over all models and select the one with the minimal time. This scheduler will always run with $O(n)$ complexity.

2. **Normal heap**: put all models and their `timeNext` in a heap and use invalidation. This scheduler will have a complexity $O(log(n))$ in the best case, when nearly no invalidations occur. Such usage of heaps is often done in schedulers, as schedulers only require access to the first (sorted) element.

3. **Activity heap**: the same as the normal heap, but only insert models with a `timeNext` different from $+\infty$. This explicitly uses a notion of activity, as it has a *filtering* step. Due to the check for

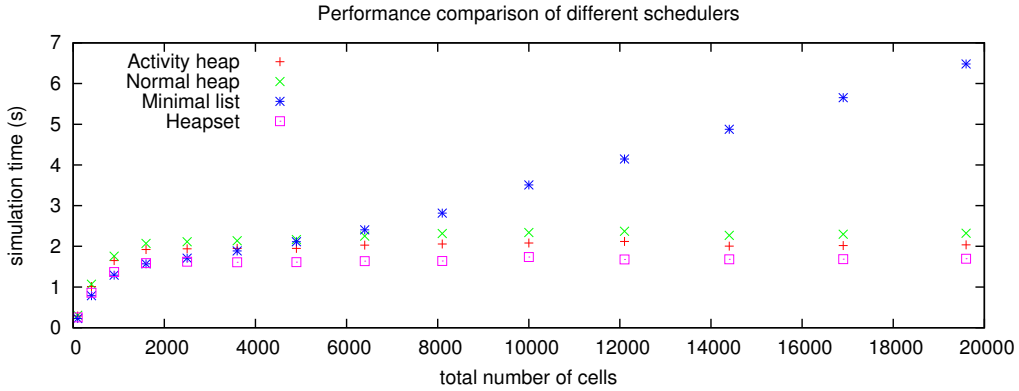


**Figure 3.** Performance comparison for different schedulers, some of which use activity.

inactive models, this scheduler will have a complexity of $O(log(a))$ in the best case, with $a \leq n$ the number of *active* models.

4. **Heapset**: build a heap containing only the `timeNext` and a hash map to map the `timeNext` to the models. This implicitly uses a notion of activity as all *inactive* models will be mapped to the same `timeNext` entry. The complexity of the scheduler itself is thus not increased as the number of entries at the same `timeNext` increases. The complexity is equal to $O(log(k))$ on average, with $k \leq a$ the number of distinct `timeNext` values which are scheduled.

Note that these complexities are for scheduling and retrieving a single model. If multiple models are (re-)scheduled simultaneously, some of the above schedulers might be capable of more drastic optimizations.

Performance results are shown in Figure 3. It is clear that the use of activity may yield significant speedups for coupled models with a high number of inactive atomic models. Additionally, the *heapset* scheduler is the fastest by a slight margin as the fire spread model has all of its transitions at the same time. This entirely removes the need for heap operations. The simulation's complexity is however not $O(1)$, since the number of transitions is still $O(r)$.

The simulation runs that used activity stopped using additional computational resources as soon as no more *active* cells were added. At about 2000 models, all new models stayed inactive and were therefore not taken into account at all in the simulation. In other words, $n$ kept increasing linearly, whereas $a$ stayed constant due to these additional models being inactive. This shows how the use of an appropriate problem/domain-specific scheduler removes all *accidental computational complexity*. Only the *essential computational complexity* remains: the computation of those cells (in the fire front) that are actually active. This is an example of the appropriate use of date structures to encode the *activity region* (the set of active models)[5].

The difference in complexity between the *normal heap* and *activity heap* is not that visible, as both still have logarithmic complexity, though for another term. It is clear however, that the activity heap always performs faster for this model.

Note that the above results are from the simulation only and do not include model construction. Quantization was also used to prevent communication from spreading too fast and thus activating all models (almost) at once. Quantization makes sure that only significantly different messages are passed. Since the surrounding temperature only has a slight influence on the cell temperature, sur-

rounding cells will have a smaller increase (or decrease) in temperature as they get further from the source.

