Debugging Parallel DEVS

Reference:

Van Mierlo Simon, Van Tendeloo Yentl, Vangheluwe Hans.- Debugging Parallel DEVS
Simulation - ISSN 0037-5497 - (2016), p. 1-28
Full text (Publishers DOI): http://dx.doi.org/doi:10.1177/0037549716658360

Debugging Parallel DEVS

Journal Title
XX(X):1–28

DOI: 10.1177/ToBeAssigned
www.sagepub.com/

Simon Van Mierlo[1], Yentl Van Tendeloo[1] and Hans Vangheluwe[1,2]

Abstract

To this day, debugging support for the DEVS formalism has been provided, at best, in an ad-hoc way. The intricacies of dealing with the interplay of different notions of (simulated) time, the formalism semantics, and user input, has not been thoroughly investigated. This paper presents a visual modelling, simulation, and debugging environment for Parallel DEVS, which builds on a theoretical foundation for debugging DEVS models. We take inspiration from both the code debugging and the simulation world to model our environment; we transpose a set of useful code debugging concepts onto Parallel DEVS, and combine those with simulation-specific operations, such as as-fast-as-possible simulation and (scaled) real-time execution. Apart from these common debugging operations, we introduce new features to the debugging of Parallel DEVS models, such as "god events", which can alter the model state during simulation, and reversible debugging, which allows to go back in time. To achieve this, the PythonPDEVS simulator is de- and reconstructed: the modal part of the simulator/debugger, as well as the debugging operations, are modelled using the Statecharts formalism. These models are combined, resulting in a model of the timed, reactive behaviour of a debuggable simulator for Parallel DEVS. The code for the simulator is automatically synthesized from this model. To improve usability, we combine the simulator with a visual modelling environment, allowing for visual and interactive live debugging.

Keywords

Debugging; Parallel DEVS; Statecharts

Submission for the *Special Issue of Simulation: Modelling and Simulation in the Era of Big Data and Cloud Computing: Theory, Framework and Tools.*

Received: 30-Nov-2015
Revised: 02-May-2016
Accepted: 17-May-2016

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

Corresponding author:
Simon Van Mierlo
Department of Mathematics and Computer Science
Middelheimlaan 1
2020 Antwerpen, Belgium
Email: simon.vanmierlo@uantwerpen.be

Prepared using sagej.cls *[Version: 2015/06/09 v1.01]*

One Page Description

Simulation debugging is an emerging field, as debuggers for a variety of formalisms are introduced. The debugging support, however, is often ad-hoc and not based on a sound theoretical basis. The intricacies of dealing with the interplay of different notions of (simulated) time, the formalisms semantics, and user input, has not been thoroughly investigated. In this paper, we construct a visual modelling and debugging environment for the Parallel DEVS formalism. We take inspiration from both the code debugging world and the simulation world to arrive at a set of useful debugging operations. We use a technique called de- and reconstruction to instrument the behaviour of the simulator. This allows us to deal with the inherent complexity of this task by explicitly modelling this behaviour. Compared to the state-of-the-art DEVS tools, we support an unmatched set of debugging operations. Moreover, explicitly modelling the behaviour of the debugger allows for a better understanding and extendibility.

The most advanced modelling and visualization tool for DEVS that we know of is presented in [1] by Maleki et al. The authors introduce a new way of visualization for both the design of the DEVS system, as the execution trace. They attempt to convey the meaning of the models by making the simulation process known to the end-user. They do not, however, offer an extensive set of debugging operations with which to manipulate the simulation process at runtime.

MS4Me [2] by Seo, Zeigler, Coop, and Kim, as well as DEVS-Suite [3] by Kim, Sarjoughian, and Elamvazhuthi have (ad-hoc) debugging support in the form of stepping, injecting events, and visualization.

The initial idea for de- and reconstruction was introduced in [4] by Mustafiz and Vangheluwe, where a debugging environment for Statecharts was constructed.

1 Introduction

Programmers spend a large portion of their time debugging the code they write [5]. This is supported by a variety of debugging techniques such as pause/resume, breakpoints, stepping over functions, tracing runtime variables, etc. More recently, advanced techniques such as program slicing, algorithmic debugging, and omniscient debugging attempt to speed up the debugging process. This has lead to what Zeller calls "scientific debugging", where programmers try to find the source of an observable error by systematically formulating and refuting hypotheses [6].

The systems we analyse, design, and develop today are characterized by an ever growing complexity. Modelling and Simulation (*M&S*) [7] become increasingly important enablers in the development of such systems, as they allow rapid prototyping and early validation of designs. Domain experts, such as automotive or aerospace engineers, build models of the system being developed and subsequently simulate them having a set of "goals" or desired properties in mind. Ideally, every aspect of the system is modelled at the most appropriate level(s) of abstraction, using the most appropriate formalism(s) [8]. The M&S approach can only be successful if there is sufficient tool support, including debuggers that enable locating modelling errors. Often, models are used to generate code, so an obvious and frequently used option is to use traditional code debuggers to debug the generated code. That implies, however, that the modeller has to make a context switch, and errors in the generated code might be difficult to link back to model elements. There is thus a need for specialized model debugging environments at the most appropriate level of abstraction, using the most appropriate notations and operations.

The construction of such a debugging environment is an inherently complex task. The interplay of formalism execution semantics, different notions of simulated time (*e.g.*, scaled real-time and as-fast-as-possible), as well as user interaction through an interface, are all challenging to capture and implement correctly using traditional code-centric software development techniques. In this paper, we present a visual modelling, simulation, and debugging environment for Parallel DEVS. To manage the inherent complexity, we de- and reconstruct the PythonPDEVS simulator [9] and add debugging support to its modal part (which is explicitly modelled in the Statecharts [10] formalism) of the simulator. This is based on our previous work [11], where we introduce the de- and reconstruction approach for instrumenting model simulators, and [12], where we first explore debugging for the Parallel DEVS formalism, and which is extended in this work. We combine the simulator with the visual modelling tool AToMPM [13], allowing for the visual interactive control of DEVS model simulations. Although we chose two research tools with which we are familiar, the same technique can be applied to other environments and simulators with similar capabilities.

Structure Section 2 provides the necessary background on which the rest of the paper builds. Section 3 explores related work. Section 4 gives an overview of how Parallel DEVS models can be debugged, taking inspiration from code debugging and simulation. Section 5 explains the technique used for modelling the debuggable simulator. Section 6 explains how we couple this simulator to a visual modelling environment, to create a fully interactive, visual debugger. Section 7 compares our visual debugger to other DEVS modelling and simulation tools. Section 8 concludes the paper and makes suggestions for future work.

2 Background

We introduce the reader to the two formalisms used throughout the remainder of this paper: Parallel DEVS and Statecharts. We also briefly survey traditional code debugging techniques, as we transpose these onto useful debugging techniques for Parallel DEVS.

2.1 Parallel DEVS

DEVS, as introduced by Zeigler [14], is a popular formalism for modelling 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 [15]. A popular extension (and the default in many simulation tools) of Classic DEVS is Parallel DEVS [16], which has better support for parallelism. A number of tools have been constructed by academia and industry that allow the modelling, simulation, and in some cases, (limited)

debugging of DEVS models. The basic building blocks of Parallel DEVS are *atomic DEVS* models, which are structures of the form

$$M =< X, Y, S, \delta_{int}, \delta_{ext}, \delta_{conf}, \lambda, ta >$$

where the *input set* X denotes the set of admissible inputs of the model. X is a structured set $X = \times_{i=1}^{m} X_i$ where X_i denotes the admissible inputs on port i. The *output set* Y denotes the set of admissible outputs of the model. Y is a structured set $Y = \times_{i=1}^{l} Y_i$ where Y_i denotes the admissible outputs on port i. The *state set* S is the set of sequential states. The *internal transition function* $\delta_{int} : S \to S$ defines the next sequential state, depending on the current state. The *output function* $\lambda : S \to Y^b$ maps the sequential state set onto an output bag. The *external transition function* $\delta_{ext} : Q \times X^b \to S$ with $Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$ gets called whenever an *external input* ($\in X$) is received. The *time advance function* $ta : S \to \mathbb{R}^{+}_{0,+\infty}$ defines the simulation time the system remains in the current state before triggering its *internal transition function*. The *confluent transition function* $\delta_{conf} : S \times X^b \to S$ is called if both an internal and external transition collide at the same simulation time, replacing both functions.

A network of atomic DEVS models is called a *coupled DEVS* model. Output ports of one atomic DEVS model can be connected to one or more input ports of other atomic DEVS models using "channels", optionally defining a *transfer function* to translate output to input messages. Parallel DEVS is closed under coupling, which means that a coupled model can always be expressed in terms of an atomic model. As such, a coupled model can be used in place of an atomic model, making it possible to nest models of arbitrary depth.

The operational semantics of Parallel DEVS are described in [17], in the form of an abstract simulator. A simulator computes the next state of the system (a "step") until its end condition is satisfied. Each step consists of the following phases:

1. Compute the set of atomic DEVS models whose internal transitions are scheduled to fire (imminent components).
2. Execute the output function for each imminent component, causing events to be generated on the output ports.
3. Route events from output ports to input ports, translating them in the process by executing the transfer functions.
4. For each atomic DEVS model, determine the type of transition to execute, depending on it being imminent, receiving input, or both.
5. Execute, in parallel, all enabled internal, external, and confluent transition functions.
6. Compute, for each atomic DEVS model, the time of its next internal transition (specified by its time advance function).

During simulation, the value of an internal clock (the simulated time) is updated according to the information encoded in the transition functions. We look into the details of the simulator, including simulated time, in the next section.

2.2 Statecharts

The Statecharts formalism was introduced by David Harel [10]. Statecharts is an extension of state machines and state diagrams, with hierarchy, orthogonality, and broadcast communication. Its basic building blocks are *states* that are connected to each other trough *transitions*, encoding how the system behaves: a transition specifies to which event it conditionally reacts, and optionally which event should be raised when the transition is fired. *Hierarchical states* group a number of states, of which one is the default. When the hierarchical state is entered, its default state is entered as well. *Orthogonal states* specify behaviour that is executed concurrently and are contained by a hierarchical state. Its semantics are defined by flattening the orthogonal components (*i.e.*, taking the cross products of the states of all hierarchical components). Different parts of a Statechart communicate with each other through events: these can either be local to the enclosing hierarchical state, or global (*i.e.*, broadcast communication). Statecharts are used for the specification and design of complex discrete-event systems, and is popular for the modelling of reactive systems, such as graphical user interfaces. A Statechart generally consists of the following elements:

- states, either basic, orthogonal, or hierarchical;
- transitions between states, either event-based or time-based;
- actions, executed when a state is entered or exited;
- guards on transitions, modelling conditions that need to be satisfied in order for the transition to "fire";
- history states, a memory element that allows the state of the Statechart to be restored.

We use Statecharts to model the behaviour of our debuggable simulator, described in Section 5.

2.3 Code Debugging Techniques

Debugging program code is facilitated by numerous debugging techniques and operations. The most popular operations are listed below.

- **State Inspection and Manipulation** operations allow to inspect the runtime state as the program is executed. This includes the runtime stack, current value of the instruction pointer, and variable values. Debuggers often allow the programmer to alter the state of a program when its execution is paused, by, for example, changing the value of a variable. Quick assessments can then be made about how a change affects the behaviour of the program.
- **Breakpoints** allow to automatically pause when a condition on the runtime state is satisfied. These conditions can be arbitrary, for example "the value of a equals 5 and the value of b is less than 10". To improve efficiency, however, most debuggers require to specify a specific line of code on which to pause.
- **Steps** allow to deviate from the usual run-to-completion semantics of program code. When a program is paused (either manually by the programmer, or automatically as the result of a breakpoint triggering), a user might choose to step through individual computation steps. Common operations include
  - *single step* (or *step into*) pauses on the next statement regardless of scope;
  - *step over* pauses on the next statement in the current scope;
  - *step out* pauses on the next statement in the scope from which the currently executing function was called.

