# An evaluation of DEVS simulation tools

**Yentl Van Tendeloo[1]* and Hans Vangheluwe[1,2,3]***

## Abstract
DEVS is a popular formalism for modeling complex dynamic systems using a discrete-event abstraction. Owing to its popularity, and the simplicity of the simulation kernel, a number of tools have been constructed by academia and industry. However, each of these tools has distinct design goals and a specific programming language implementation. Consequently, each supports a specific set of formalisms, combined with a specific set of features. Performance differs significantly between different tools. We provide an overview of the current state of eight different DEVS simulation tools: ADEVS, CD++, DEVS-Suite, MS4 Me, PowerDEVS, PythonPDEVS, VLE, and X-S-Y. We compare supported formalisms, compliance, features, and performance. This paper aims to help modelers in deciding which tools to use to solve their specific problems. It further aims to help tool builders, by showing the aspects of their tools that could be extended in future tool versions.

## Keywords
DEVS, tools, functionality, performance

## 1 Introduction

DEVS is a popular formalism for modeling complex dynamic systems using a discrete-event abstraction. In fact, it can serve as a simulation ''assembly language'' to which models in other formalisms can be mapped.[1] A number of tools have been constructed by academia and industry that allow the modeling and simulation of DEVS models. Each of these tools was developed for a specific application domain, resulting in different design goals.

Since there is no common DEVS format, used by all tools, modelers are tied to their tool. Switching between tools is far from trivial, as each tool uses its own API and language for model specification. Porting models between different tools equates to rewriting the model from scratch. This makes the initial choice of tool important, as switching comes at an arbitrary high cost for huge models.

The problem is further aggravated in that each tool supports a different set of formalisms and features. Even worse, performance between different tools varies by orders of magnitude, depending on the domain and model. It is therefore necessary to provide an overview of the current state of different tools. Modelers will find an overview that can help them to decide which tool to use. Tool builders will have a better idea of how each tool is positioned relative to similar tools.

Previous efforts at comparing different tools are mostly sketched briefly in ''related work'' sections of new contributions.[2,3] These comparisons only include the most closely related tools, and only use criteria relevant to their contribution. Other researchers conduct an in-depth analysis, but only for a limited set of simulation tools, and for a single dimension. For example, there exist in-depth performance comparisons between two simulation tools,[4,5] or a DEVS compliance check of two tools.[6] Other survey papers compare several tools, but do not go in-depth, nor do they use an exhaustive set of criteria that has been externally validated.[7] End-users therefore have no complete comparison between different tools, and information is scattered.

In this paper, we compare a significant number of modern simulation tools (eight in total, selected according to several criteria), using an exhaustive set of functionality criteria, a set of DEVS compliance criteria, and a detailed performance analysis. To guarantee objective criteria selection, we use existing, independently defined criteria for functionality,[8] DEVS compliance,[6] and performance.[9] Only well-argued deviations from these criteria are made.

[1]Department of Mathematics and Computer Science, University of Antwerp, Belgium
[2]Flanders Make, Belgium
[3]McGill University, Montréal, Canada
*SCS member

**Corresponding author:**
Yentl Van Tendeloo, Department of Mathematics and Computer Science, University of Antwerp, Middelheimlaan 1, 2020 Antwerpen, Belgium.
Email: Yentl.VanTendeloo@uantwerpen.be

We will continue by briefly introducing the four main DEVS formalisms implemented by the tools under study (Section 2), as well as the simulation tools themselves (Section 3). The presented tools are compared in terms of functionality (Section 4) and performance (Section 5). Section 6 concludes the paper.

## 2 Background

This section briefly introduces the four DEVS formalisms used in the remainder of this paper. This list of formalisms is by no means complete.

### 2.1 Classic DEVS

Classic DEVS[10] is the earliest DEVS formalism, and forms the foundation for all subsequent variants. Two significant shortcomings were encountered in the literature: the lack of parallelism[11] and the static structure of systems.[12] In response to these shortcomings, different DEVS variants were created.

### 2.2 Parallel DEVS

Parallel DEVS[11] was introduced to solve the performance problem caused by the (artificial) *select* function of Classic DEVS. Parallel DEVS is widely supported today, and has replaced Classic DEVS in most tools. Parallel DEVS models can be more easily parallelized than Classic DEVS models, owing to the lack of the *select* function. The abstract simulator[13] shows parallel potential, by triggering all transition functions simultaneously. While it is argued that this might not be the best option in some cases,[14] it clearly shows the possibility for parallelism.

### 2.3 Dynamic Structure DEVS

Both Classic DEVS and Parallel DEVS lack support for dynamically changing models. While it is possible to emulate dynamically changing models (e.g., by putting the model structure in the state too), this introduces a significant amount of accidental complexity. Several Dynamic Structure DEVS variants were introduced to make structural changes more intuitive, removing the need for manual "tricks." Examples are DSDEVS[15,16] and DynDEVS.[17] These formalisms provide a mapping from an extended DEVS formalism, supporting dynamic structure, to the basic DEVS structures, proving their equivalence.[12] Performing such a mapping during simulation is inefficient, so implementations frequently use shortcuts.

### 2.4 Cell DEVS

Cell DEVS[18] is a combination of DEVS and Cellular Automata. Cellular Automata use a discrete time base, making them increasingly resource intensive with increasing time granularity. Conversely, DEVS uses a discrete-event time base, only taking into account points in time where events are processed and exchanged. It is not optimized, however, for synchronous communication, as is the case in Cellular Automata. Cell DEVS merges these two formalisms to combine a discrete-event time base with synchronous communication. A Parallel Cell DEVS[19] variant was also introduced with the advent of Parallel DEVS.

Quantization[10] can be used to prevent the propagation of insignificant state changes, thus improving performance. As the definition of "insignificant" varies from model to model, the user must define the significance threshold manually. Using quantization is a trade-off between accuracy and performance.

## 3 Simulation tools

This section provides a brief introduction to the DEVS simulation tools under study. We have selected the tools based on the following criteria.

1. *Features.* Owing to the wide variety in simulation software, all tools support a different set of features. In our evaluation, we consider tools with an extensive feature set, but also tools with a very small number of special features.
2. *Performance.* For large-scale simulations, performance is critical. While we certainly want to include highly optimized simulation tools, we also want to include tools with a potentially bad performance, in case there is some gain in other dimensions (e.g., more features).
3. *Popularity.* Finally, popularity is an important criterion, owing to inertia: a popular tool will still be used as long as there are not enough convincing arguments to move to an all-round better tool. There is probably a well-argued reason for the tool's popularity.

A summary of our selected tools is given in Table 1. For each tool, we provide a brief introduction, an example or screenshot where appropriate, and our rationale for inclusion of this tool.

### 3.1 ADEVS

ADEVS[20] is a lightweight C++ library, offering DEVS simulation. Both atomic and coupled models are written in C++ code, which must include the ADEVS headers. Owing to the extensive use of templates, the headers contain all required source code. The simulation kernel and model are compiled into a single executable, and must therefore be recompiled after every model edit.

**Table 1.** Simulation tools under study.

|  | Version | Release | Language | License |
|---|---|---|---|---|
| *ADEVS* | 2.8.1 | 2014 | C++ | FreeBSD |
| *CD++* | 2.0-R.45 | 1999 | C++ & custom | (unspecified) |
| *DEVS-Suite* | 2.1.0 | 2009 | Java | LGPLv2 |
| *MS4 Me* | 1.5.0 | 2015 | Java & custom | Proprietary |
| *PowerDEVS* | 2.4rev | 2015 | C++ | GPLv3 |
| *PythonPDEVS* | 2.3 | 2015 | Python | Apache-2.0 |
| *VLE* | 1.2 | 2014 | C++ & XML | GPLv3 |
| *X-S-Y* | 1.0.0 | 2012 | Python | LGPL |

*3.1.1 Example.* As models are pure C++ code, there is nothing to show as an example. Examples of how the API works can be found in the documentation.

*3.1.2 Rationale.* Because it is lightweight and has significant potential for static optimization, ADEVS is included for its potential performance. The use of efficient algorithms[21] and results from previous performance evaluations[4,7,22] indicate that it will probably be (one of) the most efficient simulation tools under study. It also offers some interesting features, such as simulation of hybrid systems and OpenModelica[23] bindings.

## 3.2 CD++

CD++[24] is a DEVS simulator written in C++. Simulation of Cell DEVS models is its main feature, though normal DEVS models can be simulated too. DEVS models can also be coupled to Cell DEVS models. Atomic models are written in C++ and are linked to the simulation tool. Coupled models are written in a custom syntax, which is interpreted at simulation time. Changes to atomic models require recompilation and linking to the simulation tool. Changes to coupled models do not require any recompilation at all, as these are interpreted during simulation. The complete behavior of Cell DEVS models is defined using the custom syntax, which is completely interpreted. A graphical modeling environment, called CD++Builder,[2] can be used to create the models.

Note that every exchanged event must be a floating point number. Sending complex events, such as records, requires a workaround by mapping all attributes to a separate port.