Without quantization, cells will *always* receive (even minuscule) temperature changes, update their own state and subsequently create new output messages to its neighbors. In other words, every simulation step will activate all models that are neighbors of the currently active models. A model with $n^2$ models will thus be completely active after $n$ simulation steps. With quantization, a model will not pass too small temperature changes and thus fewer models will become active. The actual influence of quantization is dependent on the used threshold and the speed at which cells change their temperature.

## 4 Activity in Distributed simulation

The optimizations to the simulation algorithm described above remain in effect for distributed simulation. Distributed simulation offers some additional opportunities for performance optimization when combined with activity information.

Since we use *Time Warp*, there is always the possibility for rollbacks (of transitions and events sent) due to Time Warp's optimistic synchronization. To solve this problem, the activity of a transition is saved together with the state. In the most basic (and default) case, this *activity* is simply a float that contains the time the transition function took (in seconds). Custom activity can save arbitrary types, though simple types are recommended. State saving is already part of Time Warp to support rollback, so the simulator architecture does not require any changes. We do not take into account the *lost computation* due to roll backs, mainly because relocations could alter these rollbacks in an unknown way. If the computational load is evenly distributed, the models should also advance their current simulation time at approximately the same rate. When the deviation between the simulation times on the distributed computation nodes is low, a rollback will be rather small and neglecting lost computation is acceptable. When the computational load is not evenly distributed, neglecting the lost computation might not be acceptable. In such a case, Time Warp-based distribution is either not suitable or initial allocation as well as load balancing need to be modified.

### 4.1 Rationale

The main optimization that is now possible is to perform migrations, of models to different computational nodes, based on the activity currently exhibited by the model (and ideally, based on the predicted future activity of models). While it may be possible for the modeller to determine the best allocation at the start of the simulation, the behavior of the model, and hence its computational load, can change drastically during the the course of a simulation. Such changes could turn an initially good (or even optimal) allocation into a bad allocation. One solution to this problem is the use of *static migration rules*. These require detailed insight into the behavior that the model will exhibit and tedious re-work, often based on trial and error, each time the model is altered.
Additionally, a model's computational load may be highly sensitive to (i.e., vary drastically with) the initial configuration values. Activity may provide hints to a *migration component* of the simulator, which can then attempt to equalize the amount of computational activity on the individual simulation nodes. This gives us *load balancing* based on the current computational load of models. The modeller may provide his own definition of activity, possibly different from computational load, which can provide a *prediction of future activity*.

As a synthetic example, take the model in Figure 4 which contains atomic models arranged in a ring. Each of these models will pass on its input value to the next model, after a certain delay.

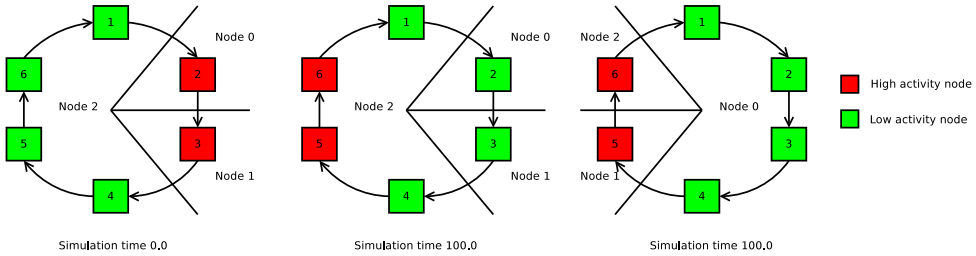


**Figure 4.** An example of a model with varying activity, with relocation applied.

The internal transition function does however contain some significant computation, depending on the value of the event that was received. As a result, a small section of the ring requires significant computation, while the rest of the ring requires nearly no computation. Simply deciding on a good enough initial allocation is clearly insufficient as the load migrates through the ring as the simulation progresses.

A migration component of the simulator that uses information about the activity can simply track which models are highly active and relocate them to dedicated computation nodes. The other models, which have nearly no computations, can then be put on a single node. As simulation progresses, this region of *low-activity* models will also migrate, leading to an ideal distribution of computation, most of the time. This relocation is illustrated in Figure 4.