  Steps are generally used to provide fine-grained control over program execution, allowing the programmer to inspect how each statement, or sequence of statements, affects the runtime state. More recently, **omniscient debugging** techniques were added, allowing to "go back in time" by offering reverse steps [18]. These techniques are generally based on tracing the program execution.

In Section 4, we transpose these techniques onto Parallel DEVS.

3 Related Work

Model debugging and simulation debugging has only recently received increasing attention from the research community. Code debugging techniques, however, are well-established and a significant research body is dedicated to them. While new contributions to code debugging are made continuously, they are not the focus of this paper. Instead, we refer to Zeller's book [6], which provides an excellent overview of existing code debugging techniques. The book additionally explains how to combine these techniques in a "scientific debugging" process to find, manage, and remove defects in software in an optimal way.

We believe debugging support for modelling and simulation has to be provided at the most appropriate level of abstraction (*i.e.*, using the abstractions of the formalism, instead of relying on low-level code). In [19], the authors explore requirements for modelling and simulations tools that support verification, validation and testing. One of those is the need to present concepts at the domain-specific level. Debuggers for some other formalisms already exist. In [4], Mustafiz and Vangheluwe construct a debugging environment for Statecharts, by instrumenting the Statecharts model with appropriate transitions. We take inspiration from their approach, but generalize their technique to apply it to other formalisms as well. A debugger for Modelica was developed in [20]. Model transformation debugging, in particular omniscient debugging techniques, were explored by Corley in [21].

Closely related are debugging techniques for Model-Driven Development (MDD), and more specifically the Unified Modelling Language (UML), a family of (executable) languages for specifying the structure and behaviour of software systems. In [22], the authors map code debugging concepts onto the Story Diagrams formalism, and build a visual debugging user interface in the open-source development environment Eclipse. Their architecture consists of a debugging client (with which the user interacts) that communicates with a debugging server, which controls the execution of the interpreter. Both [23] and [24] extend the fUML—a subset of the UML for which precise semantics are defined—with debugging support.

Methods specific to the debugging of Domain-Specific Languages (DSLs) have also been researched. In [25], Mannadiar and Vangheluwe address the need for debugging models in DSLs and propose a mapping of code debugging concepts to model-based design. A notable work that attempt to generalize techniques for adding debugging support to DSLs is the Moldable Debugger [26], a reusable framework for developing debuggers for DSLs. It allows to implement a set of debugging operations such as stepping, state querying, and visualization at the most appropriate (domain-specific) level of abstraction. In [27], the authors describe a partly generic debugger that can be extended with domain-specific trace management functions. They allow the definition of a set of debugging operations that traverse, query, and manage these execution traces. We take inspiration from their work to map code debugging operations onto domain-specific debugging operations (in our case, specific to Parallel DEVS).

We contribute to this emerging field by developing a debugger for Parallel DEVS. In particular, we define a set of useful debugging operations, based on existing code debugging techniques as well as simulation-specific operations. We add these operations to the existing PythonPDEVS [9] simulation kernel using a generic technique which we call the de- and reconstruction of the simulation kernel. We couple it to a (basic) interactive visual user interface. In [1], a set of visual interfaces for working with Classic DEVS models is introduced. The goal is to convey the meaning of the models intuitively, by providing a visual notation for the design, but also the execution trace of the simulation. This allows model debugging through the visual inspection of this trace. Closely related is the work by Kemper [28], who presents a method for debugging stochastic models based on a visualization of their trace, from which irregular patterns can be discerned. Our work is complementary, as we focus on dynamic debugging techniques using different types of steps, execution modes, breakpoints, etc. Our visual interface displays limited information, and, because of our modular architecture, can be replaced by a more advanced one.

4 Debugging DEVS Models

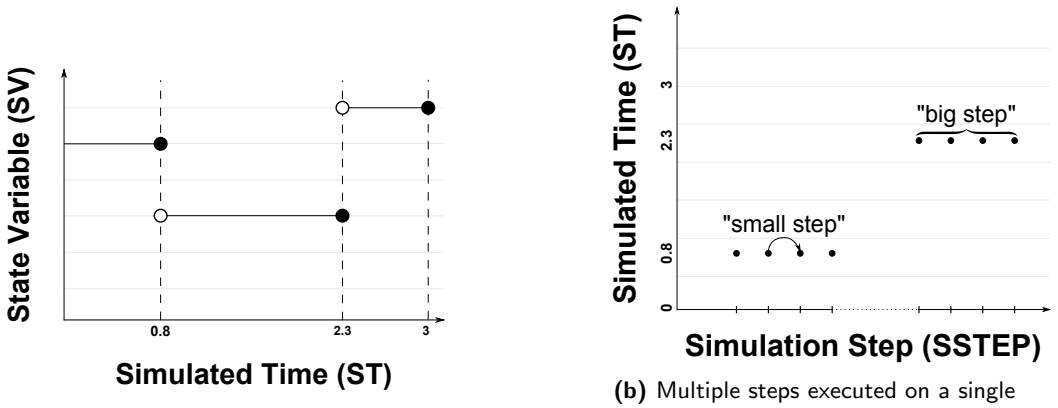
This section explores the different ways in which Parallel DEVS models can be debugged. We assume an operational view: the model is simulated through the use of an external simulator, instead of compilation to code.

When experimenting with the system, the modeller runs a set of simulation and observes their result. When the modeller observes an erroneous result, he might—similar to what a programmer would do—want to inspect the dynamics of the simulation. An appropriate model debugger should offer functionality similar to code debuggers: stepping, pausing, setting breakpoints, etc. Furthermore, formalism-specific debugging operations should be provided, especially if the model exhibits real-time behaviour.

We first discuss the different notions of time, and how time can be properly paused. Next, state inspection and manipulation, steps, and lastly textual tracing during simulation is discussed.

4.1 Time

The notion of *time* plays a prominent role in simulation [29]. Simulated time differs from the wall-clock time: it is the internal clock of the simulator, instead of the time in the real world. In general, a simulator updates some state variable vector, which keeps track of the current simulation state, each time increment. In contrast to wall-clock time, simulated time can be arbitrarily updated. This is shown visually in Figure 1a. The state is updated by some computations, transitioning from one consistent state to the other. All computation required between these two consistent states is called a "step". For each of these steps, a number of smaller computational steps may be involved. This is visualized in Figure 1b, where one "big step" is broken up into



(a) Change of the state variable over time.

(b) Multiple steps executed on a single simulation time instance.

Figure 1. Simulation time and steps.

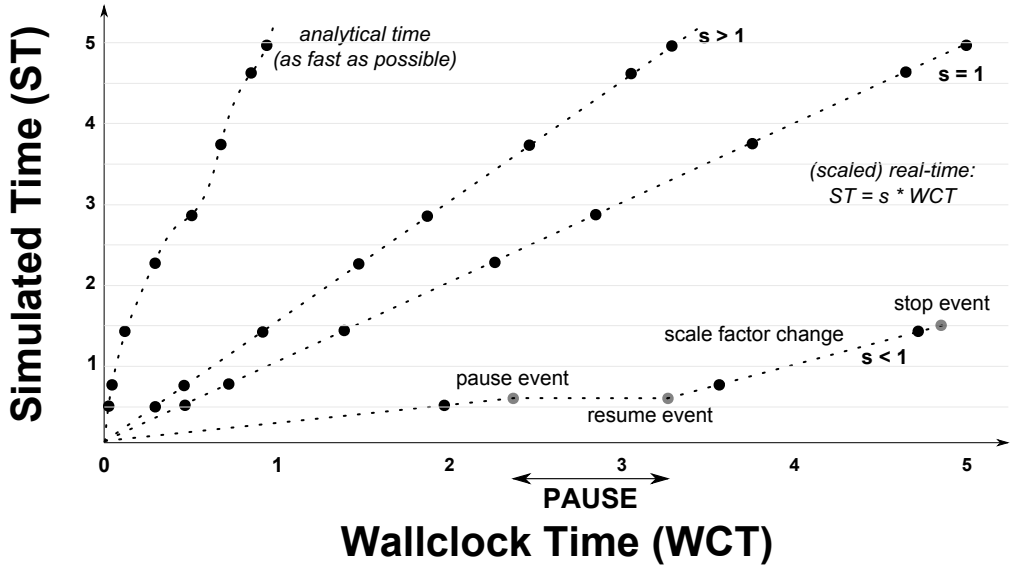


Figure 2. Different notions of simulated time.

a number of "small steps". Note that the simulated time stays constant in between small steps, and only increases after a big step has been completed. While the state is guaranteed to be consistent after a big step has completed, this is not the case for small steps.

In contrast to program code, which is always executed as fast as possible (*i.e.*, the speed of the program is limited by the resources of the machine executing it), simulated time can have different relations to the wall-clock time, shown in Figure 2. In as-fast-as-possible simulation, there is no relation between simulated time and wall-clock time, meaning that simulated time is simply a variable in the simulator. In real-time simulation (useful when simulating models of real-time systems), simulated time is synchronized with the wall-clock time. This implies that the simulation steps have a hard real-time deadline (*i.e.*, the values of the runtime variables have to be computed before the wall-clock time reaches the simulated time). A scale factor s can be applied to speed up or slow down simulation, while maintaining the linear relation between simulated time and wall-clock time. Additionally, operations such as pausing or stepping back can be performed on simulated time, but are not allowed on wall-clock time.

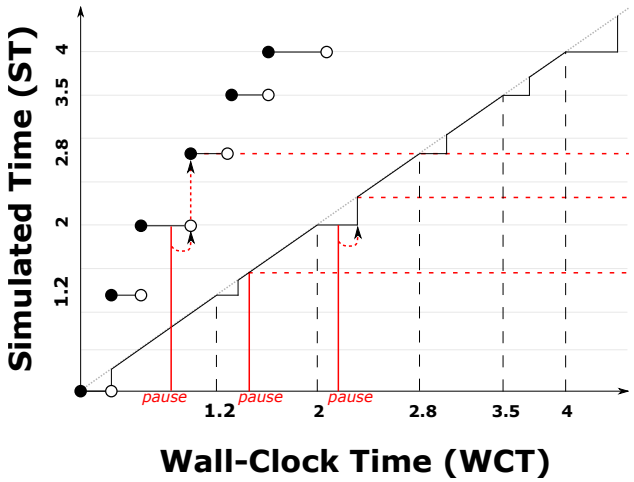


Figure 3. Pausing simulation: difference between as-fast-as-possible and real-time simulation.

4.2 Pausing

Pausing is a useful debugging operation, as it allows to interrupt a running program or simulation and inspect the current state of the system. A pause can either be manually requested by a user, or triggered automatically as a result of a breakpoint. The breakpoint specifies a condition on the runtime state, which, as soon as it evaluates to true, pauses the simulation.

While pausing is possible in real-time simulation, an important distinction is made between real-time and as-fast-as-possible simulation. This is shown visually in Figure 3: during as-fast-as-possible simulation (visualized by the stepwise function), the simulated time is incremented as quickly as the executing system allows it to. The horizontal parts represent the computation time necessary to compute the state after the next "big step". In as-fast-as-possible simulation, the computations are executed one after the other, without any waiting period in between. If a pause is requested (denoted by the red vertical bar with the label "pause"), the simulator is always busy, computing the next state. Processing the pause is then delayed until the step is completed (and the value of the simulated time has been updated). The pause cannot be executed immediately, as the system would be in an inconsistent state—it would still be executing the smaller computation steps. The simulated time at which simulation actually pauses is indicated by the horizontal red dotted line.