*3.2.1 Example.* An example of the custom syntax, together with a workaround for complex events, is shown in Listing 1. Normally, a car is represented as a single object, and passed as such. In CD++, every attribute has its own port.

```
[top]
components : gen@Generator col@Collector proc@RoadSegment
Link : car_out_id@gen car_in_id@proc
Link : car_out_departure_time@gen car_in_departure_time@proc
Link : car_out_v_pref@gen car_in_v_pref@proc
Link : car_out_v@gen car_in_v@proc
Link : car_out_dv_pos_max@gen car_in_dv_pos_max@proc
Link : car_out_dv_neg_max@gen car_in_dv_neg_max@proc
Link : car_out_d_travelled@gen car_in_d_travelled@proc
Link : car_out_flush@gen car_in_flush@proc
Link : Q_send@gen Q_recv@proc
Link : Q_sack_id@proc Q_rack_id@gen
Link : Q_sack_t_until_dep@proc Q_rack_t_until_dep@gen
Link : Q_sack_flush@proc Q_rack_flush@gen
Link : car_out_id@proc car_in_id@col
Link : car_out_departure_time@proc car_in_departure_time@col
Link : car_out_v_pref@proc car_in_v_pref@col
Link : car_out_v@proc car_in_v@col
Link : car_out_dv_pos_max@proc car_in_dv_pos_max@col
Link : car_out_dv_neg_max@proc car_in_dv_neg_max@col
Link : car_out_d_travelled@proc car_in_d_travelled@col
Link : car_out_flush@proc car_in_flush@col

[gen]
IAT_min : 2.0
IAT_max : 2.0
v_pref_min : 20.0
v_pref_max : 20.0
dv_pos_max : 3.0
dv_neg_max : 5.0

[proc]
l : 10.0
v_max : 18.0
observ_delay : 0.1
```

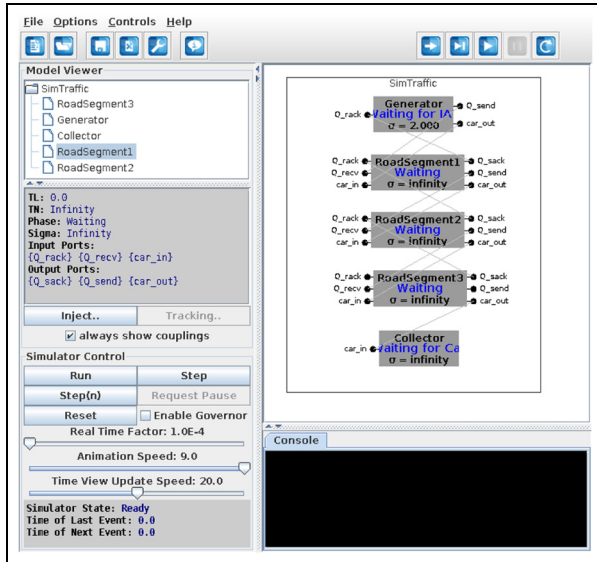**Listing 1.** Example CD++ coupled model.

**Figure 1.** DEVS-Suite SimView example.

*3.2.2 Rationale.* CD++ is a mature tool and is widely used in the literature for its Cell DEVS functionality.[25–27] It is included in our evaluation because of its popularity.

*3.2.3 Remarks.* Several versions of CD++ exist, such as N-CD++,[24] PCD++,[19] and Dynamic Structure CD++.[28] We decided to use N-CD++ in our comparison. Glinsky and Wainer previously compared the performance between N-CD++ and PCD++, showing that N-CD++ had a lower overhead,[9] and should thus be faster. N-CD++ is also the version compared with ADEVS by Wainer et al.,[5] leading us to believe that this is the most mature version of their tool.

## 3.3 DEVS-Suite

DEVS-Suite[29] is the successor of DEVSJava.[30] Both are implemented in Java. Its features include visualization of coupled model simulation, event injection during simulation, and simulation tracking.

Both atomic and coupled models are written in Java and are loaded into the simulation tool through introspection. Changes require recompilation of the model, but do not require any action on the simulation tool.

*3.3.1 Example.* Figure 1 shows the SimView visualization of a simple model. At the left, an overview of the model is given, showing the components of the coupled model. Each atomic model can be clicked on, revealing further information about the selected model. Simulation control buttons can be seen below the model overview, combined with sliders to determine the speed of the visualization. At the right, the model is visualized, summarizing the most important information for every model, as well as its couplings. At the bottom of the visualization, a console is used to print errors.
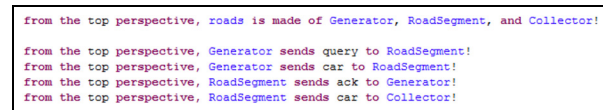


**Figure 2.** MS4 Me DNL example.



**Figure 3.** MS4 Me SES example.

*3.3.2 Rationale.* Both DEVS-Suite and DEVSJava are frequently used in the literature.[31–33] It is one of the few simulation tools that includes visualization and debugging functionality. Therefore, it is added both for its popularity and features.

## 3.4 MS4 Me

MS4 Modeling Environment (MS4 Me)[34] is a DEVS modeling environment and simulator. It is written in Java and based on the Eclipse framework.

Atomic models are created using a custom, natural language-like language called DNL, combined with fragments of Java code. Files are automatically translated to Java code, and subsequently compiled. Coupled models can be constructed using System Entity Structure (SES),[35–37] which are pruned before simulation commences.

*3.4.1 Example.* Figure 2 presents DNL notation, used for the construction of atomic models. It shows the inclusion of Java code and the use of Java types. Figure 3 presents SES notation, used for the construction of coupled models.

Figure 4 presents the SimViewer, used for visualization of model simulation. Visualization is fairly similar to DEVS-Suite: each model can be inspected separately, combined with a graphical representation of the complete model. The complete content of exchanged events is also visualized. It is possible to hide the couplings or the ports, or both. Simulation status, such as the current time and the number of transitions, is shown at the right. At the bottom, a console is present, which logs every transition.
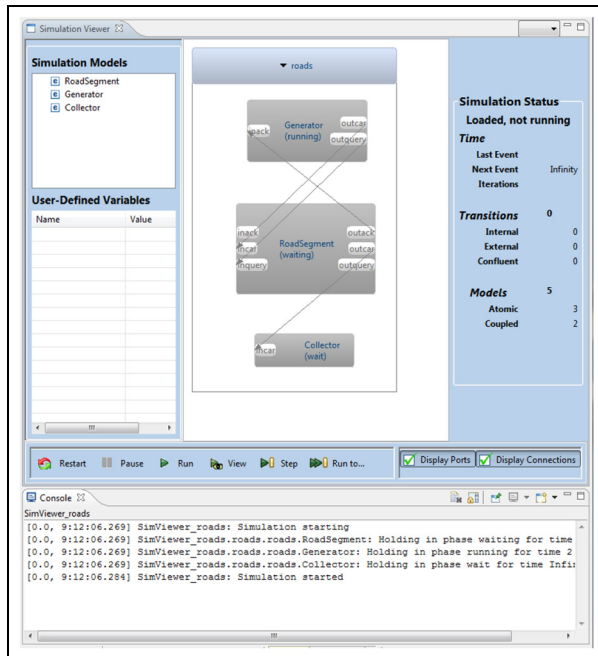
**Figure 4.** MS4 Me simulation example.



**Figure 5.** PowerDEVS graphical modeling environment.

*3.4.2 Rationale.* MS4 Me is included as it tries to move away from the use of programming languages. Whereas other tools redirect the modeler to a general-purpose programming language for the atomic models, MS4 Me defines its own syntax to aid nonprogrammers. Furthermore, it is a proprietary tool, which will also give us insight into the state-of-the-art of commercial DEVS simulation tools.

## 3.5 PowerDEVS

PowerDEVS[38,39] is a Classic DEVS modeling and simulation environment implemented in C++. It consists of a graphical modeling environment, an atomic model editor, and a code generator. The code generator generates C++ code, which can optionally also be handwritten. PowerDEVS offers an intuitive modeling environment (with user-definable icons for models), combined with a library of models that can be reused, or used as examples.

*3.5.1 Example.* Figure 5 presents the IDE, with a focus on the graphical coupling of models. The information shown is quite different from other tools, as this environment presents a *design* view, instead of a *simulation* view. In a design view, models can be constructed by adding hierarchy, adding atomic models, coupling models, or configuring simulation parameters. No information about the state of the models or the simulation is shown, as all information is independent of simulation execution. Conversely, a simulation view visualizes the state of models and previously defined hierarchy and couplings. Modifications are
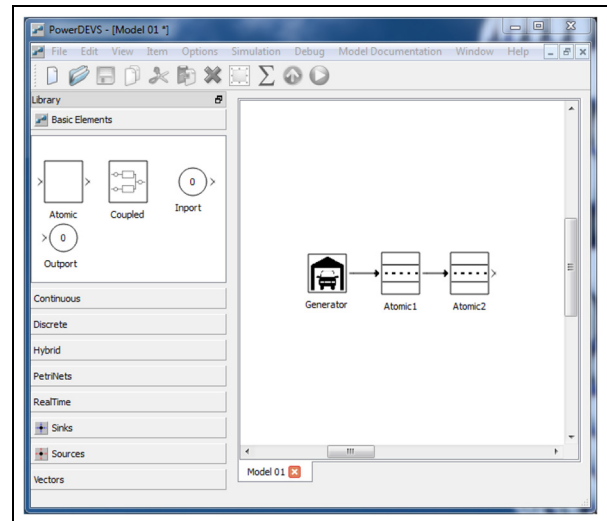
not allowed, as the model is already loaded and being simulated. Model simulation takes place without any kind of visualization.

*3.5.2 Rationale.* PowerDEVS is very similar to ADEVS, as it also focuses on the simulation of hybrid systems, and uses C++ as its implementation language. Unlike ADEVS, PowerDEVS offers a modeling environment that aids the modeler in creating valid DEVS models with a lesser degree of coding required. PowerDEVS is aimed more at nonprogrammers, while still offering the potential for high performance. It is thus included for both its potential performance and its various features.

## 3.6 PythonPDEVS

PythonPDEVS[22] is a DEVS simulator written in Python. Owing to its implementation in Python, an interpreted, dynamically typed language, fast prototyping of models becomes possible. Despite its interpretation-based nature, PythonPDEVS attempts to achieve high performance. Both atomic and coupled models are written in Python, making (re)compilation unnecessary.

PythonPDEVS is used as the simulation kernel in several other tools. For example, DEVSimPy[40] offers a graphical modeling environment for coupled models, combined with an experimentation environment. A debugging front-end[41] offers a graphical modeling environment for atomic and coupled models alike, including advanced debugging capabilities.