### 4.2 Activity tracking

Activity tracking is the simplest way of gathering activity information. The duration of the *transition* functions is added up and used as the actual activity within a specified time window (the *horizon*). We used the difference in wall clock time before and after the transition functions, since obtaining other timing metrics incurs a significantly higher overhead. Such a penalty is unacceptable as our intention is to increase simulation performance.

The advantage of this method is that it works in every situation and is thus very general, as it only uses the modules provided by Python itself, is platform-independent and the spent wall clock time is independent of the internal structure of the model. Sadly, it is relatively inaccurate due to possible thread switching: if either Python or the operating system perform a thread switch during the computation, the results will be artificially high.

Furthermore, due to the dependency on unpredictable load conditions of the system and the inability of the activity tracking to counter this, the measured times are not the same across different experiment runs, making then not repeatable. This may pose problems when debugging the performance enhancements. It is also possible that this information comes too late as the load might still significantly fluctuate after the activity information was gathered. This technique is also susceptible to intermittent load fluctuations. A situation might arise in which activity is found to be badly distributed, though the user *knows* that the current allocation is a sufficiently good one. As performing relocations takes time, it might be counterproductive to do the relocation for a slightly better allocation, only to undo that relocation in the next step because the detected load was only temporary. User information could thus be used to *hint* to the migration module that no migration is necessary at this point in time, even though the current allocation might seem suboptimal. Increasing the size of the time window might address these weaknesses to some extent. However, a larger window means that the reaction to activity changes will be detected later. As a result, a lot of time could be spent in an allocation that is no longer optimal, and some migrations might have been performed sooner.

```
preValue = model.preActivityCalculation()
model.inputValues = deepcopy(model.inputValues)
model.state = model.transition()
activity = model.postActivityCalculation(preValue)
```

**Listing 1.** Pseudo-code in Python indicating the place of both activity calls

### 4.3 Activity prediction

Activity prediction addresses the problems caused by the generality of activity tracking, This is solved by fetching the activity that should be used from the model itself. This is therefore an activity-based *augmentation* of the DEVS formalism, allowing models to provide hints to the simulator about their (current and anticipated) activity and on how they should be distributed. This breaks down the clean boundary between model and simulator, but enables higher performance.

To reduce the amount of modification necessary to our simulator, we use *activity prediction* which defaults to *activity tracking* if the user does not provide activity information. This is done by having a *pre-transition* call and a *post-transition* call to the augmented model. The augmentation consists of the `preActivityCalculation` and `postActivityCalculation` methods that get added to the model. Pseudo-code for the relevant part of the simulation algorithm is shown in listing 1.

With these two calls, it is possible for the modeller to access all information that might be necessary. Activity tracking requires these two calls, as it has to subtract the time *before* the transition from the time *after* the transition. Custom activity definitions might also use this, for example to determine the difference between the state before and after the transition. Communication between both calls is required to pass this intermediate result. Therefore, the returned value from the *pre-transition* call is passed as an argument to the *post-transition* call. The *post-transition* call then returns the final activity information that can be used by the simulator.

### 4.4 Information processing

The actual processing of the activity information is important, as otherwise the activity would be of no use to the simulation [2]. As was previously mentioned, our main purpose for this information is load balancing.

Even though some general relocation strategies exist, a *domain-specific* algorithm should give far better results. General algorithms often have a good idea of *which* nodes are overloaded, though they don't know *how* to solve this. Some strategies do exist, such as migrating models on the boundary[3] of the node until an even distribution is achieved. There are however cases where these strategies perform badly. Our border migration algorithm for example will behave badly if the *'optimal'* allocation significantly extends the border (thus having more models with a neighbor on a different node, increasing the amount of inter-node messages). It is intuitively clear that a huge border between two nodes is unlikely to be efficient due to the increased amount of inter-node communication. Our migration strategy *only* considers computational load. Certainly, extensions to this strategy could be made in order to also take the size of the boundary (and hence, inter-node communication) into account.