In real-time simulation, simulated time is "synchronized" with the wall-clock time. Computing the next state still takes up a certain amount of (wall-clock) time, but the simulator will wait in between these computation periods, in order to synchronize the simulated time with the wall-clock time. This is represented by the continuous function in Figure 3, which tries to follow the ideally synchronized line, represented by the grey dotted line. When a computation is performed, the wall-clock time advances whereas simulated time remains constant. As a result, the simulated time is desynchronized from the wall-clock time, as represented by the horizontal parts in the function. Time is synchronized again as soon as the computation is finished, as depicted by the vertical parts in the figure. A pause requested during such an idle period can thus be processed immediately, as the state is consistent. The result is that the system will be in the "current" state, and not the "next", as was the case with as-fast-as-possible simulation. The simulation time will be somewhere in between the previous transition time and the next.

Real-time simulation can pause at virtually every point in simulated time, whereas as-fast-as-possible simulation can only pause at time points where transitions happen. The notable exception being the time at which transition functions are being computed. This is visualized in Figure 3 by a pause request in the middle of a state update. The simulation is paused when the computation finishes.

4.3 State Inspection and Manipulation

The state of a Parallel DEVS system is the combination of the states of its components. Each atomic DEVS models is exactly in one state at the same time. To inspect this state, it needs to be communicated to the user during simulation. We again need to differentiate between different simulation modes. In as-fast-as-possible simulation, the goal of the user is to execute the model as quickly as the system allows, either to reach the end of simulation or trigger a breakpoint. Showing the state of the system after each "big step" might then not be ideal, as it significantly slows down simulation, and does not allow for easy state inspection, as the execution speed will still be too fast to follow the execution dynamics. In real-time simulation, however, it is more useful to show the state after each "big step": it will convey the meaning of the model to the user more easily, and if the simulation is still executed too quickly, the user can adjust the scale factor to slow it down. In a stepped simulation, as much information as possible should be passed to the user, possibly even the intermediate (inconsistent) states.

Upon pausing the simulation, the user might observe that the model is in an unexpected state. It might then be useful to force changes to the system state, to observe the effects. We differentiate between two ways of manipulating the state:

- *God events* allow to change the value of a state variable directly. The name is chosen as it is an "outside force" that changes the state, instead of the internal dynamics of the model. Upon receiving such an event, the state is modified and the time of the next transition is recomputed. Instead of computing the new next transition time based on the current simulation time, the simulation time of the last transition is used. This way the modification is as least invasive as possible.
- *Event injection* allows to manually give an external input to the model at a particular simulated time. This triggers an external transition in the model, changing the state of the system.

Both methods have their advantages and disadvantages: god events allow direct manipulation of the state, though are fairly invasive. Event injection, on the other hand, is restricted to the interface supplied by the model.

4.4 Steps

We differentiate between three different ways of stepping through the simulation of a Parallel DEVS model: "big step", "small step", and "step back".

Algorithm 1 The Parallel DEVS simulator's "main loop".

```
1: time ← 0
2: s ← INITIALIZE_STATE(model)
3: while not end_condition do
4:    imm ← COMPUTE_IMMINENTS(s)
5:    events ← []
6:    for comp in imm do
7:       events+ = OUTPUT_FUNC(i)
8:    end for
9:    ROUTE_EVENTS(events)
10:   ext ← COMPUTE_EXTERNALS(s, events)
11:   conf ← COMPUTE_CONFLUENTS(s, imm, ext)
12:   for comp in imm + ext + conf do
13:      COMPUTE_NEXT_STATE(comp, s)
14:   end for
15:   time ← TIME_NEXT(s)
16: end while
```

Before we explain the semantics of each operation, we first need to explain the simulation algorithm of Parallel DEVS, as implemented by the PythonPDEVS simulation kernel. The algorithm is shown in Algorithm 1. It consists of two main parts:

- **Initialization**: lines 1-2 set the simulated time to 0 and initialize the state.
- **Main Loop**: lines 3-16 contain the "meat" of the algorithm: it advances the state of the system until the end condition is satisfied. One iteration of the *while* loop brings the system from one consistent state to the next, which we call a "big step". During a big step, however, the simulation goes through a series of phases. First, it computes which atomic DEVS models will execute their internal transition at this point in simulated time. These are called the *imminent* models. Then it computes the output generated in each imminent component. Next, these outputs are routed over the connections between components, translating them according to the transfer function. The algorithm then checks which transition function should be executed, and marks the models accordingly. A model can be in the set of models executing their internal, external, or confluent transition function. Some models might be inactive at this point in time, and are ignored in the remainder of the step. All transition functions are then executed, computing the new state of each component. Finally, the algorithm computes the time advance for each of these models, and determines the earliest time at which an internal transition function is scheduled.

We will now explain the granularity of each kind of step.

4.4.1 Big Step To see system behaviour on a fine-grained level, a user might step through the simulation to dynamically see the state evolving. We see this "big step" as the minimum amount of computation for bringing the system from one consistent state to the next (*i.e.*, one iteration of the *while* loop in Algorithm 1).

4.4.2 Small Step For even more fine-grained control over the simulation, modellers might be interested in seeing the effects of the different phases in the simulation algorithm. We see a "small step" as the execution of one of the six phases in the *while* loop in Algorithm 1. Note that the simulation is in an inconsistent state while "small stepping". It is only when the last phase is executed that the system is in a valid configuration again.

4.4.3 Step Back Stepping back in time causes the simulation to revert to the state before the last big step. This is realized through state saving: every consecutive state is stored in memory. When requested, previous states are put in place again, and the global simulation state is reverted. This works fairly similar as optimistic distributed simulation (*e.g.*, Time Warp [30]), where simulations are able to "roll back" to a previous point in time. As in Time Warp, the history of all states might become too large to completely store in memory. Techniques such as reverse computation [31] or incremental state saving [32] tackle this problem.

4.5 Tracing

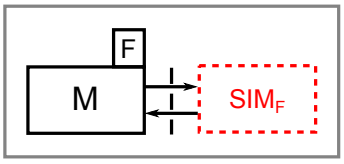
The previous techniques were concerned with "live debugging", which debugs the model during simulation. While this is useful, it is sometimes necessary to get a full overview of the system's execution history after simulation has finished. We call this *post-mortem* debugging.

This is possible by keeping a trace of the system state during simulation, which is analyzed after simulation. Many simulators, including the PythonPDEVS simulation kernel, already support the textual tracing of the execution history.

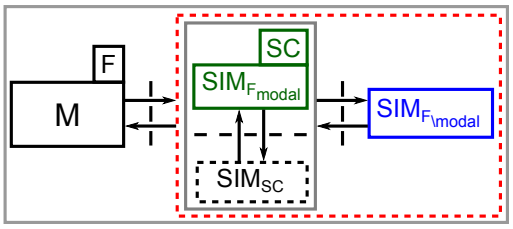
5 Method

In this section, the process for adding debugging support to the PythonPDEVS simulation kernel is explained. We call this technique the de- and reconstruction of the simulator, which is generic and can be applied to simulators for any formalism. In general, each simulator has a "main simulation loop", often implemented in programming code. The first step of our modelling method, *deconstruction*, consists of extracting the "modal" part of the simulator and model it explicitly. Modelling the control flow of the simulator explicitly facilitates adding debugging support. The most appropriate formalism to describe this timed, reactive, autonomous behaviour is a state machine. More specifically, we model the behaviour in the Statecharts language, as it allows the description of both reactive and autonomous (timed) behaviour.

The workflow for de- and reconstructing the simulator is shown in Figure 4 and Figure 5. Figure 4a shows a model created in a formalism F and an appropriate (coded) simulation kernel for formalism F. The simulation



(a) A model M in formalism F and a simulation kernel SIM_F for F.



(b) De- and reconstructing the simulator.

Figure 4. The de- and reconstruction process.

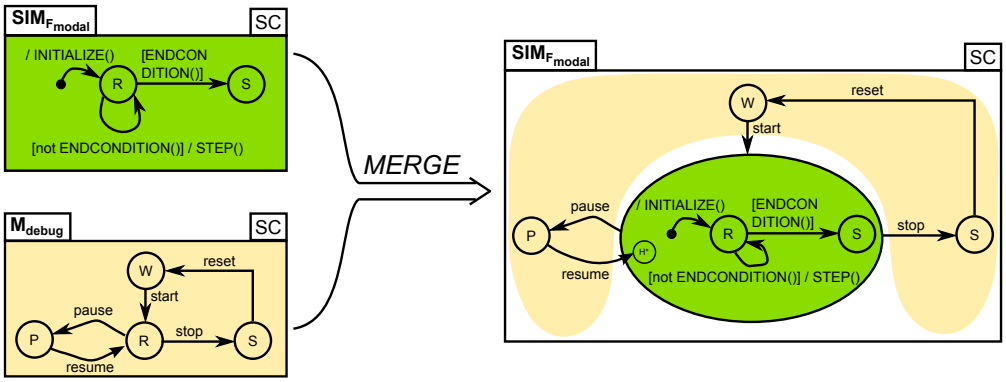


Figure 5. Merging the debugging concepts with the modal behaviour of the simulator.

kernel interacts with the model through an interface (shown as a dashed line in the figure), modifying and querying its state. This combination of model and simulation kernel can be seen as a black box, which is given input signals and produces output signals.

The de- and reconstruction process is shown in Figure 4b. The first step, deconstrucing the simulator, extracts the modal part of the simulator in a Statecharts (SC) model called $SIM_{F_{modal}}$. This model is combined with a Statecharts simulator, interpreter, or compiler called SIM_{SC} to give it operational semantics. The combination of the modal and non-modal part of the simulator results in a behaviourally equivalent simulator to the original SIM_F. From the user's point of view, the black box containing the model to be simulated and its simulator remains unchanged.

In Figure 5, the last step in creating a simulator which supports debugging is shown. We *merge* the modal part of the simulator for F with a Statecharts model of the debugging operations. This results in an instrumented model of the modal behaviour of the simulator. The last step is to replace $SIM_{F_{modal}}$ in Figure 4b with this instrumented model. Again, this does not change the behaviour of the simulator in any way if the user does not make use of the debugging functionality. Extra behaviour has been added, but running the simulator as before is still possible.

In the presented example, the debugger includes the concepts of *start*, *pause*, *resume*, and *stop*. The simulator only has two states: *running* (R), and *stopped* (S). It runs the main loop of the simulator until the end condition is satisfied, signalling that the simulation is done. This is a trivial (and fictional) example, but it demonstrates the process of de- and reconstruction.

The result of reconstructing the simulator is an instrumented version of the original simulator, enriched with debugging capabilities. From this model, the code for a debuggable simulator for formalism F can be automatically generated. In the next subsections, we explain how this process is applied to a Parallel DEVS simulator.

5.1 Code to Statechart

To add debugging support to the PythonPDEVS simulator using the de- and reconstruction approach, we first convert the modal part of the simulator to a Statecharts model to replace the code implementing the control flow of the simulation algorithm. The non-modal part of the simulator consists of computation functions written in Python. These functions are extracted from the source code of the PythonPDEVS simulation kernel. While they are possibly structured differently compared to the original kernel, to accomodate for the change to Statecharts to model the control flow, their behaviour remains unaltered. By doing this, we ensure the that the Statecharts version of the simulation kernel is behaviourally equivalent to the original simulation kernel, and can be used not only for debugging, but also for experimenting with the model.