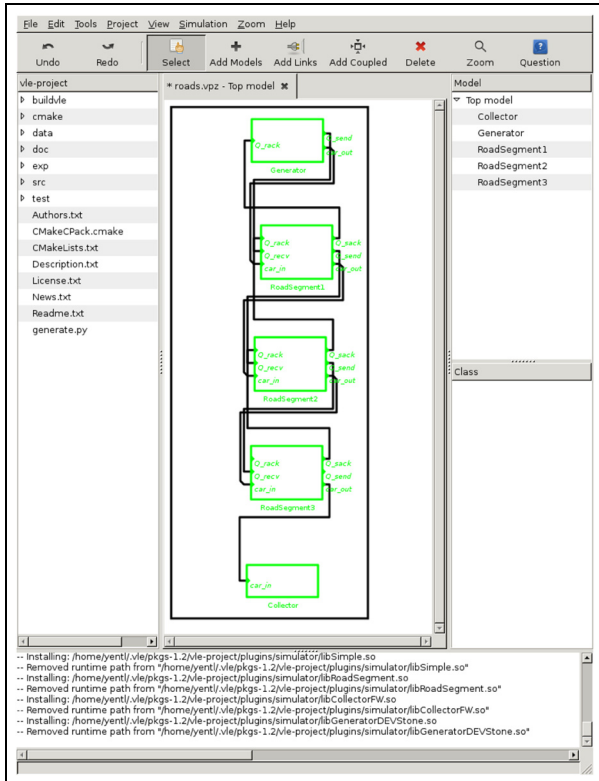*3.6.1 Example.* As models are pure Python code, there is nothing particular to show as an example. For examples of

**Figure 6.** GVLE visualization of a coupled model.

the debugging extension[41] or DEVSimPy,[40] we refer to the documentation of the respective tools.

*3.6.2 Rationale.* PythonPDEVS has a focus on performance, despite its implementation in an interpretation-based language.[22,42] We therefore included PythonPDEVS for its potential performance, but also for its set of supported features.

### 3.7 VLE

The Virtual Laboratory Environment (VLE)[43] is a multi-modeling and simulation platform written in C++. It includes an IDE for model development and experimentation. Models are combined in ''projects,'' which are managed by an automatically created CMake script.

Atomic models are written in C++ and thus require recompilation of the models after changes. The simulation kernel and IDE do not need to be recompiled. Coupled models are created using either the graphical environment (called GVLE), or by manually writing the XML files.

VLE is the simulation kernel, with several bindings and ''apps'' to add functionality, such as an IDE (GVLE), distributed simulation using MPI (MVLE), Python bindings (PyVLE), and R bindings (RVLE).

*3.7.1 Example.* Like PowerDEVS, VLE only offers a modeling environment without a simulation view. This environment is provided by GVLE, shown in Figure 6.

*3.7.2 Rationale.* VLE comes close to PowerDEVS in terms of supported features, as both offer a full modeling interface that generates a model for simulation by a (separate) simulator. Its use of C++ might also indicate that performance is one of the concerns to the developers.

### 3.8 X-S-Y

X-S-Y[44] is a DEVS simulator written in Python. Its distinguishing feature is the verification of FD-DEVS (Finite and Deterministic DEVS) models. A small command line interface is provided, allowing for simulation control.

*3.8.1 Example.* As models are pure Python code, there is nothing particular to show as an example.

*3.8.2 Rationale.* X-S-Y offers the unique feature of supporting verification of FD-DEVS models. As no other tools implement verification, it is interesting to see how it compares with them.

*3.8.3 Remarks.* During our analysis, we found that X-S-Y uses only about 10–15% of the available CPU time. Inspection of the source code revealed that a sleep of 1 ms occurs after every simulation step. This sleep was removed for our performance benchmarks, as it offers a fairer comparison between the used simulation algorithms.

## 4 Functionality

In this section, we will compare the functionality of the previously mentioned tools. Tools are first evaluated based on generic simulation tool criteria.[8] Afterward, a brief comparison is made based on some DEVS-specific criteria: which formalisms do they support, and how compliant they are to them.

### 4.1 General evaluation criteria

The simulation tool evaluation criteria from Nikoukaran et al.[8] are used as a basis. Some criteria were dropped, as they were irrelevant for a DEVS-specific comparison. Others were added, in case they were deemed necessary to achieve a better distinction between the different tools.

Table 2 shows an overview of our evaluation results. A feature is either present (marked as a green ''Y''), not present (marked as a red ''N''), or only supported partially, with manual coding, or through the use of extensions (marked as a yellow ''M''). Normally, such a comparison

**Table 2.** General evaluation, based on Nikoukaran et al.[8] and Tewoldeberhan et al.[45]

| | | ADEVS | CD++ | DEVS-Suite | MS4 Me | PowerDEVS | PythonPDEVS | VLE | X-S-Y |
|---|---|---|---|---|---|---|---|---|---|
| Vendor | Pedigree | N | N | N | Y | N | N | N | N |
| | Documentation | Y | Y | N | Y | M | Y | Y | Y |
| | Support | N | N | N | Y | N | N | N | N |
| Model and input | Library | N | Y | N | Y | Y | N | N | N |
| | Coding | Y | M | Y | M | M | Y | M | Y |
| | Input | M | Y | Y | Y | M | M | N | M |
| Execution | Speed control | N | M | Y | Y | Y | Y | N | Y |
| | Multiple runs | Y | Y | N | N | Y | Y | Y | Y |
| | Batch runs | Y | Y | N | N | Y | Y | Y | N |
| | Parallel | Y | Y | N | N | N | Y | Y | N |
| | Distributed | N | Y | N | N | N | Y | Y | N |
| | Executable models | Y | N | N | N | Y | N | N | N |
| | Termination condition | Y | N | N | N | N | Y | N | N |
| Animation | Time next | N | N | Y | Y | N | M | N | N |
| | State | N | Y | Y | Y | N | M | N | N |
| | Messages | N | N | Y | Y | N | M | N | N |
| | Transitioning | N | N | N | Y | N | M | N | N |
| | Sequence | N | N | N | Y | N | N | N | N |
| Testing and efficiency | Tracing | Y | Y | Y | Y | Y | Y | Y | Y |
| | Step function | Y | N | Y | Y | Y | M | N | Y |
| | Verification | N | N | N | N | N | N | N | Y |
| | Backward clock | N | N | N | N | N | M | N | N |
| | Interaction | Y | Y | Y | Y | M | Y | N | Y |
| | Multitasking | Y | Y | Y | Y | Y | Y | Y | Y |
| | Breakpoints | N | N | N | N | N | M | N | N |
| Output | Delivery | Y | Y | Y | Y | Y | Y | Y | Y |
| | Graphics | N | Y | Y | Y | M | N | N | N |
| User | Orientation | N | M | N | M | M | N | N | N |
| | Financial | Y | Y | Y | N | Y | Y | Y | Y |

is made using scores and weightings, as in Tewoldeberhan et al.[45] But as scoring is highly dependent on the needs of the end-user, we omit this part.

*4.1.1 Vendor.* Concerning the vendor **pedigree**, MS4 Me is the clear winner. RTSync is a spin-off of ACIMS, the creators of DEVS-Suite. ACIMS is headed by Dr. Bernard Zeigler, the founder of DEVS. RTSync, however, is still relatively young, and MS4 Me is still in beta testing. Other tools are developed by different research groups, specialized in DEVS simulation. Despite them being stable and usable, they are to be considered prototypes.

**Documentation** and sample models are provided by all tools. DEVS-Suite has the least documentation of all, as there is no user's guide beyond installation and running of the tool. PowerDEVS has limited documentation too: the

*user's guide* link in the help menu does not respond, and the online manual is only partially completed. However, the tool and its interface are fairly straightforward, even without documentation. Note that some parts of the CD++ documentation require registration at their website.

For **support**, MS4 Me is again the clear winner, as it is the only one offering tuition and consulting services. On a commercial level, this is a crucial advantage.

*4.1.2 Model and input.* CD++ and MS4 Me are the only tools providing an online **library of models**. The size and reusability of these models is a different matter. CD++ offers a sizable repository of models, containing a mix of reusable and stand-alone models. MS4 Me offers a relatively small model store, mostly containing example models instead of reusable models. PowerDEVS comes

bundled with a small (offline) library too, though most models are oriented toward hybrid systems.

All tools involve **coding** to a certain degree. CD++, PowerDEVS, and VLE require coding at the level of atomic models, while offering a user-friendly format for the construction of coupled models. MS4 Me goes one step further, and presents a custom language for atomic models too. It is possible to write Java code manually, which will be included in the generated code. This allows the combination of the best of both worlds: basic conditionals and states using DNL, but data structure manipulations using Java.

Most tools allow **user input** during simulation, though differences exist in the kind of input. ADEVS requires the user to implement this functionality in the model manually and experiment. CD++ reads input from a file, which is injected into the simulation at the desired time. DEVS-Suite and MS4 Me support graphical injection of events during simulation. PowerDEVS can take user input through the use of an input port. PythonPDEVS does not support user input, except during realtime simulation. (In realtime simulation, simulation time is synchronized with the wall-clock time, creating a linear relation between them.[46]) The PythonPDEVS debugging front-end[41] adds this feature for all kinds of simulation. VLE does not support input at all. X-S-Y reads its input from standard input, so the user either has to manually input the events, or write a wrapper script.

*4.1.3 Execution.* Some tools include **speed control**, which is the option to run simulation using either scaled wall-clock time (realtime simulation), or analytical time (as-fast-as-possible simulation). ADEVS and VLE are the only tools that do not support speed control by default. For CD++, the extended RT-CD++[47] is required for realtime simulation.

**Multiple runs**, each with different parameters, are widely supported, but might require some coding from the user (i.e., writing the loop and the parameter variation). DEVS-Suite and MS4 Me do not support this at all, as these are GUI-only tools, and their GUI does not offer this function.

**Batch runs** are similar, except for X-S-Y, which waits for user input before terminating the simulation.

**Parallel simulation** is supported by ADEVS, CD++, PythonPDEVS, and VLE. ADEVS allows this through the use of conservative synchronization, where each core is at a different point in simulated time. CD++, PythonPDEVS, and VLE offer distributed simulation, which can also be used for parallel simulation by hosting multiple nodes at the same machine. This generally imposes additional overhead, owing to the use of more general algorithms.

Many differences exist in terms of **distributed simulation**. CD++, and in particular PCD++, supports distributed simulation using different synchronization methods, provided by the Warped library.[48] PythonPDEVS uses Time Warp optimistic simulation, with MPI as the middleware. VLE does not support *model* distribution, but does support *experiment* distribution. A single model is not split up and distributed over multiple nodes, but the same model can be executed at different nodes simultaneously, using varying parameters. While this is fine when simulating several scenarios, a single model cannot be distributed or parallelized.

ADEVS and PowerDEVS support **executable models**, as they compile both the simulation tool and model into a single executable. All other simulators have models as files that are loaded by the simulator.

