Explicit Modelling of a Parallel DEVS Experimentation Environment

Simon Van Mierlo †
simon.vanmierlo@uantwerpen.be

Yentl Van Tendeloo †
yentl.vantendeloo@uantwerpen.be

Bruno Barroca ‡
bbarroca@cs.mcgill.ca

Sadaf Mustafiz ‡
sadaf@cs.mcgill.ca

Hans Vangheluwe †‡
hv@cs.mcgill.ca

† University of Antwerp, Belgium
‡ McGill University, Montréal, Canada

ABSTRACT

In this paper, we explicitly model an interactive debugging and experimentation environment for the simulation of Parallel DEVS models. We take inspiration from the code debugging world, as well as from the simulation world (including different notions of time) to model our environment. We support both as-fast-as-possible and (scaled) real-time execution of the model. To achieve this, the PythonPDEVS simulator is de/re-constructed: the modal part of the simulator, 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 debugger for Parallel DEVS, which is used to regenerate the code of the simulator. It is then combined with a modelling and simulation environment to visually model, simulate, and debug Parallel DEVS models.

Author Keywords
Parallel DEVS; Debugging; Statecharts

ACM Classification Keywords
I.6.7 SIMULATION AND MODELING: Simulation Support Systems; D.2.5 SOFTWARE ENGINEERING: D.2.5 Testing and Debugging

INTRODUCTION

The systems we analyse, design, and develop today are characterized by an ever growing complexity. Modelling and Simulation (*M&S*) [20] become an increasingly important enabler 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 (software-intensive) 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) [7]. The M&S approach can only be successful if there is sufficient tool support, *i.e.*, if modellers have access to tools which effectively support each phase in the M&S process. This is no different from traditional, code-based software development methods: programmers have access to various helpful tools such as version control software, testing tools, and debuggers. Debuggers allow developers to locate the source of a defect (which was detected by a failing test, meaning that one of its properties was not satisfied) using breakpoints, stepping, and tracing of runtime variables [23].

Support for simulation debugging, however, is currently limited. This is mainly due to its inherent complexity: the interplay of formalism execution semantics, different notions of simulated time such as (scaled) real-time and as-fast-as-possible execution semantics, as well as user interaction through an interface are all challenging to capture and correctly implement using traditional software development techniques. Debugging is not only useful for program code, however, and is a necessary condition for lifting the M&S approach to a true engineering discipline with widespread acceptance.

DEVS, as introduced by Zeigler [22], 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 [19]. A popular extension (and the default in many simulation tools) of Classic DEVS is Parallel DEVS [2], 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.

In this paper, we describe the construction of a visual modelling, simulation, and debugging environment for Parallel DEVS. To achieve this, we de-/re-construct the PythonPDEVS simulator [18] as described in [21] and add debugging support to the modal part (which is explicitly modelled in the Statecharts [4] formalism) of the simulator. We combine the simulator with the visual modelling tool AToMPM [15], allowing for the visual interactive control of DEVS model simulations. Full details of this work can be found in a technical report [17].

The rest of the paper is structured as follows. Section BACKGROUND provides a short explanation of the Parallel DEVS and Statecharts formalisms. Section DEBUGGING DEVS MODELS presents a set of useful operations for the debugging of Parallel DEVS models. Section METHOD discusses the method of de-/re-constructing a simulator, applied to the PythonPDEVS simulator. Section USER INTERFACE explains the visual user interface of the debugger. Section RELATED WORK discusses related work and compares our environment to existing DEVS modelling and simulation environments. Section CONCLUSION AND FUTURE WORK concludes.

BACKGROUND

In this section, we introduce the reader to the Parallel DEVS formalism, as it is a prerequisite to understand the debugging operations introduced later on. We also give a short introduction to the Statecharts formalism, as it will be used to model our debugger explicitly.

DEVS, and in particular Parallel DEVS, is used to model the behaviour of discrete event systems. Its basic building blocks are *atomic DEVS* models, which are structures

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

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* $\delta_{int} : S \rightarrow S$ defines the next sequential state, depending on the current state. The *output function* $\lambda : S \rightarrow Y^b$ maps the sequential state set onto an output bag. The *external transition function* $\delta_{ext} : Q \times X^b \rightarrow S$ 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* $ta : S \rightarrow \mathbb{R}_{0,+\infty}^{+}$ defines the simulation time the system remains in the current state before triggering its *internal transition function*. The *confluent transition function* $\delta_{conf} : S \times X^b \rightarrow S$ 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", defining a *transfer function* to translate output to input messages. Parallel DEVS is closed under coupling, which means that coupled models can be nested to arbitrary depth.