Figure 6 presents our Statecharts model. It consists of two orthogonal components: `simulation_state` and `simulation_flow`.

The `simulation_state`, which determines the current state of the simulation, is the only part that listens to user events. Therefore, it can also be seen as a description of the interface, showing how the user can influence the simulation. There are three different simulation states: `realtime`, `paused`, and `continuous`. Simulation starts in the `paused` state, and will change to another state depending on the user event received. In either of these states, the simulation state goes back to `paused` as soon as termination is detected. This detection is encoded in the `simulation_flow` orthogonal component, discussed below. Additionally, based on the simulation state, the behaviour of the simulation flow will change. Through the explicit modelling in Statecharts, the behaviour of the simulation algorithm is deduced easily. For example, it is explicitly clear what would happen if a realtime simulation is running and a request for continuous simulation is received (in this case, nothing). Without explicit modelling, this would be a detail that required looking at the simulation code, or trying it out in the running tool.

The `simulation_flow` orthogonal component determines the flow of a single simulation step, and explicitly encodes the invocation order of the non-modal functions. This is an explicit representation of the control flow that was implemented in the `while` loop in the coded implementation. The orthogonal component consists of two main components: `check_termination` and `do_simulation`. In the `check_termination` state, it is checked whether or not simulation should continue. This check inspects the state of the orthogonal component `simulation_state`, and, if realtime simulation is selected, also checks whether the next simulation step can already be performed, or if the simulation algorithm needs to wait. In case we should wait for the next simulation step, the `wait` state is entered. Otherwise, the `do_simulation` state is entered. The `do_simulation` composite state simply traverses through 6 stages (encoded as states), each responsible for a part of the simulation algorithm. This traversal of phases cannot be interrupted by the user, therefore all transitions are unconditional. Linking back to the original code implementation, the `check_termination` state is the condition of the `while` loop, whereas `do_simulation` is the body of the loop.

5.2 Augmenting the Statechart

With our simulation kernel deconstructed into a modal and non-modal part, we are able to augment it with debugging operations. We include all features listed in Section 4. Most notably, it becomes possible to interrupt a running simulation in a variety of ways (*e.g.*, pause, modify state, or inject events). Implementing these operations using traditional, code-centric approaches, would add accidental complexity to the already inherently complex implementation task. Using Statecharts makes this process easier: manual concurrency management is replaced with constructs native to the Statecharts formalism. We need not worry about locks or race conditions, as this is handled transparently.

The augmented Statecharts model is shown in Figure 7. Many features are added to the simulation kernel, though the overall structure of the Statecharts model remains the same. The original behaviour is still preserved, and there are mostly only additional outgoing transitions, as well as orthogonal components. This means users of the augmented simulation kernel can choose to ignore the debugging features, and experiment with their simulations as was possible before with the uninstrumented simulation kernel. There is therefore no need to switch between two tools (a "simulation" and a "debugging" tool).

The most significant change is in the `simulation_flow` state, which has some new transitions. Most of these transitions deal with the small step logic: a small step is a paused simulation, but simulation is advanced fully

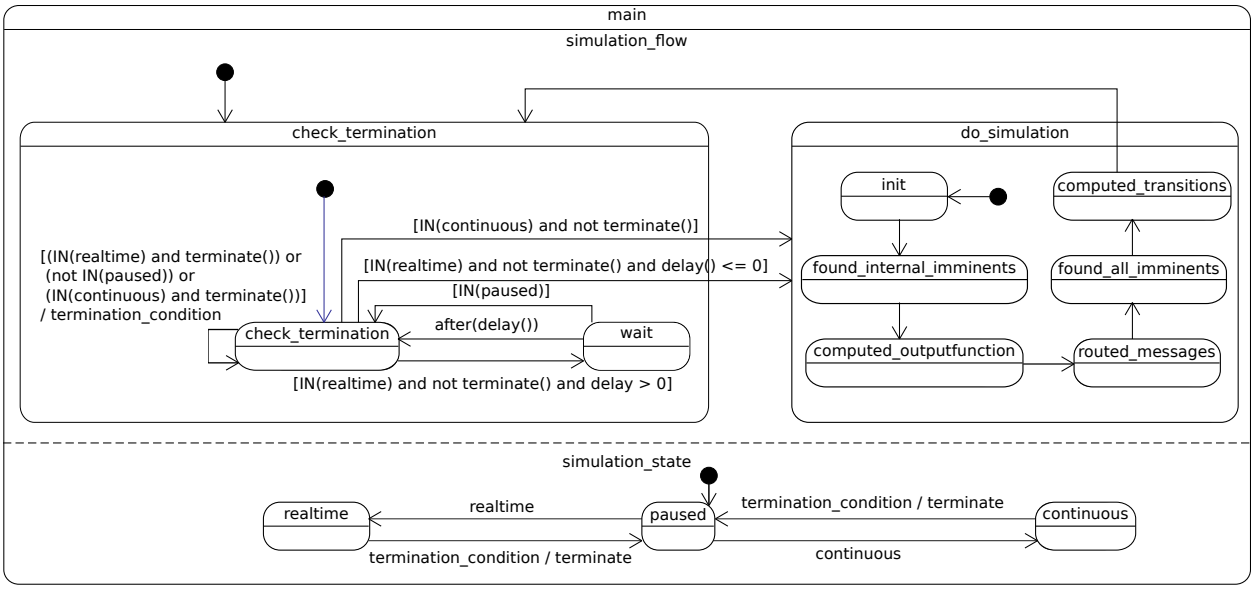


Figure 6. PythonPDEVS Statechart

manual by user inputs caught in the `simulation_state` orthogonal component. In the `check_termination` composite state, some new transitions were also necessary to cope with god events and stepping back in time. By adding these transitions here, we ensure god events and stepping back in time are only possible when a simulation "big step" is currently not executing. Instead of only a termination check, there are now also checks for breakpoints, which are implemented almost identically as the termination condition. The difference is that a breakpoint has an associated identifier, which is communicated to the modelling environment, making it possible to highlight the one that caused the pause. Pausing a realtime simulation also requires some modifications, though most of this is handled in the non-modal part. When it is determined that simulation can progress, we transition to the `do_simulation` orthogonal component. The outline is still rather similar, but instead of doing an unconditional transition, we now require that simulation is either not paused, or that a small step is requested. When progress happens due to a small step, we send out additional information about the effects of the small step. Otherwise, we simply go to the next phase. At the end of the `do_simulation` state, there are three possibilities, each with a different response to the user. If simulation is running in realtime or a big step was being executed, all information is passed on to the user. Should simulation be done using a small step, only a part of the information needs to be sent, as all other information has already been sent during the previous steps. If simulation is running as fast as possible, no information is sent out, as it would only slow down simulation.

The `simulation_state` orthogonal component is augmented with a new state: `big_step`. This is a third way to make the simulation progress: when a big step is requested, the simulation advances to the next consistent simulation state, and then pauses. Small steps are implemented as a paused simulation with manual triggering of transitions. Only continuous and realtime simulation respond to the pause event, as a big step cannot be interrupted (it would leave the model in an inconsistent state). The pause event, however, does not transition immediately to the `paused` state: the current simulation step first needs to be finished to make sure that the model is left in a consistent state. As such, the pause event only makes sure that a subsequent termination check terminates the simulation.

The remaining changes are new orthogonal components, which listen to user events and call the respective non-modal function. They are orthogonal to the previous two orthgonal components, as they work in any possible simulation state. While not strictly necessary to all be orthogonal components, we did so for clarity, as all of these are conceptually orthogonal to each other. Four kinds of orthogonal components are added: `inject`, which receives an event that needs to be injected in the model's input queue; `trace`, which outputs

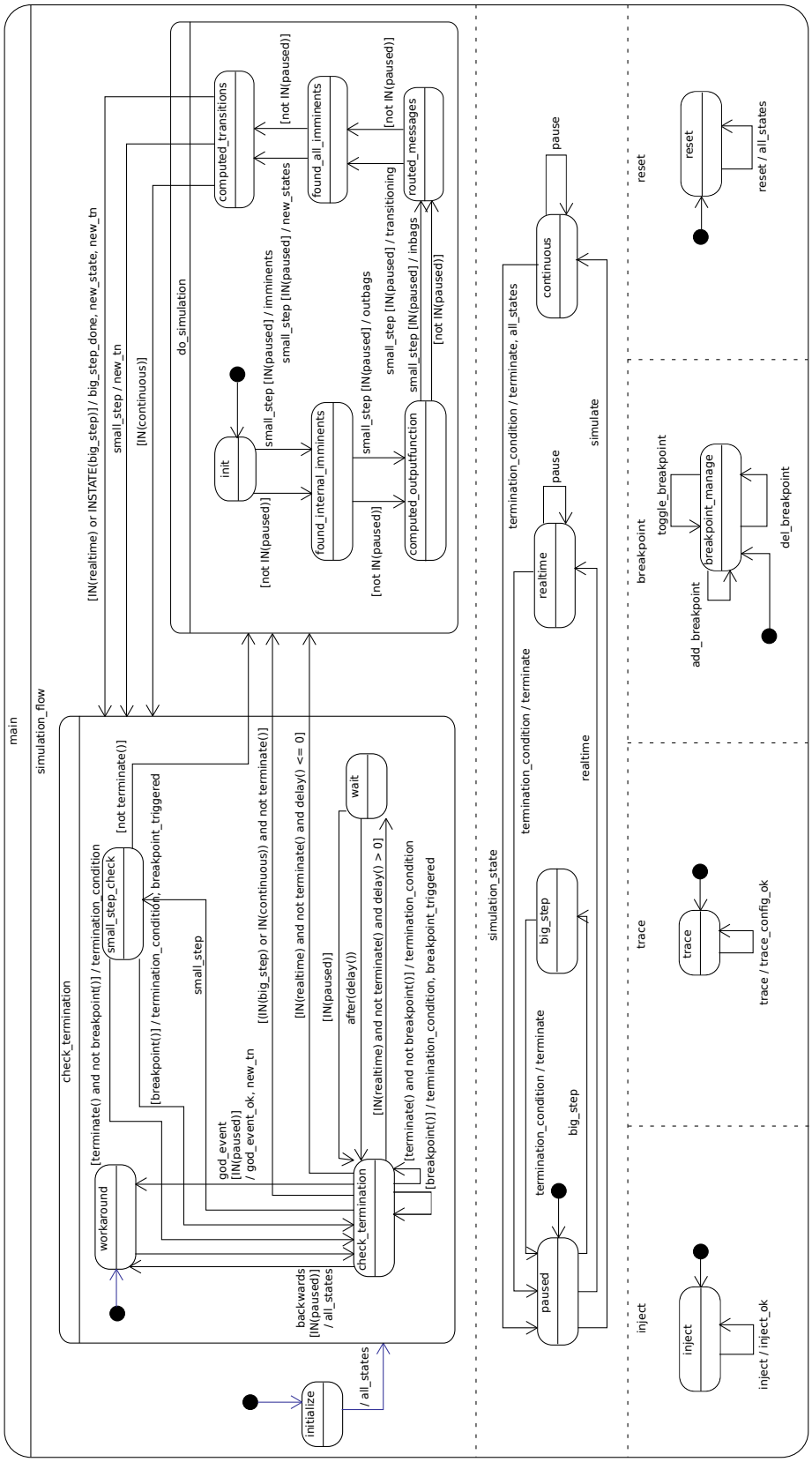


Figure 7. PythonPDEVS Statechart with debugging functionality

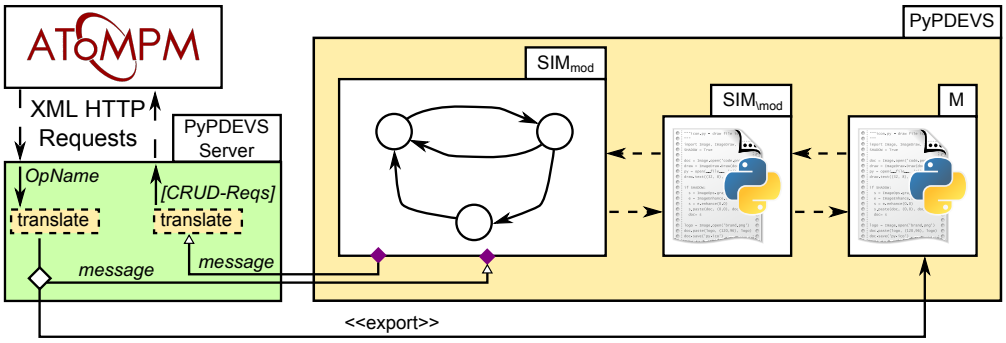


Figure 8. Architecture.

a textual trace; `breakpoint`, which adds, deletes, or toggles a breakpoint; and `reset`, which resets the simulation.