**Termination conditions** are included as an additional criteria. They determine how the modeler can specify when simulation terminates. Three different options were found: (1) step count, as found in DEVS-Suite and MS4 Me; (2) simulation time, as found in the others; and (3) termination condition, as found in ADEVS and PythonPDEVS.

With step counting, simulation halts after a predetermined number of simulation steps. Steps are unrelated to the simulation time, making it difficult for the user to predict exactly when simulation will terminate. The use of simulation time is more common, where simulation will halt at a predetermined point in the simulation. This has the advantage that it is closer to the problem domain. Should the model change (e.g., different parameters), simulation will still run up to the same point in simulated time. A termination condition is even closer to the problem domain, and allows the user to specify a termination *state*. As soon as this state is reached, simulation will terminate. This can be useful in design space exploration, where specific states in simulation are known to be unacceptable. Thus, it becomes possible to quickly prune these branches, even before the usual termination time is reached.

*4.1.4 Animation.* For animation, we deviate from the criteria, since these criteria do not distinguish between the tools in our study. We specify which part of the simulation algorithm is visualized. Message sequence visualization, to allow for manual dependency analysis, is another of our criteria. PythonPDEVS does not support any of these features, though its debugging extension does. As it is an extension, these entries are marked with an ''M,'' even if they are fully present. The features that we selected for visualization correspond to the different stages seen in the abstract simulator.

The first step of DEVS simulation is to find out which models are triggered at what **time**. DEVS-Suite and MS4 Me offer this with their simulation viewer: each model is annotated with the time remaining until internal transition, or the absolute time of its next transition.

The second step is **state** visualization. Again, DEVS-Suite and MS4 Me support this by showing the name of the current state. CD++ can output a grid of floating point numbers, indicating the state of the cells in Cell DEVS

simulation. After simulation, these values can be visualized with the "drawlog" tool that is provided with CD++.

The third step is **message** visualization. DEVS-Suite allows visualization of the type of message exchanged (e.g., "Car"), but does not show the content. MS4 Me additionally visualizes event attributes (e.g., current velocity, maximal speed, or color).

The final step is **transitioning** model visualization, optionally distinguishing between internal, external, and confluent transitions. DEVS-Suite does not support this, though it can be inferred by manually following the exchanged messages. MS4 Me visualizes models that perform their transition by highlighting them, but makes no distinction between transition types. In PythonPDEVS, models executing their internal, external, or confluent transition are highlighted in distinct colors.

Additionally, MS4 Me allows simulation visualization with a **sequence diagram**. This sequence diagram visualizes exchanged messages, enabling simple causality analysis.

*4.1.5 Testing and efficiency.* All tools support **tracing**, though there is a distinction between textual or graphical, or both. Most textual output can be parsed, to allow for visualization with a different tool.

Simulation **stepping** is possible with several tools. ADEVS supports it, but the user must implement it manually. DEVS-Suite and MS4 Me support stepping through the simulation "or" DEVS-Suite and MS4 Me offer stepping through the simulation, combined with visualization. They also support stepping for a certain number of steps with a specified realtime scale. PowerDEVS only supports the execution of a model for a specific number of steps. For PythonPDEVS, the debugging front-end can be used for stepping, as well as for visualization. X-S-Y allows stepping through its command line interface.

**Verification** is only supported by X-S-Y, and only for FD-DEVS models.

**Backward clock**, or stepping back, is only supported by the PythonPDEVS debugging extension.

All tools, apart from VLE, support simulation **interaction**. Different tools support different degrees of interaction: from high-level simulation control (e.g., pausing and resuming), to low-level state modification (e.g., the modification of model states at simulation time).

**Multitasking** is supported by all tools, as they operate on text files instead of binary files. Several instances of a tool can be executed, allowing for a number of concurrent simulation executions. Models can be altered during simulation, using a text editor.

**Breakpoints** are only supported by the PythonPDEVS debugging extension.

*4.1.6 Output.* All tools **deliver** some output, though it is often textual. Of special interest are CD++, DEVS-Suite,

and MS4 Me, which can create **graphics** of the simulation. PowerDEVS can make graphical output too, though this requires the use of built-in blocks from the library. Other tools require additional software to create graphical output.

For CD++, these graphics are heatmap-like figures, which show the simulation state. For DEVS-Suite and MS4 Me, the graphics are simulation state traces, which show the simulation times at which some attribute of interest has changed.

*4.1.7 User.* Owing to the use of general-purpose programming languages, all tools are developer-**oriented**. An exception is MS4 Me, which uses a natural language-like specification, combined with methods written in Java. This allows domain experts to work using DNL, while developers write utility functions in Java. CD++ is also noteworthy, as Cell DEVS models can be completely specified using a custom language. With CD++Builder, it is also possible to use the DEVS-Graph formalism,[2] which allows graphical construction of atomic DEVS models. A graphical front-end for PythonPDEVS also exists,[49] which uses a neutral language. Although it still relies on writing code, the used language is specific to DEVS and can filter out illegal constructs, such as state modifications in the output function. All other tools require the modeler to write code in a general-purpose programming language.

**Financially**, all tools are open-source and freely available, except for MS4 Me, which is proprietary and requires a paid license. Installation and maintenance costs should also be considered, though these are difficult to estimate. Familiarity of the modelers with the used language is also of importance, as otherwise a significant amount of training is required.

## 4.2 DEVS-specific evaluation criteria

Some additional criteria were added to check for strict conformance to the DEVS formalism, based on a previously defined set of criteria.[6] Several were dropped, whereas some were added to extend the criteria to Parallel DEVS.

First, the supported formalisms of all tools are compared. Only the four previously introduced DEVS formalisms are considered here. Some tools support additional formalisms. An overview is shown in Table 3.

The remainder of this paper will not present examples and performance results of either Cell DEVS or Dynamic Structure DEVS, as these are not widely supported by the tools under study. Similarly, we do not go deeper into some other aspects of DEVS simulation, such as hybrid simulation or HLA-compliance: most tools under study have only limited (if at all) support for both. A comparison detailing any of these aspects, either through detailed features or performance analysis, would require a different set of tools to make a fair comparison.

**Table 3.** DEVS-specific evaluation, loosely based on Li et al.[6]

| | | ADEVS | CD++ | DEVS-Suite | MS4 Me | PowerDEVS | PythonPDEVS | VLE | X-S-Y |
|---|---|---|---|---|---|---|---|---|---|
| Formalisms | Parallel DEVS | Y | M | Y | Y | N | Y | Y | N |
| | Classic DEVS | N | Y | N | N | Y | Y | N | Y |
| | Dynamic structure | Y | M | N | N | N | Y | Y | N |
| | Cell DEVS | N | Y | N | N | N | N | Y | N |
| Compliance | Translation functions | M | N | N | N | N | Y | N | N |
| | Event modularity | N | M | N | N | N | M | N | N |
| | Positive time | Y | N | N | M | N | Y | Y | N |
| | Select function | – | N | – | – | M | Y | – | N |
| | Confluent | Y | – | Y | N | – | Y | Y | – |

*4.2.1 Supported formalisms.* Each simulation tool sets out to support a different set of formalisms. **Parallel DEVS**, the successor of Classic DEVS, is supported in all tools, except for CD++, PowerDEVS, and X-S-Y. For CD++, this is because we used N-CD++ instead of PCD++, which is a Classic DEVS simulator instead of a Parallel DEVS simulator. For X-S-Y and PowerDEVS, no Parallel DEVS version is available at the moment.

**Classic DEVS** is used in those tools that do not support Parallel DEVS. PythonPDEVS supports both Parallel DEVS and Classic DEVS, the former for performance and the latter for support for legacy models.

On top of the previously defined DEVS formalisms, some tools offer **dynamic structure**. Owing to the variety of dynamic structure formalisms, such as DSDEVS[12] and DynDEVS,[17] we have grouped all of these under a common term. A modified version of CD++[28] exists, which supports dynamic structure.

Finally, CD++ and VLE have specific modeling and simulation options for **Cell DEVS** models.

*4.2.2 DEVS compliance.* Our first criterion is the presence of the **translation function**, which is the function denoted by the $Z_{i,j}$ in the formal definition. It translates event from output-to-input, output-to-output, and input-to-input ports. Despite the arguments in favor of this function, only PythonPDEVS implements this function. For ADEVS, it is possible to overload the routing mechanism, introducing message modification there.

DEVS is a **modular** formalism: models can only communicate through event exchange. There is no way to read or change the state of another model, except through the exchange of an event that causes the model to alter its own state. While breaking modularity allows for improved performance,[50] modularity is a necessity in DEVS, as it is the basis of its closure under coupling. Owing to the use of general-purpose programming languages in the models, several methods to break modularity exist. While some

limitations can be imposed (e.g., preventing the passing of pointers or references in events), there is always the possibility of abusing language constructs (e.g., global variables). PythonPDEVS partially enforces modularity by making deep copies of exchanged events. CD++ also enforces modularity in this respect. However, this is mainly caused by restriction of events, which can only be floating point values. Some work has been conducted on using static analysis of models, using a neutral language, to prevent such constructs.[49]

A third requirement is for the **time advance function** to be **nonnegative**. As a negative time advance is clearly impossible in reality, this is also disallowed in a DEVS model (though a time advance of 0 is allowed). Most tools do not check this, assuming that the user follows the formalism, and can therefore give incorrect simulation results. A notable case is MS4 Me: statically detectable negative time advances are flagged as modeling errors, though there is no run-time check for dynamically obtained values.

A fourth requirement is the presence of a *select* **function**, which only applies to Classic DEVS models. The *select* function determines the model to execute if more than one model is scheduled to execute an internal transition function at the same time. Of the four tools supporting Classic DEVS, only PythonPDEVS allows users to define the *select* function explicitly. PowerDEVS allows the definition of a priority list, though it is not possible to define arbitrary functions. CD++ and X-S-Y implicitly use a hard-coded *select* function, such as selecting the first model after alphabetic sorting on model name.

Our final requirement is the presence of a **confluent transition function**, which only applies to Parallel DEVS models. The confluent transition function is triggered if both the internal and external transition fire at exactly the same point in time. All of the Parallel DEVS simulators support this requirement, except for MS4 Me, where a default is assumed. Other tools often provide the same default, though the user can override this default.