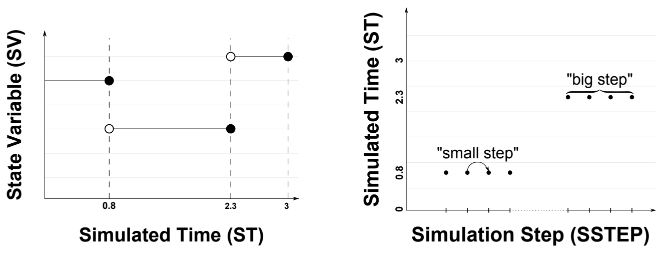
An abstract simulator for Parallel DEVS is described in [3]. Such a simulator computes the next state of the system (a "step") until its end condition is satisfied. Each step consists of the following phases:

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. Determine the type of transition to execute for the atomic DEVS model, depending on it being imminent and/or receiving input.



(a) Change of state over simulated time.

(b) Multiple steps execute a single simulated time.

Figure 1. Simulation time and steps.

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).

The Statecharts formalism was introduced by David Harel [4]. Statecharts is an extension of state machines and state diagrams with hierarchy, orthogonality, and broadcast communication. It is 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 and/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.

In the remainder of the paper we describe how the timed, reactive, interruptible behaviour of the PythonPDEVS simulator is modelled as a Statechart. This model is extended with debugging operations to allow interactive control of the simulation.

DEBUGGING DEVS MODELS

In this section, a useful set of operations for the debugging of Parallel DEVS models is constructed. We take inspiration from traditional code debugging (state manipulation and breakpoints) as well as simulation-specific concepts (manipulation of simulated time).

Time

The notion of *time* plays a prominent role in model simulation. Simulated time differs from the wall-clock time: it is the internal clock of the simulator. In general, a simulator updates some state variable vector, which keeps track of the current simulation state, each time increment. This is depicted in Figure 1a. The state is updated by some computations, or

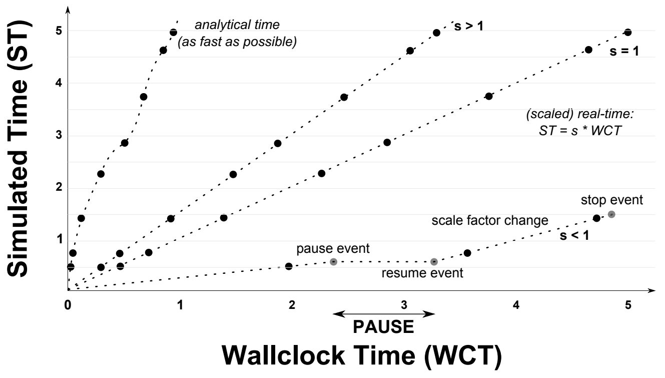


Figure 2. Different notions of time.

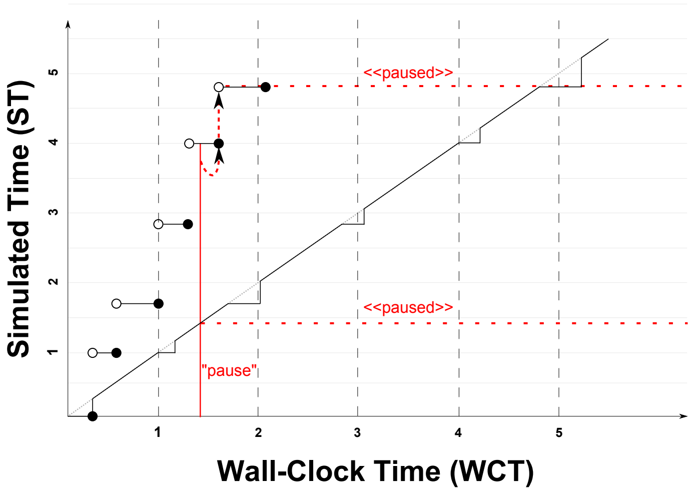


Figure 3. Pausing simulation: difference between as-fast-as-possible and real-time simulation.