From this model, the code for the debuggable simulator can be generated using an appropriate Statecharts compiler.

6 User Interface

The previous sections explained how a debuggable simulator can be obtained from a code simulator in a structured way using the de- and reconstruction approach. Until this point, we assumed the model to be simulated was readily available in a compiled format which the PythonPDEVS kernel can understand. Moreover, we have not yet defined how the user interacts with the simulator. In this section, we explain how the debuggable simulator can be combined with an existing visual modelling interface, resulting in an interactive visual modelling, simulation, and debugging interface. First, we explain the architecture of our solution. Then, we explain how Parallel DEVS models can be visually modelled, simulated, and debugged.

6.1 Architecture

The architecture of our solution is shown in Figure 8. On the top left, the visual modelling tool AToMPM [13] is shown, which is used to create Parallel DEVS models. We extend it with several languages, explained in the following subsections. AToMPM allows for two-way communication, which is vital to our approach. Any other modelling tool can be substituted, as long as its interface supports at least Create, Read, Update, and Delete (CRUD) requests to alter the model, as well as functions to highlight elements. These operations are required to modify the visual representation of the model during simulation and debugging. The tool should also offer a way for the user to send requests to the simulator.

A server is responsible for translating requests coming from AToMPM to input events for the simulator, and translating output events from the simulator to requests sent to AToMPM.

6.2 Modelling DEVS

AToMPM allows to generate domain-specific modelling environments from a language definition comprising of abstract syntax (in a metamodel) and its concrete syntax representation. We use this mechanism to define a visual language for designing Parallel DEVS models. It defines the following concepts:

- Atomic DEVS models, which have a set of sequential states interconnected with internal, external, and confluent transition functions. Each state defines the output generated when leaving the state, as well as the time to stay in the state before the internal transition function is executed. Each atomic DEVS model has an interface defined by a number of named input and output ports. Furthermore, each atomic DEVS model has a number of runtime variables, which are part of its state.

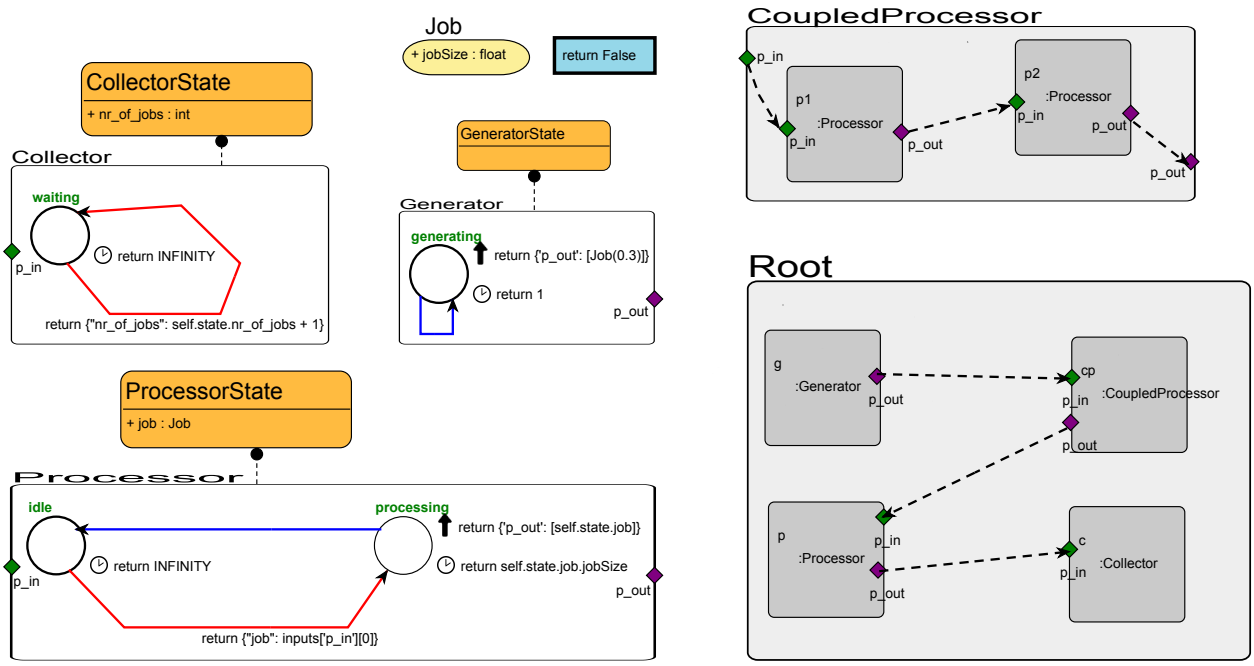


Figure 9. The design model of the producer-consumer DEVS model.

- Coupled DEVS models, which contain a number of instances of DEVS models (either atomic or coupled) that are connected through their interfaces. The top-level *Root* coupled DEVS model has no interface and is used for simulation.
- Event templates, which define the events that can be sent and received by DEVS models.
- The end condition, which defines when the simulation should halt.

In Figure 9, an example producer-consumer model is shown in the generated AToMPM modelling environment. It consists of the following atomic DEVS models:

- **Generator** outputs a new Job each 0.3 seconds on its output port.
- **Collector** accepts a Job on its input port and counts the number of messages it has received.
- **Processor** accepts a Job on its input port, does some computation on it for a predetermined amount of time, and then outputs the Job on its output port.

A **CoupledProcessor** consists of two connected processors, and the *Root* is composed of a generator, a coupled processor, a processor, and a collector. A *Job* event template is defined, and the end condition is simply 'false', meaning that the simulation will run forever.

A separate visual modelling language was created to represent a Parallel DEVS model at runtime (*i.e.*, during simulation). This is necessary for a variety of reasons:

- The need to keep track of runtime information, such as:
  - The current simulation time.
  - The current state (for each atomic DEVS model).
  - The attribute values for states and atomic/coupled DEVS models.
  - The time at which each atomic DEVS model will transition next.
  - Configured breakpoints, and whether they are enabled or not.
- To instantiate the references to coupled DEVS models. This causes these references to be expanded, making the complete structure explicit.
- To display instances of events being passed.

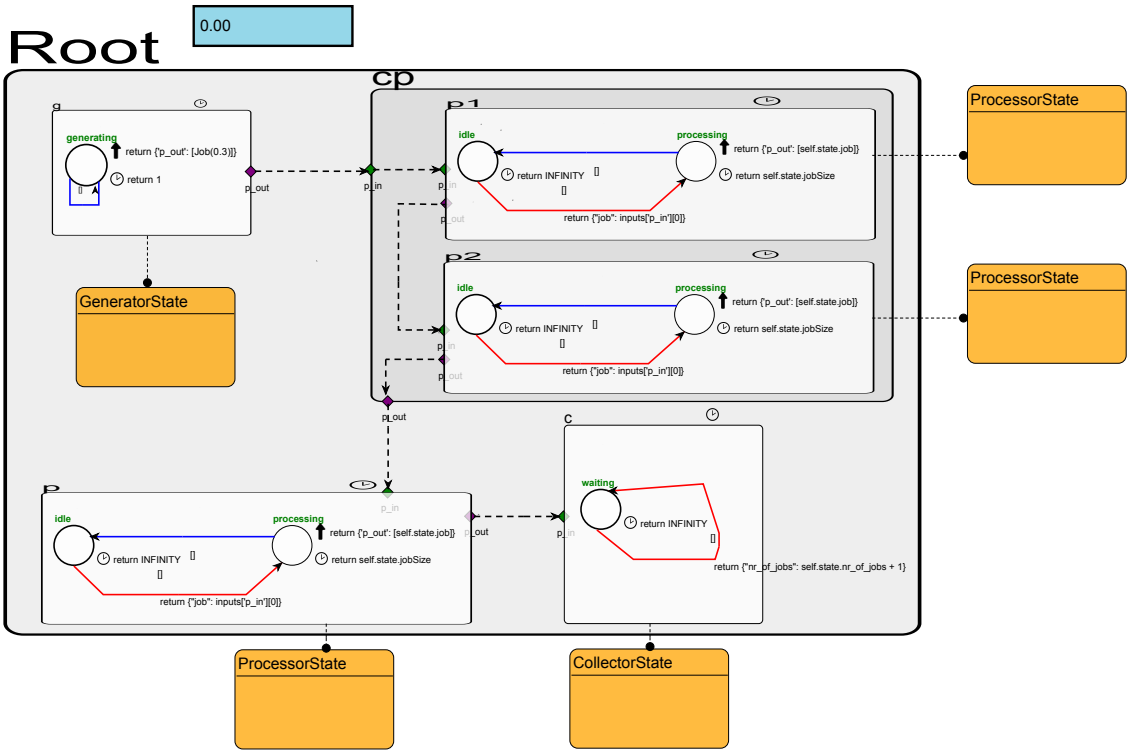


Figure 10. The runtime model of the producer-consumer DEVS model.

Figure 11. The debugging toolbar in AToMPM.

This boils down to augmenting the metamodel with the necessary runtime information, as well as expanding all references to atomic/coupled DEVS models found in the root model: they are replaced by their definitions. The expansion of models is implemented using an exogenous model transformation which transforms any valid design of a DEVS model into a runtime model. Figure 10 shows the resulting runtime model of the produce-consume example, of which the design is shown in Figure 9.

6.3 Debugging Interface

To support debugging, a toolbar was created in AToMPM, shown in Figure 11. By clicking on a button in this toolbar, a request is sent to the server, representing a particular command. The server will send the appropriate request to the simulator, which performs the requested action. As a result, a reply is sent back to the server, where it is translated to certain edit requests of the runtime model to visualize the changes for the user. The toolbar supports ten operations, explained in the following subsections.

6.3.1 Simulate Simulates the model as-fast-as-possible, until either the end condition evaluates to true, one of the breakpoints triggers, or the user manually pauses the simulation. During as-fast-as-possible simulation, state changes are not visualized, as this would significantly slow down the simulation due to the overhead of visualization requests from the simulator to AToMPM. Moreover, visually keeping track of state changes in as-fast-as-possible simulation is difficult, and most likely not what this option is used for. It should rather be used to quickly reach a breakpoint or the end of simulation.