# 5 Performance

With the growing demand for computing resources by modern simulation applications, the need for efficient simulators increases. Parallel and distributed simulation thus becomes necessary to allow multiple computers to work cooperatively on a single simulation. Distribution and parallelism do not solve all problems though, as some problems are inherently difficult to parallelize. Efficient sequential algorithms therefore stay relevant, as parallel and distributed synchronization algorithms are a layer on top of sequential algorithms.

As not all of our tools under study support parallel or distributed simulation—and even those that support it use very different synchronization protocols, making a fair comparison difficult—we have opted to study only sequential performance.

All simulations were performed on a system with an Intel i5-4570 (3.2 GHz) with 16 GB of DDR3-1600 main memory, running Gentoo Linux with kernel version 3.18.22. We used the following software versions: GCC 4.9.3, OpenJDK IcedTea6 1.13.9, Python 2.7.10, and PyPy 2.6.0. All tools were compiled with the optimizations defined in their Makefile. For ADEVS, we had to make a Makefile ourselves, where we opted for the compiler flag ''-O2.'' MS4 Me was benchmarked on the same machine, but using Windows 7 Enterprise SP1.

All of the source code used for this paper (i.e., all models, benchmarks, and a copy of the tools used where possible) can be found at http://msdl.cs.mcgill.ca/people/yentl/DEVS/tools.tgz.

Tools written in Python were benchmarked using both CPython (https://www.python.org/, the reference implementation) and PyPy (http://pypy.org/, an alternative implementation using JIT compilation). As most of the inefficiency of Python code is caused by its interpreted nature, we can partially mitigate this with PyPy, which uses just-in-time compilation. While PyPy can be used in cases where performance is important, most users will prefer to use CPython, as it is installed by default on most Linux distributions. We have opted to include both: CPython for the average user, who is unconcerned about performance, but also PyPy, for power users. As PyPy is (almost) a drop-in replacement for CPython, it is possible to develop and prototype using CPython, but perform the actual long-running simulations using PyPy.

We present the benchmarks and their results, and discuss the results.

## 5.1 Benchmarks

Three different benchmark models are used, offering insight into the performance of different aspects of the simulation algorithms. Of these three, two are synthetic, and one is more realistic, though still fairly simple.

For the synthetic benchmarks, we looked into the DEVStone[9] benchmark for inspiration. DEVStone defines four model types: (1) LI: models with low interconnection; (2) HI: models with high interconnection; (3) HO: HI models with high number of outputs; (4) HOmod: models with an exponential number of coupling and outputs. The transition functions of each model contain artificial computation, in the form of *Dhrystones*.

We found some limitations to the default DEVStone models, such as all atomic models triggering their external (and possibly internal) transitions simultaneously, which is unlikely in realistic DEVS models. The execution of Dhrystones during the transition functions does not fit our desired analysis either: some of our tools are written in different programming languages, each with distinct performance characteristics. As the operations in the transition functions are time-bounded, a more complex (i.e., more computation) model is executed in efficient programming languages. For a fair comparison, we would like all tools to simulate an identical model, that is, without any computation in the transition functions. The overhead of the simulation is no longer computable, as there is no longer any ''theoretical simulation time,'' as was the case in the original definition.[9] We can, however, compare total execution times for a specific model in a specific configuration. By minimizing the amount of model computation, execution times maximally show the time taken by the simulation kernel.

The use of deep hierarchy is not considered in our benchmarks, as models that are hundreds of levels deep are unrealistic. Additionally, modern simulation tools often provide automatic flattening, creating a single coupled model with all atomic models as its direct children. This situation closely mimics the models of our proposed benchmarks. Note that, whereas flattening has a positive effect on most realistic models, it is not necessarily always an optimization: depending on model structure, the flattening overhead might be significant compared with the small gain.

There is no doubt that the benchmark implementations, for all simulation tools, can be improved for both code efficiency, simulation performance, and conciseness.

*5.1.1 Queue model.* The ''Queue'' is a simple model, where a single generator periodically creates output and sends it to the first processor. Each processor sends the output to exactly one other processor, called its successor. There are no loops in the connections, thus forming a single line of processors, as shown in Figure 7. If a processor receives a new event while processing an event, the event is placed in a FIFO queue. After processing an event, the processor will check its queue and pop an event to process.

Our Queue model bears similarity to the HI DEVStone model, in the sense that models are connected to their successors, but also to the LI DEVStone model, as the model
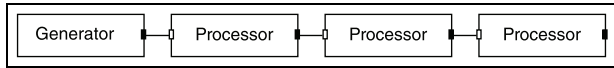
**Figure 7.** "Queue" model: every output port is connected to the input port of the previous model. Shown for three models (excluding the generator).



**Figure 8.** "High Interconnect" model: every output port is connected to every input port of a different model. Shown for four models.
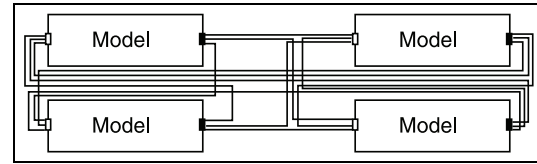
output is only connected to a single other model. We did not completely take over the HI model, as otherwise all models would receive input at the exact same time, thus triggering all external transitions (and later on, their internal transitions) simultaneously. This is an unnatural occurrence in DEVS, which specifically uses a continuous time base. However, a benchmark model with a low number of inter-model connections is necessary. The Queue model is the result of this merge.

It is still interesting to analyze the behavior of the simulation kernel when transition functions are triggered simultaneously (called *collisions* from now on). We allow for either a "full collision" model or a "no collision" model, by defining the time advance function as either fixed (to 1) or random (uniformly distributed between 0 and 2), respectively. If the time advance function always returns a fixed number, all models will transition at exactly the same time. Otherwise, the time advance function returns a random number, preventing most collisions.

*5.1.2 High Interconnect model.* The "High Inter-connect" model is similar to the queue, but has many connections. Every model is connected to every other model, as shown in Figure 8. Each model outputs an event, which is routed to all other models. On reception of an event, the models trigger their external transition function, which simply ignores the message.

This model is a more complex form of the HI DEVStone model, where instead of only one outgoing connection, many outgoing (and incoming) connections are made. The number of connections, and consequently of exchanged events, scales quadratically.

This not only benchmarks the performance in the presence of a high number of connections, but also of routing a single event to a multitude of receivers. A parameter is again provided to define whether or not collisions should happen. If collisions happen, the bag merging algorithms

of Parallel DEVS are also benchmarked, as every model outputs an event to every other model. All events must then be merged into a single bag.

Although this model might seem totally unrealistic, it was added to monitor the efficiency of simulation algorithms where many events were exchanged and processed simultaneously. Whereas other, more realistic, benchmarks could be conceived to monitor this, they would deviate further from the core aspects that are to be monitored: event exchange. By only retaining the actual aspects to be monitored, performance results are less cluttered with other aspects of the model. Additionally, while such models might be ill-suited for DEVS, sometimes DEVS is used for compatibility reasons, as argued by Vangheluwe.[1] Whether or not DEVS is the ideal formalism for this part of the model is then less relevant: it must be used to interact with other models, and even then, high performance is very relevant.

*5.1.3 Traffic model.* The "Traffic" model is a more realistic model.[51] It resembles the Queue model without collisions, though more realistic communication and computation patterns occur.

The Traffic model consists of a generator, some road segments, and a collector, as shown in Figure 9. After a randomly sampled time, a car is generated by the generator. The generator outputs the car and sends it to the connected road segment. Every road segment processes the car for a certain time (depending on the velocity), after which it is sent to the next road segment. A car can accelerate or decelerate, depending on its preferred speed, the speed limit of the road segment, and the cars in front of it. To prevent car collisions, road segments communicate with each other through the use of queries and acknowledgments. As soon as a road segment receives a new car, it
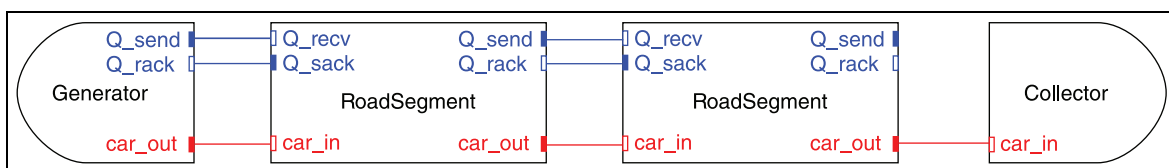


**Figure 9.** "Traffic" model, shown for two segments.

**Table 4.** Lines of code for the benchmarks.

|  | ADEVS | CD++ | DEVS-Suite | MS4 Me | PowerDEVS | PythonPDEVS | VLE | X-S-Y |
|---|---|---|---|---|---|---|---|---|
| Queue | 178 | 169* | 152 | 113* | 234 | 94 | 150* | 104 |
| Highly connected | 112 | 103* | 113 | 57* | 182 | 64 | 73* | 76 |
| Traffic | 609 | 672* | 498 | 380* | 473 | 345 | 599* | 359 |

*Excludes experiment file and coupled models.

sends a query to the next road segment, requesting whether the next road segment is free. It receives an acknowledgment, stating how long it will take for the road segment to become available. The car at the current road segment will adjust its speed accordingly, depending on the maximal acceleration and deceleration values. If a road segment does not receive an acknowledgment in time, the car goes on to the next road segment without adjusting its speed. At the end of the road segments, a collector receives all cars and computes average velocity and average deviation from the preferred velocity. These statistics are used to test the correct implementation of the model in the various simulation tools.

### 5.2 Source code size

For every tool and every benchmark, Table 4 shows the lines of code used to implement the model. This number includes the atomic models, coupled models, and experiment file. Entries marked with an asterisk only count the size of the atomic models, as other parts are constructed graphically (VLE), or by using a verbose syntax (CD++, MS4 Me). DEVS-Suite models do not contain code for experiment setup, as this is done manually by the user.

This analysis does not extend much further than the differences already known[52]: the same behavior in C++ requires more lines of code than in Java, which still requires more than in Python. Recall that MS4 Me uses a natural language-like language. This has a significant impact on the total size of the models, though it seems that Python still requires fewer lines of code. In essence, the natural language is very concise, though the total number of lines of code is still this high because of the use of some Java code in the model. As such, it is more appropriate to compare MS4 Me with DEVS-Suite, as DEVS-Suite models are written completely in Java.