[2]The activity information could still be used by the modeller as a basis for manual initial allocation. This is not dynamic and most of the time the modeller will already have insight in how the initial load is distributed.

[3]An atomic DEVS model is defined to be on the boundary, if at least one of its neighbors (either input or output) is not hosted at the same node.

Domain-specific algorithms provide further insight into which models are easy to migrate and which will not cause problems later on. Also, the modeller might know that some problem is better off not being solved due to for example additional rollbacks that would result.

### 4.5 Flowchart

Activity related operations, with the notable exception of load monitoring, only happen infrequently. Time warp simulation already implies such an infrequent process: the *fossil collection* phase, in which all unused states and messages will be removed. Performing relocations right after this fossil collection is ideal, as the amount of data to be transferred is minimal at this point in time. This fossil collection phase is extended, as shown in the flowchart in Figure 5. All activity-related components are user-configurable, offering a maximum of flexibility. At the end, the relocations are passed to a pre-defined migration interface, which will effectively migrate the selected models.

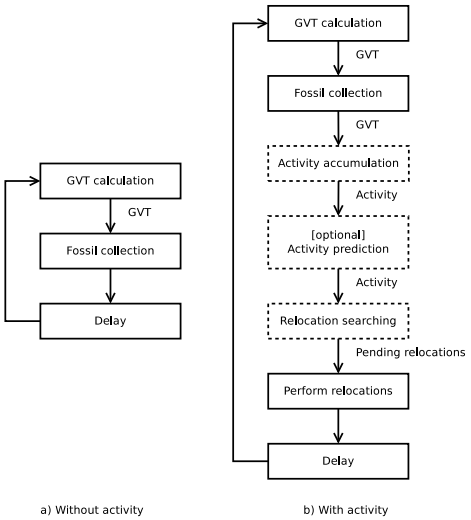


**Figure 5.** Flowchart of the extension for activity-aware migrations

### 4.6 Domain-specific activity

It is also possible to combine the *domain-specific* activity information *and* the *domain-specific* activity relocator. By doing so, the modeller can perform migrations that are completely different from the more general methods.

While this is not directly related to *activity* as it no longer directly represents the activity concept in its more general form, the modeller will generally use these features in an attempt to increase performance. Increasing performance with *hints* from the model is the reason why we use activity in the first place. Domain-specific hints and migration components can thus be seen as an alternative to our default use of activity.

### 4.7 Performance

We are currently still evaluating the impact of our activity-based migration strategies. Some synthetic examples, such as the ring model mentioned in Section 4.1 indicate that such migrations have a huge potential in models where the computational activity is not evenly distributed.

In our performance comparison, we used the previously mentioned ring model. The default activity tracking method was used, which uses the time required for the transition function. The migration component uses the activity information to exchange models on the boundary of the current node. Nodes detect that their total activity is higher than the average and try to

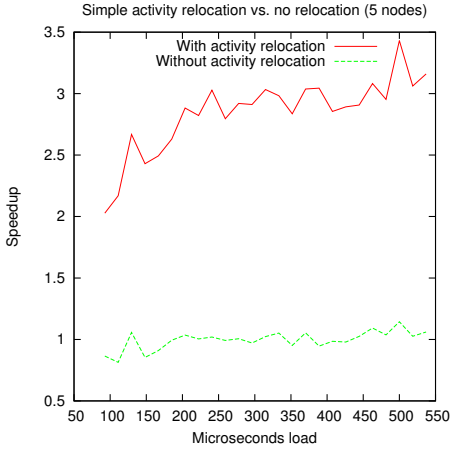


**Figure 6.** Performance comparison of using activity information in a distributed simulation

push away the models on their boundary. These steps are repeated until every node approximately reaches the average activity. Figure 6 shows the difference in simulation time between using activity-based migration and no migration at all. Obviously, the speedup that is achievable with distribution depends on the actual computational load in these *high activity* models. Activity-based migration can be seen to have a considerable impact on simulation performance, in particular when an unacceptable initial allocation is used.