6.3.2 Realtime Simulate Simulates the model in real-time. This means that the scheduler will try to meet all real-time deadlines, as specified in the time advance functions of the states in the atomic DEVS models. The values returned by these functions are interpreted as seconds. It is possible to pass a *scale factor* to real-time simulation. This floating point number specifies how much faster (if the value is larger than 1) or slower (if the value is smaller than 1) the simulation should be run. In other words, the scale factor specifies the linear relationship between the wall clock time and the simulated time: if it is 1, they are equal, if it is 2, simulated time advances twice as fast as wall clock time, etc. During real-time simulation, the model's visualization is updated, meaning that the user can follow the simulation process.

6.3.3 Pause Pressing the pause button will result in a running (as-fast-as-possible or real-time) simulation being paused as soon as possible. In as-fast-as-possible simulation, simulation is stopped after the currently executing big step is finished. In real-time simulation, it is additionally possible that the simulator is waiting until its next transition should be fired. The simulation is then paused in this waiting phase. When the simulation is paused, the current state of the model is visualized. Resuming is done by either stepping, as-fast-as-possible simulation, or realtime simulation. Additionally, the scale factor can be changed when starting a realtime simulation.

6.3.4 Big Step When the simulation is paused, the user can choose to continue the simulation by *stepping*. A big step computes output functions, the internal, external, and confluent transitions, and the time advance. The resulting state is visualized in the simulation environment.

6.3.5 Small Step A big step consists of six distinct phases. A small step allows to visualize these phases, called *small steps*. We list below for each phase what happens, and how it is visualized. The phases are shown in Figure 12.

1. Computing all imminent components: those DEVS models whose internal transitions are scheduled to fire. The imminent components are highlighted in blue.
2. Executing output functions for each imminent component, which results in events being generated on their output ports. To visualize this, an event is instantiated on the position of the output port.
3. Routing events from output ports to input ports, while executing their transfer functions. Visually, the event instance is moved to the position of the input port.
4. Deciding which models will execute their external (or, in the case it is an imminent component, confluent) transition function as a result of events on their input ports. DEVS models that will execute their external transition function are coloured red, those that will execute their confluent transition function are coloured purple.
5. Executing, in parallel, all enabled internal, external, and confluent transition functions. The current state is shown in green, and the values of its instance variables are displayed.
6. Computing, for each atomic DEVS model, the time at which its internal transition function is scheduled (which is specified in its *time advance* function). This value is displayed next to a clock icon on top of each atomic DEVS model.

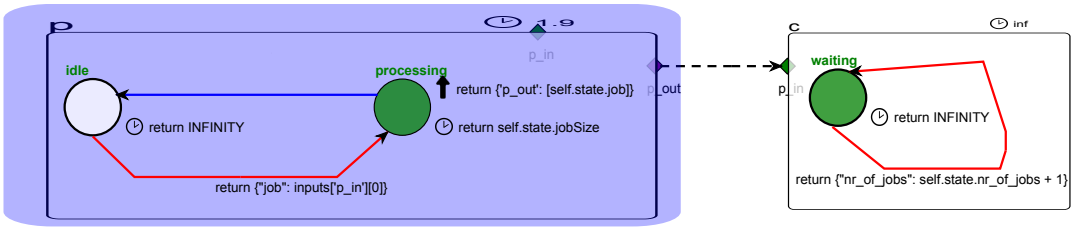
6.3.6 Step Back Undoes one "big step". The state as it was before the last big step is visualized.

6.3.7 Reset Resets the model and the simulation to their initial state. The initial state is visualized.

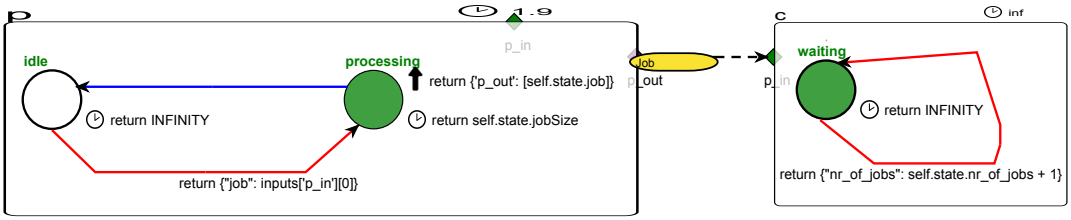
6.3.8 Show Trace During simulation, a detailed textual trace is generated. By clicking this button, the trace will be displayed in the browser's Javascript console, as shown in Figure 13.

6.3.9 Insert God Event A "god event" allows to manually change the value of a state variable. With this functionality, the environment allows to (visually) inspect and modify the internal state of DEVS models during a debugging session.

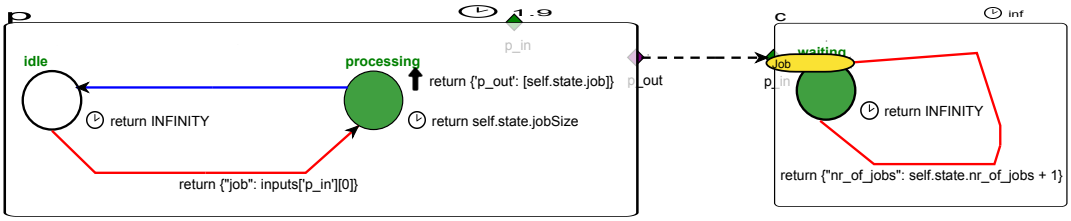
6.3.10 Inject Events Lastly, messages can be injected when the simulation is paused by instantiating the *EventInstance* class and connecting it to the port at which the event should be injected. Optionally, the user can specify the simulation time at which the event should be injected. By default, this is "now" (*i.e.*, the current simulation time). Events are not injected until this button is clicked. Once simulation is resumed,



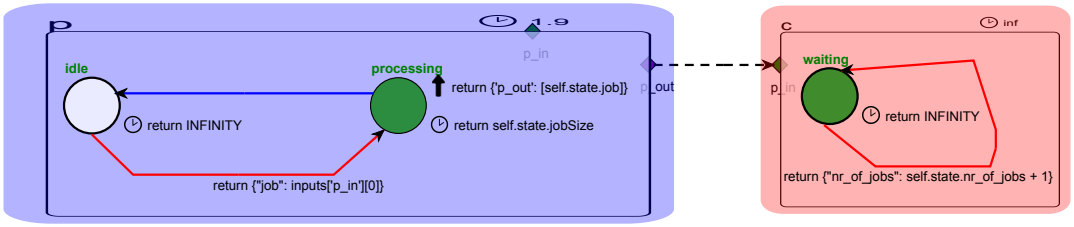
(a) Finding imminent components.



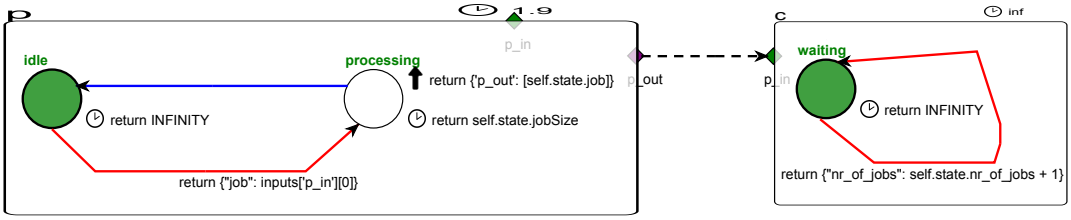
(b) Generate their output.



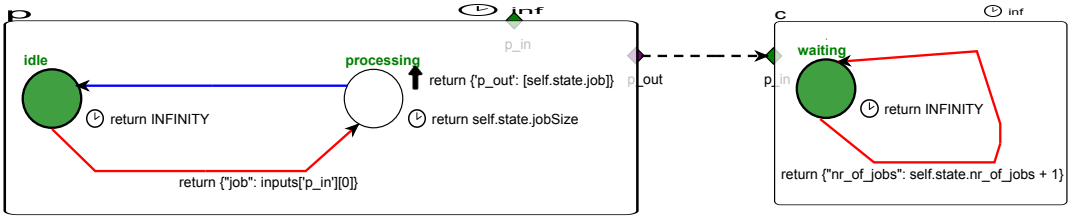
(c) Output routed from output to input port.



(d) Mark transition function to execute.



(e) Execute applicable transition function.



(f) Set time of next internal transition.

Figure 12. Sequence of small steps, forming a single big step.

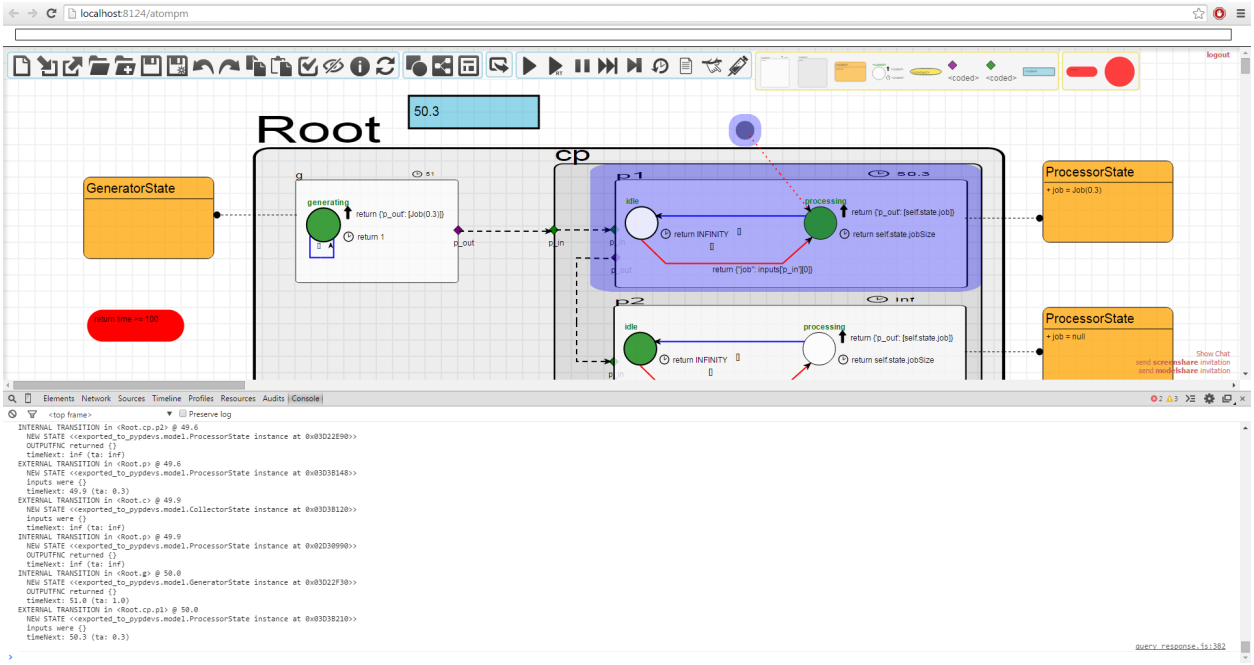


Figure 13. Viewing the trace in AToMPM after a breakpoint was triggered.

injected events are removed from the visual model. Once the time for their injection is reached, they will reappear on the appropriate port.

6.4 Breakpoints

We offer two ways of defining a breakpoint, which pauses the simulation automatically when the specified condition is fulfilled. A *global breakpoint* models such a condition, and the simulation breaks whenever the condition is satisfied. A *local breakpoint* also models a state condition, but is additionally connected to a state. The breakpoint will only trigger when both the condition is satisfied and the simulation just entered the state the breakpoint is connected to. This is comparable to breakpoints in code debugging that are associated to a specific line of code.