PowerDEVS has the highest number of lines of code; this is due to the (helpful) comments that are included by default in all models. As these benchmark models are relatively small, their overhead becomes significant. For the Traffic benchmark though, the overhead is negligible, resulting in a smaller codebase. CD++ has the most verbose code for the Traffic model, since complex events are expanded: all event attributes are passed separately over their own ports, resulting in more verbose code.

A complete usability study is outside of the scope of this paper, so we limit ourselves to the size of the model. As most tools resort to programming for atomic models, it is best to choose a simulation environment that uses a language with which the modeler is familiar. If the modeler is unfamiliar with any programming language, MS4 Me provides the simplest language to express model behavior.

### 5.3 Remarks

Owing to the comparison of different tools, each with their own design decisions (e.g., about implementation language and required input format), some remarks are required on how these results should be interpreted.

First, different representations are used for coupled models. Tools such as ADEVS use programming language constructs to create a set of models and couple them. This is highly efficient and allows modelers, familiar with programming techniques, to create arbitrary constructions. Tools such as VLE use a graphical environment to create the models. Coupled models are no longer represented with programming language constructs, but using a custom syntax. This custom syntax, like XML in VLE, induces additional overhead during simulation, as it must be parsed. The overhead is only required during initialization, though it imposes an overhead compared with the other tools. Our results include this initialization overhead, as it is required at every simulation run. Tools using compiled models do not have their compilation time included, as compilation is only required once. Compilation times are negligible in long-running simulations, or when a single (compiled) model is used in various settings. During prototyping, compilation times of several seconds, as is the case with ADEVS, offset the higher simulation performance. Some tools also perform some preprocessing of the model, such as direct connection.[53] Whether or not the compilation and initialization overhead is tolerable, compared with the simulation time, is a decision that must be left to the end-user.

Second, DEVS-Suite and MS4 Me do not provide a command line interface, so we are unable to benchmark them automatically. Results for these simulators were obtained by manually starting and stopping the simulation, while measuring the time. As a result, these measurements have a lower accuracy and precision. Being purely

graphical, these simulators continuously update their GUI with simulation information (e.g., current simulation time and number of processed transitions). We disabled as much of these visualizations as possible to obtain our results, though a significant overhead due to this animation is to be expected.

Third, MS4 Me has low performance in all simulation benchmarks. This is possibly because the tool is still in beta status, meaning that some functionality is still missing or not completely implemented, or simply that development effort is currently not focused on performance optimization. For example, all transitions are logged to the console, inducing a significant overhead. A more significant problem is that simulation performance seems efficient at first, but quickly grinds to a halt. Our analysis shows that this results from high memory usage, resulting in very frequent garbage collection, finally even causing out-of-memory errors. This could indicate the presence of a memory leak. Through some undocumented options (namely, manually invoking the generated Java files instead of running them through the GUI) it was possible to avoid this out-of-memory error and significantly increase the performance. Nonetheless, the results that were included here were obtained using this trick, but performance is still significantly lower than other DEVS simulation tools. As a result of these problems, results for MS4 Me are likely to change significantly in the future.

Fourth, as there was a huge variety in simulation performance, we plotted all tools on a logarithmic scale, and the fastest few on a linear scale.

Fifth, results when using PyPy might seem strange in comparison with the other results. This is caused by the JIT, which requires some warm-up first. For small models, the total simulation time is dominated by the interpretation phase and the JIT compiler compiling the code. Therefore, these results are fairly inaccurate for short simulations.

Sixth, CD++, PowerDEVS, and X-S-Y are Classic DEVS simulation tools, whereas the others use Parallel DEVS. Despite identical model behavior, both formalisms mandate different simulation algorithms, causing differences in simulation performance. We argue that they are still comparable, since they are equally expressive, and no simulation tool (except for PythonPDEVS) supports both at the same time. This means that users are restricted to the formalism of the tool they choose.

### 5.4 Queue results

Results for the Queue benchmark are shown in Figure 10. The results indicate that ADEVS is fastest in both cases, with PowerDEVS coming extremely close.

For a random time advance, as in the left part of Figure 10, no collisions occur. For a fixed time advance, as in the right part of Figure 10, collisions always occur. Several
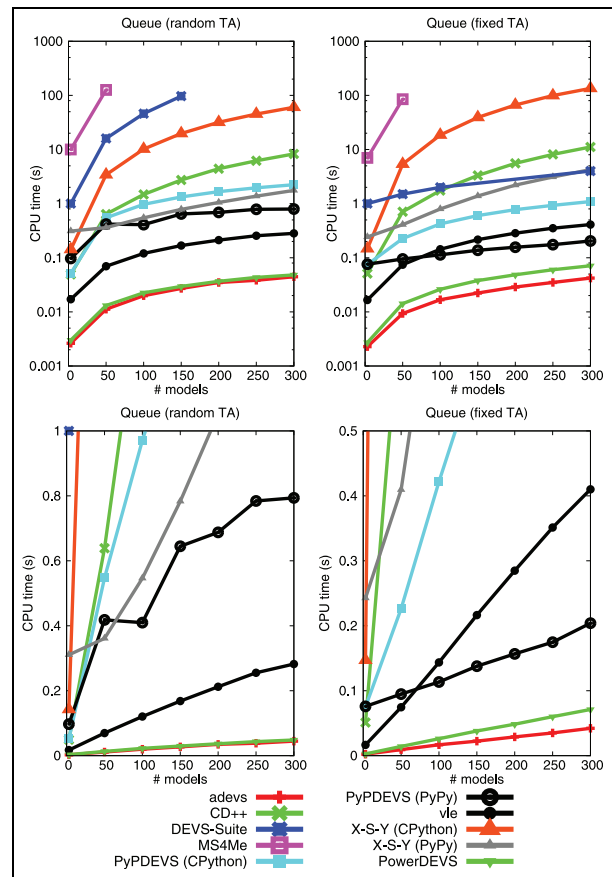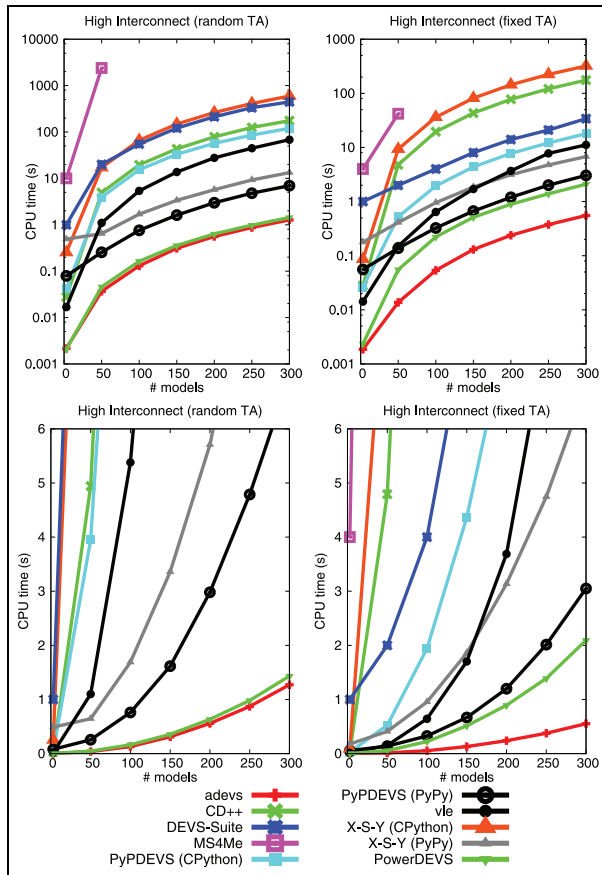


**Figure 10.** Benchmark results for the "Queue" benchmark. Top figures use a logarithmic scale, bottom figures are zoomed in on the fastest tools and uses a linear scale.
TA: time advance.

important differences can be seen when comparing the two.

First, DEVS-Suite is a lot faster with collisions than without collisions. There can be several reasons for this. One option is that the main simulation loop is inefficient (e.g., due to GUI updating), as far fewer simulation steps are executed if all models collide.

Second, PythonPDEVS outperforms VLE in the presence of collisions. This can be explained by the activity-based data structures implemented in PythonPDEVS.[54] As a list-based scheduler would be better in cases where many collisions occur,[22] the data structure modifies itself to a list-based one. VLE and ADEVS have a static, heap-based, scheduler, which is not optimized for this situation.

Third, using PyPy instead of CPython has a significant impact on both PythonPDEVS and X-S-Y. With CPython, X-S-Y is far slower than PythonPDEVS, though they come much closer when using PyPy. This indicates that much of the performance improvement of PythonPDEVS with CPython is gained as the result of CPython-aware optimizations (e.g., expensive function calls), which are largely mitigated using PyPy (e.g., using JIT inlining).
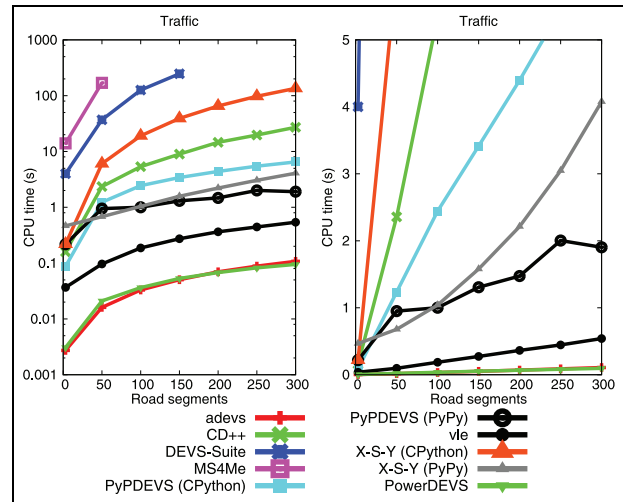
**Figure 11.** Benchmark results for the "High Interconnect" benchmark. Top figures use a logarithmic scale, bottom figures are zoomed in on the fastest tools and uses a linear scale. TA: time advance.



**Figure 12.** Benchmark results for the "Traffic" benchmark. The left figure uses a logarithmic scale, whereas the right figure is zoomed in on the fastest tools and uses a linear scale.