"steps": a big step corresponds to the computation of the next value of the state variable, and consists of a number of small steps. This is shown in Figure 1b. In the context of DEVS, each phase in computing the next state (as presented in Section BACKGROUND) is regarded as a small step.

Executing program code is always done as fast as possible, *i.e.*, the speed of the program is limited by the machine executing it. Simulations, however, can be run as-fast-as-possible, or in (scaled) real-time, which is useful for simulating models of real-time systems which might be deployed as such on a real-time device. In this case, there is a linear relation between the wall-clock time and the simulated time. The relation of the different notions of simulated time and the wall-clock time is depicted in Figure 2. Note that there is no linear relation in as-fast-as-possible simulation, meaning that the "current simulated time" is simply a variable in the simulator. Moreover, operations can be performed on simulated time, such as pausing, or stepping back, which are naturally not allowed on wall-clock time.

Pausing is a useful debugging operation, as it allows to interrupt a running program or simulation and inspect the current state of the system. We already mentioned that it is a valid operation on simulated time, but we do have to make a distinction between different simulation modes (as-fast-as-possible and (scaled) real-time) to determine the semantics of a pause. Figure 3 depicts the difference between the two modes. In as-fast-as-possible mode (visualized by the stepwise function), the simulated time is incremented as quickly as the executing system allows. The horizontal parts represent computation time necessary to compute the state after the next "big step". In as-fast-as-possible mode, these computation parts are executed one after the other, without any waiting in between. This means that if a pause is requested (denoted by the red vertical bar with "pause" next to it), the simulation will only be paused after completion of that big step computation. Halting immediately might otherwise leave the system in a (macroscopically) inconsistent state.

In real-time mode, simulated time is "synchronized" with the wall-clock time. The time needed to compute the next state is still there, but now the simulator will wait in between these computation periods 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, thus desynchronizing the simulated time from the wall-clock time. This is represented by the horizontal parts in the function. When the computation is finished, the simulator will synchronize both times immediately, as depicted by the vertical parts. If a pause is requested during a waiting period, the simulator immediately pauses. The result is that the system will be in the "current" state, and not the "next", as was the case for as-fast-as-possible mode. The simulated time will be in between the previous transition time and the next.

Due to this difference, an as-fast-as-possible simulation cannot pause at times in between two different simulated times. On the other hand, real-time simulation can pause at virtually every point in simulated time. The notable exception being the time at which transition functions are being computed.

State Manipulation

Debuggers allow to modify the state of the system. What constitutes the "state" of the system is formalism-dependent. In the case of program code, the state consists of the values of the runtime variables and the program counter. In the case of a Parallel DEVS system, the state consists of the state of each of its atomic models. In PythonPDEVS, each such a state has a number of attribute values. Modifying the state during simulation is achieved by changing these values, known as a *god event*, as some "force" outside of the simulator manually changes the state.

During simulation, atomic DEVS models generate events on their output ports and accept events on their input ports. This generating, routing, and accepting of events are phases in computing the next step and is done by the simulator in the "computation phase", as introduced in Figure 1. They indirectly influence the state, as they trigger the external transition function in atomic DEVS models. A related useful debugging operation is to manually *inject* events.

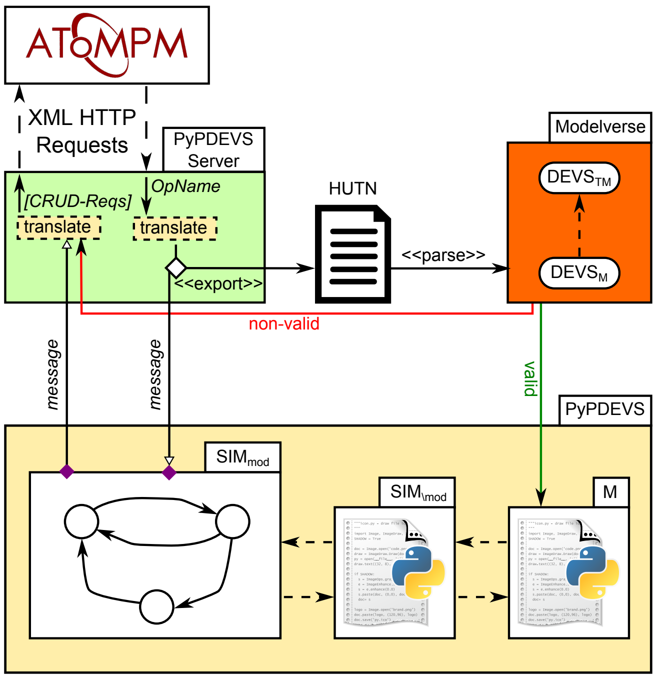


Figure 4. Architecture.