Figure 13 shows a paused simulation after a local breakpoint was triggered. The breakpoint that caused the pause is highlighted in blue. Afterwards, the breakpoint is greyed out, as it was disabled when it triggered. The user can configure whether the breakpoint should be automatically disabled after triggering.

7 Comparison with Other DEVS Tools

There are a number of commercial and research DEVS modelling and simulation tools that provide some form of debugging. In the next subsections, we look at a number of popular DEVS simulation environments. We discuss which debugging features they offer to the user, and compare them with our own tool.

7.1 ADEVS

ADEVS [33] is a DEVS simulator written in C++, and makes heavy use of templates. The tool focuses on performance and lightweightness, and does not offer a graphical user interface. In fact, it only offers simulation primitives to the user, such as executing a single simulation step (a *big step* in our terminology). While defaults are provided, access is offered to the lower levels. When making modifications, the user is fully responsible for the simulation algorithm, allowing the manual implementation of most debugging features. Primitive simulation operations are given, which can be used to implement small steps. ADEVS is therefore

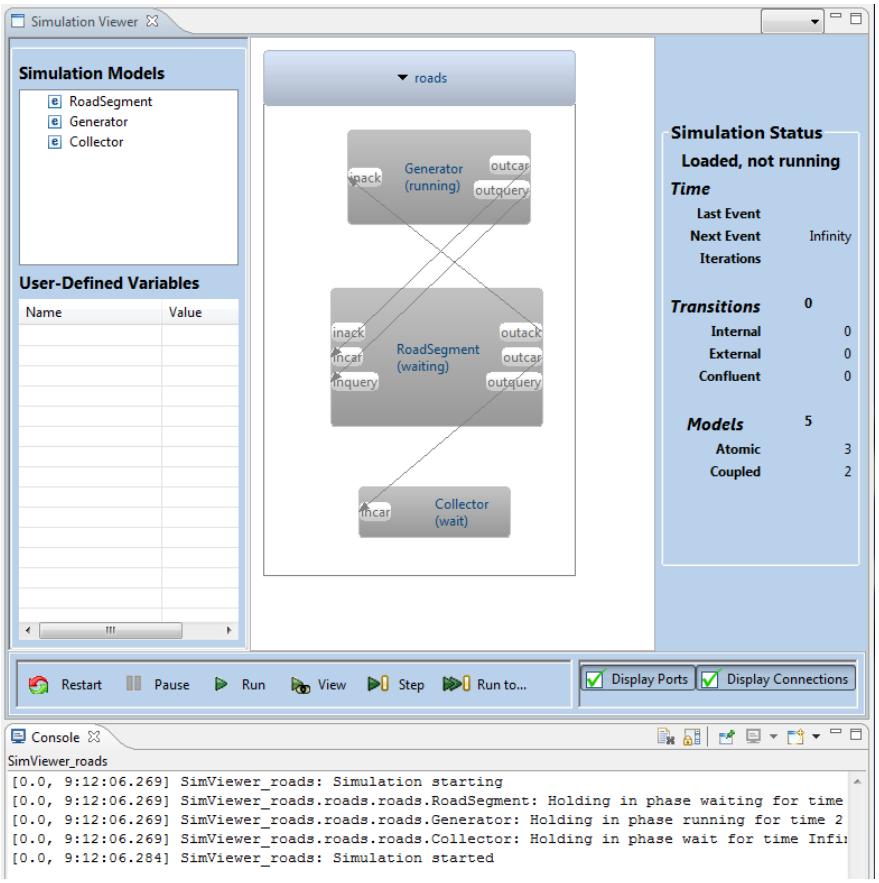


Figure 14. Debugging in MS4 Modelling Environment

versatile, but not very user-friendly, as simulating a DEVS model requires the modeller to write common simulation constructs manually. If advanced debugging operations are required, such as backward stepping, they need to be implemented each time they are needed.

7.2 MS4 Modelling Environment

The MS4 Modelling Environment [2] is an Eclipse-based environment for Parallel DEVS modelling and simulation. Models are created in a textual notation – *DEVS Natural Language* (DNL) for atomic models, and *System Entity Structure* (SES) for coupled models – after which they are compiled to Java files. While this offers an environment that is usable by non-programmers, it also introduces further complexity in terms of debugging. Tracability links need to be kept to the ultimately simulated model (in Java), and the model presented to the user (in DNL). Sadly, this is not always the case, making some syntactic errors more difficult to debug. Errors are sporadically reported at the level of the generated Java files, both in modelling and simulation, making debugging by non-programmers more difficult. MS4 ME contains a model visualizer, which can visualize the models (and their state) during simulation, as well as the exchanged events.

In Figure 14, the simulation of an example model is shown. This view allows for visual simulation of the model, and provides some debugging support, but it is limited to big stepping. Atomic DEVS models are visualized as boxes, showing the name of the model along with its current state. The operations supported by the environment, as well as their semantics, are as follows:

- **Restart** resets the simulation to the initial state.
- **Pause** pauses the simulation after the current step is finished.

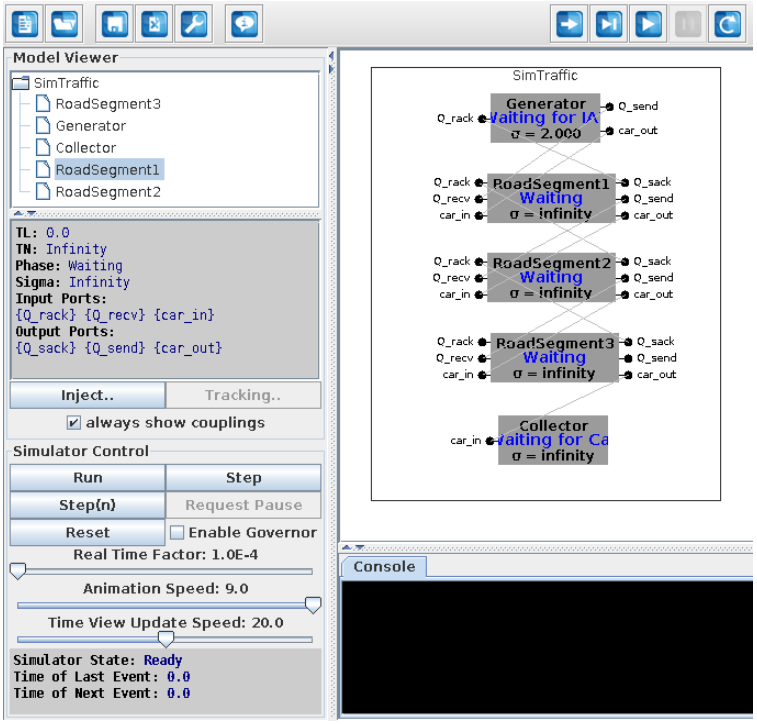


Figure 15. Debugging in DEVS Suite

- **Run** runs the simulation until all models passivate.
- **View** runs the simulation until the end, but visualizes messages sent between atomic DEVS models. Each transition takes a fixed amount of time (around one second), and the messages "travel" visually from the output port to the input port.
- **Step** allows to perform one simulation step when the simulation is paused. A step here is a "big step" (*i.e.*, it advances the simulation from one consistent state to the next).
- **Run to...** allows to specify a fixed number of iterations or a simulation time at which to stop the simulation.
- Inputs can be **inject**ed by right-clicking an atomic DEVS model and choosing the appropriate option.

The end of the simulation can only be specified in number of simulation steps. More complex termination conditions, such as terminating at a specific simulation time, or based on a condition, are not supported. Since the programming interface exposed to the user is minimal, it is also not possible to implement this manually.

The environment displays some simulation runtime variables too:

- **Simulation time** shows the current simulation time;
- **Time of last event** shows the simulation time at which the last event was processed;
- **Time of next event** shows the simulation time at which the next internal event is scheduled;
- **Iterations** shows the number of iterations already executed;
- **Transitions** shows, for each type of transition, the amount which were executed.

7.3 DEVS Suite

DEVS Suite [3] offers a visual debugging environment for the interactive simulation of DEVS models. A simulation of an example model is shown in Figure 15. As was the case in the MS4 Modelling Environment, the atomic DEVS models are shown as rectangles, displaying the name of the model, as well as its current

state and its in- and output ports. Moreover, the time until the next internal transition is also shown. The operations supported by the environment are listed below:

- **Inject...** allows to inject a message at a specific input port.
- **Tracking...** allows to set tracking options. These options include the visualization and logging of state changes and messages sent/received.
- **Run** runs the simulation until completion.
- **Request Pause** pauses the simulation after the currently executing step.
- **Step** performs one big step.
- **Step(n)** performs n big steps.
- **Reset** resets the simulation to the initial state.
- A slider allows to set the **Real Time Factor**, which is used to slow down or speed up the real time simulation.
- A second slider allows to set the **Animation Speed**.

As can be seen from Figure 15, messages are visualized on the output ports when they are generated, and transition to the input ports when they are sent. DEVS Suite is very similar to MS4 ME in terms of functionality, with the exception that models need to be specified completely in Java.

7.4 VLE: The Virtual Laboratory Environment

VLE [34] supports the modelling and simulation of DEVS models. Atomic DEVS models are written in C++ code, but coupled DEVS models can be created both textually (in XML) or visually (using GVLE). Simulation of the created models is supported, though there is no support for any form of simulation debugging; the interface only presents a progress bar with a start and stop button. It is possible to visualize the coupled model at design-time, though it only shows the coupling of the models. Behaviour of atomic models still needs to be specified completely in code, which gets dynamically loaded. Some simulation debugging options can be set, such as tracing, but overall support is limited and mostly not graphical.

7.5 X-S-Y

X-S-Y [35] is a DEVS simulator written in Python. It provides a set of classes from which users can inherit to build their DEVS models. There is no visual user interface, but a command line interface is provided, as seen in Listing 1. It supports the following operations:

- **Exit** the current simulation.
- **Inject** an event on a port. A special callback function (provided by the user) is in charge of translating the command line input to an event.
- Change the **mode** of the simulation by setting simulation parameters, listed below:
  - **StepByStep** - simulate by stopping after each increment of one (simulated time) second.
  - **Scale Factor** - change how fast the simulation runs (in scaled real-time).
  - **Display Time Mode** - change how much detail of the time advance functions is visible.
  - **Verbose Mode** - show or hide output.
  - **Simulation Time Bound** - set the point in time at which simulation should stop.
  - **Autoreset** - change whether or not the simulation should reset after the time bound is reached.
  - **Number of Simulation Runs**.
- **Pause** the simulation.
- **Reset** the simulation.
- Get **statistics** of the current simulation.