Fourth, ADEVS and PowerDEVS come very close. This probably indicates that they implement similar algorithmic optimizations, and that the compiler optimizes most implementation details.

### 5.5 High interconnection results

Results for the High Interconnection benchmark are shown in Figure 11. Again, ADEVS is clearly the fastest, with PowerDEVS coming close for a random time advance.

For PowerDEVS, the reason for this slower performance mostly lies with the use of Classic DEVS, where other tools (except CD++ and X-S-Y) use Parallel DEVS. In the presence of simultaneous events, Classic DEVS models trigger their *select* function, selecting a single model. For $n$ colliding models, as is the case with a fixed time advance, this causes $n$ separate lookups in the scheduling data structures, explaining the difference. This same occurrence is aggravated here, as every model must trigger its external transition once for every other model. Each external transition is called $n$ times more in Classic

DEVS simulation kernels, owing to the formalism. As external transitions contain nearly no computation, this effect is not clearly visible. Because this was not the case in the previous benchmark, the difference between Classic DEVS and Parallel DEVS was relatively small.

VLE performs much more slowly than expected—even slower than X-S-Y (using PyPy)—certainly in the absence of simultaneous events. With simultaneous events, the same behavior can be seen, though VLE is now much closer to X-S-Y, and PythonPDEVS further distinguishes itself from the others. We expect this low performance to be due to the high number of connections, which must be parsed at simulation time.

DEVS-Suite again shows better results in the presence of simultaneous events, just as in the previous benchmark.

### 5.6 Traffic results

Figure 12 shows the results of the Traffic benchmark. Results are very similar to those for the Queue model, as the basic principles are the same. Once again, ADEVS and PowerDEVS are tied, followed by VLE. Some differences are visible because the transition functions are now more complex, and more event passing happens.

### 5.7 Conclusions on performance

Our analysis has shown that ADEVS is currently the fastest for Parallel DEVS, and PowerDEVS for Classic DEVS. The performance of these tools is unmatched in any kind of model we have tried, leaving competitors behind by a fair margin. However, this performance comes at the cost of functionality and debuggability. Debugging

ADEVS models, written in C++, combined with ADEVS's meager debugging capabilities, is significantly more work than with other simulation tools. For example, there is no tracing functionality included by default. PowerDEVS includes a minimal simulation tool in the model, but at least contains a debugging feature that traces the methods being invoked.

VLE comes fairly close in more realistic models, but loses terrain when the model has many connections. While this is a fairly uncommon case, it is remarkable that this simulation kernel falls that far behind in this specific configuration. The performance gap with ADEVS is most probably caused by the use of an XML-based language for coupled models, which needs to be parsed at simulation time.

PythonPDEVS comes in third for the Queue model, and clearly profits from its activity-based scheduler, as it comes quite close to VLE for models with many collisions. For models with a high number of connections, it comes in second, owing to the suddenly degraded performance of VLE. Combined with the use of an interpreted language, thus not needing compilation, PythonPDEVS is useful for prototyping, even of large-scale models. Simulation using CPython is clearly slower than using PyPy, though even simulation using CPython is not among the slowest.

X-S-Y comes in fourth, though sometimes falls behind, owing to the use of Classic DEVS. The performance gap between CPython and PyPy is much larger than it was for PythonPDEVS. PythonPDEVS has many built-in optimizations that avoid slow paths in CPython, reducing the potential speedup of PyPy. Such optimizations are still important, as many users will use CPython instead of PyPy. This is certainly true for users who do not seek maximal performance.

CD++ comes in fifth, even though it is implemented in C++. A likely explanation is the use of a custom language for coupled models, which is parsed at simulation time.

DEVS-Suite seems to be inefficient in a few situations, though fairly efficient in some others. These results are probably related to the GUI, which was constantly updating the textual values, even with the graphical simulation view disabled. Other simulation tools do not provide the same level of detail on the running simulation, nor do they offer visualization of the running simulation.

Lastly, MS4 Me could only be used for relatively small models, owing to low performance. Without fixing the memory leak issue, even these relatively small models could not be simulated until their termination time.

## 6  Conclusions

We have shown that many differences exist between DEVS simulators. Differences ranged not only between functionality and performance, but also between the programming languages used for modeling. The lack of a standardized DEVS representation, in a programming language-independent form, causes models to be incompatible between simulators. Owing to the arbitrary cost of porting models between different simulation tools, and the different programming languages, simulation tool lock-in can occur. This makes the initial choice of tooling an important decision. Our comparison aims to aid in this important decision.

The final decision on which is the "best" tool is dependent on the requirements set out by the team that will be using the tool. Performance analysis showed the expected results: low-level simulation tools achieve significantly higher performance, at the cost of reduced readability and (debugging) functionality. We briefly summarize each tool, and give a recommended target audience.

- *ADEVS* offers a limited set of features, but allows for very efficient simulation of Parallel DEVS models. It is, however, difficult to use by nonprogrammers as it boils down to programming in C++. We would recommend ADEVS to modelers who are familiar with C++ and wish to obtain every bit of performance, at the cost of functionality.
- *CD++* is mainly specialized in the simulation of Cell DEVS models by nonprogrammers. While Classic DEVS models are also supported, extensions such as CD++Builder are recommended for nonprogrammers. We would recommend the use of CD++ for the development of Cell DEVS models, or in combination with CD++Builder.
- *DEVS-Suite* presents a nice simulation environment that provides much insight in the semantics of Parallel DEVS models. Because of its additional features, performance is rather slow, making it unsuited for large-scale simulations. We would recommend DEVS-Suite for educational purposes, where the steps of DEVS simulation must be thoroughly explained.
- *MS4 Me* provides an intuitive modeling and simulation environment that tries to hide programming at the level of both atomic and coupled models. The availability of consulting options distinguishes it from the rest, though it is still in beta testing and performance is insufficient for any reasonably sized model. We would recommend MS4 Me only once the tool has stabilized in terms of performance, after which it is a tool appropriate for nonprogrammers.
- *PowerDEVS* offers an integrated modeling environment, though it still relies on the modeler writing C++ code. Its performance is often on a par with ADEVS, though it only supports Classic DEVS, despite Parallel DEVS being the more popular formalism nowadays. We would recommend PowerDEVS to modelers familiar with C++, but

not so familiar with DEVS itself, or those seeking to use DEVS for the simulation of hybrid systems.

- *PythonPDEVS* provides users with a DEVS simulator in Python, offering features that are relevant to beginning users of DEVS. While performance is decent compared with most other tools, it is vastly outperformed by other efficient simulation tools. We would recommend the use of PythonPDEVS for educational purposes (owing to its close compliance to the formalism), or for prototypes (owing to its relatively efficient implementation in Python).
- *VLE* provides an integrated modeling environment similar to PowerDEVS, but remains at a more basic level. There is, however, more support for the creation of experiments and execution on multiple machines. We would recommend VLE when a single environment is desired for every operation: from writing utility functions in C++ to running experiments with various configurations.
- *X-S-Y* is unique in that it offers support for a verifiable subset of DEVS, implemented in Python. Performance is lacking though, making it unfit for large-scale models. We would recommend X-S-Y if model verification is important.

Some aspects were left out of our comparison, such as an analysis of parallel and distributed simulation performance, or a more detailed performance comparison (e.g., memory usage). Usability evaluation of the tools is an important aspect that we did not tackle. Our intuition tells us that graphical tools can make up for their low performance, by significantly reducing required training and model development time. The same goes for debuggability: tools offering advanced debugging capabilities are orders of magnitude slower, though model errors are probably found earlier. In future, performance analysis of the parallel and distributed simulation should be considered, as well as usability evaluation of these tools.

## Funding

## References

1. Vangheluwe H. DEVS as a common denominator for multi-formalism hybrid systems modelling. In: *IEEE international symposium on computer-aided control system design*, Anchorage, USA, 25–27 September 2000, pp.129–134. Piscataway, NJ: IEEE.
2. Bonaventura M, Wainer G, and Castro R. Graphical modeling and simulation of discrete-event systems with CD++Builder. *Simulation* 2013; 89(1): 4–27.
3. Vicino D, Niyonkuru D, Wainer G, et al. Sequential PDEVS architecture. In: *Proceedings of the 2015 spring simulation multiconference*, Alexandria, USA, 12–15 April, 2005, pp.906–913. San Diego, CA: SCS.
4. Gutierrez-Alcaraz M and Wainer G. Experiences with the DEVStone Benchmark. In: *Proceedings of the 2008 spring simulation multiconference*, Ottawa, Canada, 13–16 April 2008, pp.447–455. San Diego, CA: SCS.
5. Wainer G, Glinsky E and Gutierrez-Alcaraz M. Studying performance of DEVS modeling and simulation environments using the DEVStone benchmark. *Simulation* 2011; 87: 555–580.
6. Li X, Vangheluwe H, Lei Y, et al. A testing framework for DEVS formalism implementations. In: *Proceedings of the 2011 spring simulation multiconference*, Boston, USA, 3–7 April 2011, pp.183–188. San Diego, CA: SCS.
7. Franceschini R, Bisgambiglia PA, Touraille L, et al. A survey of modelling and simulation software frameworks using Discrete Event System Specification (ed. R Neykova and N Ng). In: *2014 Imperial College computing student workshop*, London, UK, 25–26 September 2014, pp.40–49. Wadern, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik.
8. Nikoukaran J, Hlupic V, and Paul RJ. Criteria for simulation software evaluation. In: *Proceedings of the 1998 winter simulation multiconference*, Washington DC, USA, 13–16 December 1998, pp.399–406. Piscataway, NJ: IEEE.
9. Glinsky E and Wainer G. DEVStone: a benchmarking technique for studying performance of DEVS modeling and simulation environments. In: *Proceedings of the 2005 9th IEEE/ACM international symposium on distributed simulation and real-time applications*, Montreal, Canada, 10–12 October 2005, pp.265–272. Piscataway, NJ: IEEE.
10. Zeigler BP, Praehofer H, and Kim TG. *Theory of modeling and simulation*. 2nd ed. San Diego, CA: Academic Press, 2000.
11. Chow ACH and Zeigler BP. Parallel DEVS: a parallel, hierarchical, modular, modeling formalism. In: *Proceedings of the 1994 winter simulation multiconference*, Orlando, USA, 11–14 December 1994, pp.716–722. San Diego, CA: SCS.
12. Barros FJ. Dynamic structure discrete event system specification: a new formalism for dynamic structure modeling and simulation. In: *Proceedings of the 1995 winter simulation multiconference*, Arlington, USA, 3–6 December 1995, pp.781–785. Piscataway, NJ: IEEE.
13. Chow ACH, Zeigler BP, and Kim DH. Abstract simulator for the parallel DEVS formalism. In: *AI, simulation, and planning in high autonomy systems*, Gainesville, USA, 7–9 December 1994, pp.157–163. Piscataway, NJ: IEEE.
14. Himmelspach J and Uhrmacher AM. Sequential processing of PDEVS models. In: *Proceedings of the 3rd European modeling & simulation symposium*, Barcelona, Spain, 4–6 October 2006, pp.239–244. Rende, Italy: CAL-TEK S.r.l.
15. Barros FJ. Modeling formalisms for dynamic structure systems. *ACM Trans Model Comput Simul* 1997; 7: 501–515.
16. Barros FJ. Abstract simulators for the DSDE formalism. In: *Proceedings of the 1998 winter simulation multiconference* (ed. DJ Medeiros, EF Watson, JS Carson, et al.) Washington,

DC, 13–16 December 1998, pp.407–412. Piscataway, NJ: IEEE.