## 5 Related work

The filtering of inactive models was also done in simulators such as vle[6] and adevs[7]. They do not allow the user to define a custom (possibly domain-specific) scheduler and consequently make it impossible to support custom activity definitions or domain-specific knowledge.

DEVSimPy[8], which uses PythonDEVS as its simulator, offers an activity tracking plug-in. The results of this tracking are currently only used to inform the modeller and not for any kind of direct performance improvement.

The PythonPDEVS simulator is based on the original PythonDEVS[9] prototype, but was extended to support, among others: a modular scheduler, distributed simulation, custom activity, and activity tracking/prediction. All of these are concerned with performance optimization and with offering the modeller the possibility to add domain-specific information to the model.

## 6 Conclusions and Future Work

In this paper, we elaborated on the use of activity in the PythonPDEVS simulator. A comparison between several notions of activity was introduced together with a motivation for our choices.

The interpretation and implementation of activity in sequential simulations was subsequently presented. We noted that all activity information could simply be processed by the scheduler without the need for a specific activity component, thanks to our modular and user-configurable scheduler. In addition to simple activity-based scheduling, offering a modular scheduler also made further domain-specific optimizations to the scheduling possible. A performance comparison between several different schedulers was presented. This clearly showed the potential of activity-based scheduling.

Distributed simulation was described next, where activity could also be used as a performance metric to aid in load-balancing between the different simulation nodes. Opportunities for both activity tracking and domain-specific activity information were presented, together with a user-defined migration module that could use the information previously gathered. Again, simulation results were shown in a synthetic model.

PythonPDEVS with our activity additions, presented examples, and extensive documentation can be found at http://msdl.cs.mcgill.ca/people/yentl/50_master.

Our future work focuses on using this approach in less artificial situations, such as fire spread and city traffic, and evaluating performance improvements. This will require further optimizations to the PythonDEVS migration algorithms, but will also require us to insert more domain-specific information into both the activity tracking and migration components.

Other uses of activity will also be considered in the near future, for example to support activity-based initial allocations of the model. Some more (general) applications of activity for the scheduler are also being considered.

## References


[1] A.C.H. Chow, B.P. Zeigler, *Parallel DEVS: a parallel, hierarchical, modular, modeling formalism*, in *Proceedings of the 26th conference on Winter simulation* (SCS, San Diego, CA, USA, 1994), WSC '94, pp. 716–722, ISBN 0-7803-2109-X, http://dl.acm.org/citation.cfm?id=193201.194336

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

[3] D.R. Jefferson, ACM Transactions on Programming Languages and Systems **7**, 404 (1985)

[4] A. Muzy, F. Varenne, B.P. Zeigler, J. Caux, P. Coquillard, L. Touraille, D. Prunetti, P. Caillou, O. Michel, D.R.C. Hill, Simulation **89**, 156 (2013)

[5] A. Muzy, L. Touraille, H. Vangheluwe, O. Michel, D.R. Hill, M.K. Traoré, *Activity Regions in Discrete-Event Systems*, in *SpringSim/TMS-DEVS* (SCS, 2010), pp. 176 – 182

[6] G. Quesnel, R. Duboz, E. Ramat, M.K. Traoré, *VLE: a multimodeling and simulation environment*, in *Proceedings of the 2007 summer computer simulation conference* (Society for Computer Simulation International, San Diego, CA, USA, 2007), SCSC, pp. 367–374, ISBN 1-56555-316-0, http://dl.acm.org/citation.cfm?id=1357910.1357968

[7] J.J. Nutaro, *Adevs*, http://www.ornl.gov/~1qn/adevs/ (2013)

[8] L. Capocchi, J.F. Santucci, B. Poggi, C. Nicolai, *DEVSimPy: A Collaborative Python Software for Modeling and Simulation of DEVS Systems*, in *Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2011 20th IEEE International Workshops on* (2011), pp. 170–175, ISSN 1524-4547

[9] J.S. Bolduc, H. Vangheluwe, Tech. rep., McGill Univ. (2001)