Listing 1: X-S-Y command-line interface

```
-- the current state--
(PingPong:C={p1:tL=0.000,tN=inf,p2:tL=0.000,tN=1.000},tL=0.000,tN=1.000) at time 0.000.
```

	ADEVS	DEVS Suite	MS4 Me	VLE	X-S-Y	PyPDEVS
Pause	C	Y	Y	N	Y	Y
(Scaled) Realtime	C	Y	Y	N	Y	Y
Big Step	Y	Y	Y	N	Y	Y
Small Step	C	N	N	N	N	Y
Termination condition	Y	N	N	N	N	Y
Breakpoints	N	N	N	N	N	Y
Event Injection	C	Y	Y	N	Y	Y
State Changes	N	N	N	N	N	Y
State Visualisation	C	C	C	N	C	Y
Event Visualisation	N	Y	Y	N	N	Y
Tracing	C	Y	Y	Y	Y	Y
Model Visualisation	N	Y	Y	Y	N	Y
Reset	N	Y	Y	C	Y	Y
Step back	N	N	N	N	N	Y

Table 1. A comparison of the different tools discussed in this section.

```
-- simulation menu --
e(x)it, (in)jet, (m)ode, (p)ause, reset, run, stat(i)stics> i
================================================================
statistics of the current trajectory
P(C={p1:tL=2.000,tN=inf,p2:tL=2.000,tN=3.000})=0.167,P(C={p1:tL=5.000,tN=6.000,p2:tL=5.000,
    tN=inf})=0.167,P(C={p1:tL=3.000,tN=4.000,p2:tL=3.000,tN=inf})=0.167,P(C={p1:tL=1.000,tN
    =2.000,p2:tL=1.000,tN=inf})=0.167,P(C={p1:tL=4.000,tN=inf,p2:tL=4.000,tN=5.000})=0.167,P
    (C={p1:tL=0.000,tN=inf,p2:tL=0.000,tN=1.000})=0.167,
================================================================
```

As was the case in most other tools, a big step is the atomic operation in the simulation. State visualization is only partially supported: the current state is shown textually due to the nature of the tool. Note that control is never passed to the user again after simulation has started, making it impossible to inject events or query the model.

7.6 Comparison

In Table 1, the functionality of the five tools are compared to PythonPDEVS. For each function, we list whether or not the tool implements it (**Y** for yes, **N** for no), or if there is a comment (**C**). The comments are explained below.

For ADEVS, most debugging features can manually be implemented if so desired. This is because ADEVS is more a simulation library than a simulation tool. Entries marked as "not available" are therefore still implementable, though they require significant code additions.

Most tools are able to present the state textually, though not graphically visualized as is the case in the PythonPDEVS user interface. In PythonPDEVS, the sequential state of the model is shown in the form of a state machine, thereby giving more detailed insight in past and future behaviour.

VLE does provide some capabilities for resetting the model, though this is not automated: users need to do so manually. While only PythonPDEVS supports breakpoints natively, breakpoints can also be implemented using a termination condition which just checks all breakpoint conditions. As such, it could be argued that ADEVS can support the use of breakpoints. This is not as powerful as native support though. With native support, it is possible to directly assess which condition has triggered the termination. Also, it is often desired to disable or enable breakpoints throughout the simulation. For example, a breakpoint that triggers

if the simulation time becomes larger than a certain value, should be disabled after it was triggered once. Additionally, native breakpoints can be more tightly integrated in the simulation kernel, resulting in more relevant information. For example, PythonPDEVS allows the definition of a breakpoint in case a certain state is entered. This would not be trivial in the other tools, as the simulators do not offer information on which models had a state change. Adding this support requires tedious manual bookkeeping of all models and their history.

Three features are exclusively supported out of the box by PythonPDEVS: breakpoints, manual state manipulation—so called *god events*—and stepping back (reversible debugging). Neither of these are trivial to implement manually, and have a significant impact on the simulation kernel and its performance. Additionally, PythonPDEVS is the only simulation kernel that is not implemented exclusively in code: its control flow is explicitly modelled using Statecharts. This allows to almost trivially add further features to the simulation core. Most low-level operations also become independent of the used platform, which is not the case with the other simulation kernels. Others use the underlying platform (*e.g.*, threads and sleeps) directly, though cannot be coupled to different platforms (*e.g.*, GUI event loops).

8 Conclusion

In this paper, we show how to create an advanced debugging and experimentation environment for Parallel DEVS models. The environment provides the user with a level of control unmatched by any of the state-of-the art tools. The supported operations are similar to those of traditional code debugging tools, though some operations specific to DEVS were added.

Simulation can be paused, resumed, and stepped through. Breakpoints are modelled in the simulated model as an expression. The environment allows users to switch between execution modes, to alter the speed of the simulation, inject events, and modify the state of the system.

To tackle the complexity of building this debugger, the modal part of the simulator is explicitly modelled as a Statecharts model. Statecharts are the most natural choice as a formalism for modelling real-time, autonomous and reactive system. The non-modal parts of the simulator remains coded.

The debugger is combined with a visual environment to allow visual design and debugging of Parallel DEVS models: a toolbar is provided to control the simulation process by sending appropriate messages to the debugger. The runtime model is visually updated during simulation. An architecture supporting our approach was presented, showing the distinction between the visual front-end, and the simulation kernel back-end. This architecture is expected to be general enough to support debugging of models in multiple formalisms (and combinations thereof).

We compared our debugging features to those offered by several popular Parallel DEVS simulation environments. Apart from being explicitly modelled, our approach adds features that are not found in any other tool, such as god events and stepping back in time.

In the future, we want to extend this approach to other formalisms, such as Petrinets [36] and rule-based model transformations [37], which both introduce non-determinism.

Acknowledgements

This work was partly funded by the Automotive Partnership Canada (APC) in the NECSIS project and with PhD fellowship grants from the Agency for Innovation by Science and Technology in Flanders (IWT) and the Research Foundation - Flanders (FWO).

References

[1] Maryam M. Maleki, Robert F. Woodbury, Rhys Goldstein, Simon Breslav, and Azam Khan. Designing DEVS visual interfaces for end-user programmers. *Simulation*, 91(8):715–734, 2015.

[2] Chungman Seo, Bernard P. Zeigler, Robert Coop, and Doohwan Kim. DEVS modeling and simulation methodology with MS4Me software. In *Symposium on Theory of Modeling and Simulation - DEVS (TMS/DEVS)*, 2013.

[3] Sungung Kim, Hessam S. Sarjoughian, and Vignesh Elamvazhuthi. DEVS-Suite: a simulator supporting visual experimentation design and behavior monitoring. In *Proceedings of the Spring Simulation Conference*, 2009.

[4] Sadaf Mustafiz and Hans Vangheluwe. Explicit modelling of Statechart simulation environments. In *Summer Simulation Multiconference*, pages 445 – 452. Society for Computer Simulation International (SCS), July 2013. Toronto, Canada.

[5] Boris Beizer. *Software Testing Techniques (2Nd Ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[6] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[7] Hans Vangheluwe. Foundations of modelling and simulation of complex systems. *ECEASST*, 10, 2008.

[8] P. J. Mosterman and Hans Vangheluwe. Computer Automated Multi-Paradigm Modeling: An Introduction. *Simulation*, 80(9):433–450, September 2004.

[9] Yentl Van Tendeloo and Hans Vangheluwe. The modular architecture of the Python(P)DEVS simulation kernel. In *Proceedings of the 2014 Symposium on Theory of Modeling and Simulation - DEVS*, pages 387–392, 2014.

[10] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[11] Simon Van Mierlo. Explicitly modelling model debugging environments. In *Proceedings of the ACM Student Research Competition at MODELS 2015 co-located with the ACM/IEEE 18th International Conference MODELS 2015*, pages 24–29, 2015.

[12] Simon Van Mierlo, Yentl Van Tendeloo, Bruno Barroca, Sadaf Mustafiz, and Hans Vangheluwe. Explicit modelling of a Parallel DEVS experimentation environment. In *Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, DEVS '15, pages 107–114, San Diego, CA, USA, 2015. Society for Computer Simulation International.

[13] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Huseyin Ergin. AToMPM: A web-based modeling environment. In *MODELS'13 Demonstrations*, 2013.

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

[15] Hans Vangheluwe. DEVS as a common denominator for multi-formalism hybrid systems modelling. *CACSD. Conference Proceedings. IEEE International Symposium on Computer-Aided Control System Design*, pages 129–134, 2000.

[16] Alex Chung Hen Chow and Bernard P. Zeigler. Parallel DEVS: a parallel, hierarchical, modular, modeling formalism. In *Proceedings of the 26th Winter Simulation Conference*, pages 716–722, 1994.

[17] Alex Chung Hen Chow, Bernard P. Zeigler, and Doo Hwan Kim. Abstract simulator for the parallel DEVS formalism. In *AI, Simulation, and Planning in High Autonomy Systems*, pages 157–163, 1994.

[18] Bil Lewis. Debugging backwards in time. *arXiv preprint cs/0310016*, (September):225–235, 2003.

[19] Nicholas A. Allen, Clifford A. Shaffer, and Layne T. Watson. Building modeling tools that support verification, validation, and testing for the domain expert. In *Proceedings of the 37th Winter Simulation Conference*, WSC '05, pages 419–426. Winter Simulation Conference, 2005.

[20] Adrian Pop, Martin Sjölund, Adeel Asghar, Peter Fritzson, and Casella Francesco. Static and Dynamic Debugging of Modelica Models. In *Proceedings of the 9th International Modelica Conference*, pages 443–454, November 2012.

[21] Jonathan Corley. Debugging for model transformations. In *Proceedings of the MODELS 2013 Doctoral Symposium co-located with the 16th International ACM/IEEE Conference on Model Driven Engineering Languages and Systems (MODELS 2013), Miami, USA, October 1, 2013.*, pages 17–24, 2013.

[22] A. Krasnogolowy, S. Hildebrandt, and S. Wätzoldt. Flexible debugging of behavior models. In *Proceedings of the 2012 IEEE International Conference on Industrial Technology (ICIT)*, pages 331–336, March 2012.

[23] Tanja Mayerhofer. Testing and debugging UML models based on fUML. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1579–1582, Piscataway, NJ, USA, 2012. IEEE Press.

[24] Yoann Laurent, Reda Bendraou, and Marie-Pierre Gervais. Executing and debugging UML models: An fUML extension. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1095–1102, New York, NY, USA, 2013. ACM.

[25] Raphael Mannadiar and Hans Vangheluwe. Debugging in domain-specific modelling. In Brian Malloy, Steffen Staab, and Mark Brand, editors, *Software Language Engineering*, volume 6563 of *Lecture Notes in Computer Science*, pages 276–285. Springer Berlin Heidelberg, 2011.

[26] Andrei Chiş, Marcus Denker, Tudor Gîrba, and Oscar Nierstrasz. Practical domain-specific debuggers using the moldable debugger framework. *Computer Languages, Systems & Structures*, 44(PA):89–113, December 2015.

[27] Erwan Bousse, Jonathan Corley, Benoit Combemale, Jeff Gray, and Benoit Baudry. Supporting efficient and advanced omniscient debugging for xdsmls. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2015, pages 137–148, New York, NY, USA, 2015. ACM.

[28] P. Kemper. A trace-based visual inspection technique to detect errors in simulation models. In *2007 Winter Simulation Conference*, pages 747–755, Dec 2007.

[29] Richard M. Fujimoto. *Parallel and Distribution Simulation Systems.* John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1999.

[30] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, 1985.

[31] Christopher D. Carothers, Kalyan S. Perumalla, and Richard M. Fujimoto. Efficient optimistic parallel simulations using reverse computation. *ACM Transactions on Modeling and Computer Simulation*, 9(3):224–253, 1999.

[32] Robert Rönngren, Michael Liljenstam, Rassul Ayani, and Johan Montagnat. Transparent incremental state saving in time warp parallel discrete event simulation. *SIGSIM Simulation Digest*, 26(1):70–77, July 1996.

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

[34] Gauthier Quesnel, Raphaël Duboz, Éric Ramat, and Mamadou K. Traoré. VLE: a multimodeling and simulation environment. In *Proceedings of the 2007 summer computer simulation conference*, pages 367–374, 2007.

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

[36] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[37] Eugene Syriani and Hans Vangheluwe. A modular timed graph transformation language for simulation-based design. *Software and Systems Modeling (SoSyM)*, 12(2):387–414, 2013.

Author Biographies

Simon Van Mierlo is a PhD student at the University of Antwerp, Department of Mathematics and Computer Science, Antwerp, Belgium.

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

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