Breakpoints

Breakpoints allow to set a specific point during simulation at which the simulator should pause. They are useful in locating possible sources of defects. In program debugging, a breakpoint can be defined on a specific *line* of the program code, and optionally define a constraint on the system state. In the case of Parallel DEVS, we distinguish two different types of breakpoints: global breakpoints, that break on a "global condition", such as the simulated time or model state, and "local breakpoints", that break when a specific state of an atomic DEVS model is entered, and an optional constraint is satisfied.

METHOD

In this section our solution architecture is explained in detail. We start by giving an overview of the architecture, then explain how the PythonPDEVS simulator was de/reconstructed to enhance it with debugging support, and finally we explain how models are validated by loading them into the Modelverse, a model repository and framework [16].

Architecture

The architecture of the DEVS debugger is shown in Figure 4. There are four main components to our solution architecture:

1. **AToMPM** [15] is a visual modelling tool. It is used to model Parallel DEVS models in a visual language, and as a front-end for the PythonPDEVS simulator and debugger.
2. The **Modelverse** [16] is a model repository and metamodelling framework. A Human-Usable Textual Notation (HUTN) allows to quickly develop a model and store it in the Modelverse, where it can be checked for conformance and consistency.
3. **PythonPDEVS** [18] is a DEVS simulator, which has been de/reconstructed, as described in the next subsection. The simulator simulates the exported model, and can receive messages to control its execution. It generates output messages that reflect the current state in the simulation.
4. Finally, the **PythonPDEVS Server** is an extension of the AToMPM model transformation server and acts as the "glue" between the visual front-end in AToMPM, the Modelverse, and the PythonPDEVS simulator.

De/Reconstructing the Simulator

De/reconstructing the simulator consists of extracting the modal behaviour of the simulator, and modelling it as a Statechart. Then, this Statechart is extended with debugging-specific operations such as pausing, breakpoints, and state manipulation. This process is explained in detail for a debugger for Causal Block Diagrams in [21] and will not be repeated here. The Statechart resulting from the de/reconstruction process describing the behaviour of the PythonPDEVS simulator, extended with debugging operations, is shown in Figure 5. It consists of a number of orthogonal states:

- **breakpoint** allows the management of breakpoints;
- **trace** allows the configuration of textual tracing;
- **inject** allows to inject events on ports;
- **reset** allows to reset the model to its initial state;
- **simulation_state** keeps track of the execution mode the simulation is in: *(scaled) real-time*, *continuous* (as-fast-as-possible), *big step*, or *pause*;
- **simulation_flow** performs the computation and consists of two states. The *check_termination* state checks when simulation should halt, either due to the termination condition or a breakpoint. Simulation automatically pauses after a big or small step has finished. When simulation is paused, *god event*s are accepted and processed. *do_simulation* implements the simulation algorithm as listed in Section BACKGROUND. In *continuous*, *realtime*, or *big step* mode, these states are just traversed. In *paused* mode, a *small step* can manually trigger a single transition.

A user interface can send messages to this Statechart to control the simulation of a DEVS system and receives messages from that same Statechart to display the state of the system to the user. A command-line reference implementation was created to demonstrate the usage of the DEVS simulator and debugger, and in the next section, we will explain how a visual, more user-friendly modelling and debugging environment was constructed in AToMPM.

HUTN and Modelverse

A neutral action language is used to specify the internal behaviour (*i.e.*, conditions and actions) of the functions in the AToMPM DEVS models. In order to both validate and verify the designed models, we built an exporter to the Modelverse that translates DEVS models into a textual notation called DEVSLang. This notation was designed in order to be