17. Uhrmacher AM. Dynamic structures in modeling and simulation: a reflective approach. *ACM Trans Model Comput Simul* 2001; 11: 206–232.

18. Wainer G and Giambiasi N. Timed cell-DEVS: modeling and simulation of cell spaces. In: Sarjoughian HS and Cellier FE (eds) *Discrete event modeling and simulation technologies*. New York: Springer-Verlag, 2001, pp.187–214.

19. Troccoli A and Wainer G. Implementing parallel Cell-DEVS. In: *Proceedings of the 2003 spring simulation symposium*, Orlando, USA, 30 March–2 April 2003, pp.273–280. Piscataway, NJ: IEEE.

20. Nutaro JJ. adevs, http://www.ornl.gov/~1qn/adevs/ (2015, accessed 16 November 2016).

21. Muzy A and Nutaro JJ. Algorithms for efficient implementations of the DEVS & DSDEVS abstract simulators. In: *1st open international conference on modeling and simulation (OICMS)*, Clermont-Ferrand, France, 12–15 June 2005, pp.273–279.

22. Van Tendeloo Y and Vangheluwe H. The modular architecture of the Python(P)DEVS simulation kernel. In: *Proceedings of the 2014 spring simulation multiconference*, Tampa, USA, 13–16 April 2014, pp.387–392. San Diego, CA: SCS.

23. Fritzson P, Aronsson P, Lundvall H, et al. The OpenModelica modeling, simulation, and software development environment. *SNE* 2005; 44: 8–16.

24. Wainer G. CD++: a toolkit to develop DEVS models. *Softw Pract Exp* 2002; 32: 1261–1306.

25. Muzy A and Wainer G. Comparing simulation methods for fire spreading across a fuel bed. In: *Proceedings of AIS'2002*, Lisbon, Portugal, 7–10 April 2002, pp.219–224. San Diego, CA: SCS.

26. Shang H and Wainer G. A model of virus spreading using Cell-DEVS. In: Sunderam VS, van Albada GD, Sloot PMA, et al. *Computational Science ICCS 2005* (*Lecture Notes in Computer Science*, vol. 3515). Berlin: Springer, 2005, pp.373-377.

27. Wainer G and Giambiasi N. Application of the Cell-DEVS paradigm for cell spaces modelling and simulation. *Simulation* 2001; 76: 22–39.

28. Kgwadi M, Shang H, and Wainer G. Definition of dynamic DEVS models: dynamic structure CD++. In: *Proceedings of the 2008 spring simulation multiconference*, Ottawa, ON, Canada, 13–16 April 2008, pp.10:1–10:4. San Diego, CA: SCS.

29. Kim S, Sarjoughian HS, and Elamvazhuthi V. DEVS-Suite: a simulator supporting visual experimentation design and behavior monitoring. In: *Proceedings of the 2009 spring simulation multiconference* (ed. GA Wainer), San Diego, USA, 22–27 March 2009, pp.161:1–161:7. San Diego, CA: SCS.

30. Sarjoughian H and Zeigler B. DEVSJava: basis for a DEVS-based collaborative M&S environment. *Simulation* 1998; 30: 29–36.

31. Chezzi CM, Tymoschuk AR, and Lerman R. A method for DEVS simulation of e-commerce processes for integrated business and technology evaluation (WIP). In: *Proceedings of the 2013 spring simulation multiconference*, San Diego, USA, 7–10 April 2013, pp.13:1–13:6. San Diego, CA: SCS.

32. Palaniappan S, Sawhney A, and Sarjoughian HS. Application of the DEVS framework in construction simulation. In: *Proceedings of the 38th winter simulation conference*, Monterey, USA, 3–6 December 2006, pp.2077–2086. Piscataway, NJ: IEEE.

33. Ferayorni AE and Sarjoughian HS. Domain driven simulation modeling for software design. In: *Proceedings of the 2007 summer computer simulation conference*, San Diego, USA, 15–18 July 2007, pp.297–304. San Diego, CA: SCS.

34. Seo C, Zeigler BP, Coop R, et al. DEVS modeling and simulation methodology with MS4Me software. In: *Proceedings of the 2013 spring simulation multiconference*, San Diego, USA, 7–10 April 2013, pp.33:1–33:8. San Diego, CA: SCS.

35. Kim T, Lee C, Christensen E, et al. System entity structuring and model base management. *IEEE Trans Syst Man Cybern* 1990; 20: 1013–1024.

36. Zeigler BP, Seo C, Coop R, et al. Creating suites of models with System Entity Structure: global warming example. In: *Proceedings of the 2013 spring simulation multiconference*, Virginia Beach, USA, 23–26 April 2013, pp.32:1–32:8. San Diego, CA: SCS.

37. Zeigler B, Seo C, and Kim D. System entity structures for suites of simulation models. *Int J Model Sim Sci Comput* 2013; 4: 1340006.

38. Bergero F and Kofman E. PowerDEVS: a tool for hybrid system modeling and real-time simulation. *Simulation* 2011; 87: 113–132.

39. Kofman E, Lapadula M, and Pagliero E. PowerDEVS: A DEVS-based environment for hybrid system modeling and simulation. Technical report, School of Electronic Engineering, Universidad Nacional de Rosario, 2003.

40. Capocchi L, Santucci JF, Poggi B, et al. DEVSimPy: a collaborative Python software for modeling and simulation of DEVS systems. In: *Workshop on enabling technologies: infrastructure for collaborative enterprises*, Paris, France, 27–29 June 2011, pp.170–175. Piscataway, NJ: IEEE.

41. Van Mierlo S, Van Tendeloo Y, Barroca B, et al. Explicit modelling of a Parallel DEVS experimentation environment. In: *Proceedings of the 2015 spring simulation multiconference*, Alexandria, USA, 12–15 April 2015, pp.860–867. San Diego, CA: SCS.

42. Van Tendeloo Y and Vangheluwe H. PythonPDEVS: a distributed Parallel DEVS simulator. In: *Proceedings of the 2015 spring simulation multiconference*, Alexandria, USA, 12–15 April 2015, pp.844–851. San Diego, CA: SCS.

43. Quesnel G, Duboz R, Ramat E, et al. VLE: a multimodeling and simulation environment. In: *Proceedings of the 2007 summer simulation multiconference*, San Diego, USA, 15–18 July 2007, pp.367–374. San Diego, CA: SCS.

44. Hwang MH. X-S-Y, https://code.google.com/p/x-s-y/ (2012, accessed 16 November 2016).

45. Tewoldeberhan TW, Verbraeck A, Valentin E, et al. An evaluation and selection methodology for discrete-event simulation software. In: *Proceedings of the 2002 winter simulation multiconference*, San Diego, CA, 8–11 December 2002, pp.67–75. Piscataway, NJ: IEEE.

46. Fujimoto RM. *Parallel and distribution simulation systems*. 1st ed. New York, NY: John Wiley & Sons, 1999.

47. Glinsky E and Wainer G. Definition of real-time simulation in the CD++ toolkit. In: *Proceedings of the 2002 summer simulation multiconference*, San Diego, USA, 14–18 July 2002. San Diego, CA: SCS.

48. Martin D, McBrayer T, Radhakrishan R, et al. Time warp parallel discrete event simulator. Technical report, Computer Architecture Design Laboratory, University of Cincinnati, USA, 1997.

49. Barroca B, Mustafiz S, Van Mierlo S, et al. Integrating a neutral action language in a DEVS modelling environment. In: *Proceedings of the 8th international ICST conference on simulation tools and techniques*, Athens, Greece, 24–26 August 2015, pp.19–28. Brussels, Belgium: ICST.

50. Sun Y and Hu X. Partial-modular DEVS for improving performance of cellular space wildfire spread simulation. In: *Proceedings of the 2008 winter simulation multiconference*, Miami, USA, 7–10 December 2008, pp.1038–1046. Piscataway, NJ: IEEE.

51. Posse E. *Modelling and simulation of dynamic structure discrete-event systems*. PhD Thesis, School of Computer Science, McGill University, 2008.

52. Ousterhout JK. Scripting: higher-level programming for the 21st century. *Computer* 1998; 31: 23–30.

53. Chen B and Vangheluwe H. Symbolic flattening of DEVS models. In: *Proceedings of the 2010 summer simulation multiconference*, Ottawa, Canada, 11–14 July 2010, pp.209–218. San Diego, CA: SCS.

54. Van Tendeloo Y and Vangheluwe H. Activity in PythonPDEVS. In: *Proceedings of ACTIMS 2014*, Zurich, Switzerland, 16–18 January 2014, pp.2:1–2:10. Red Hook, NY: Curran Associates, Inc.

## Author biographies

**Yentl Van Tendeloo** is a PhD student at the Department of Mathematics and Computer Science, University of Antwerp, Belgium.

**Hans Vangheluwe** is a full professor at the Department of Mathematics and Computer Science, University of Antwerp, Belgium. He is also an adjunct professor at McGill University, School of Computer Science, Montréal, Canada, where he was previously a full professor.