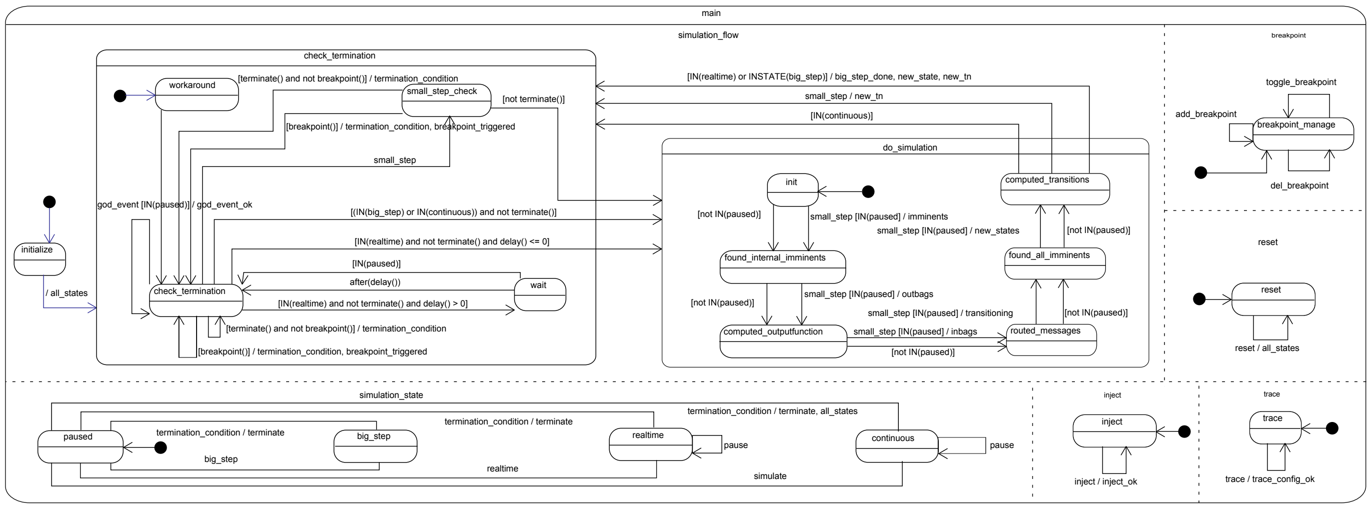

main
simulation_flow
breakpoint
check_termination
workaround
[terminate() and not breakpoint()] / termination_condition
small_step_check
[not terminate()]
[breakpoint()] / termination_condition, breakpoint_triggered
small_step
[IN(realtime) or INSTATE(big_step)] / big_step_done, new_state, new_tn
small_step / new_tn
[IN(continuous)]
do_simulation
init
computed_transitions
[not IN(paused)]
small_step [IN(paused)] / imminents
small_step [IN(paused)] / new_states
[not IN(paused)]
found_internal_imminents
found_all_imminents
small_step [IN(paused)] / outbags
[not IN(paused)]
small_step [IN(paused)] / transitioning
[not IN(paused)]
computed_outputfunction
small_step [IN(paused)] / inbags
routed_messages
[not IN(paused)]
god_event [IN(paused)] / god_event_ok
[(IN(big_step) or IN(continuous)) and not terminate()]
[IN(realtime) and not terminate() and delay() <= 0]
[IN(paused)]
after(delay())
wait
initialize
/ all_states
check_termination
[IN(realtime) and not terminate() and delay() > 0]
[terminate() and not breakpoint()] / termination_condition
[breakpoint()] / termination_condition, breakpoint_triggered
toggle_breakpoint
add_breakpoint
breakpoint_manage
del_breakpoint
reset
reset
reset / all_states
simulation_state
termination_condition / terminate, all_states
termination_condition / terminate
termination_condition / terminate
paused
big_step
realtime
pause
continuous
pause
big_step
realtime
simulate
inject
inject
inject / inject_ok
trace
trace
trace / trace_config_ok

Figure 5. The modal part of the PythonPDEVS simulator.

```
atomic Processor():
   inports p_in
   outports p_out
   mode ProcessorState('idle', job):
      any -> ProcessorState('processing', p_in[0])
      after infinity -> any
      out nothing
   mode ProcessorState('processing', job):
      after job.jobSize -> ProcessorState('idle')
      out {p_out: [job]}
    initial Processor('idle')
```

Listing 1. Processor example in HUTN.

intuitive, expressive, portable, and above all verifiable. The reader is referred to [17] for more details on the textual concrete syntax. A snippet of a DEVSLang model (translated from the model in Figure 6), is shown in Listing 1. *States* in the AToMPM model are mapped to *modes* in DEVSLang.

The exported models are instrumented (with visualization details) to allow feedback from the HUTN checker back to the AToMPM source model. Based on the error and warning feedback, the user needs to make the necessary modifications to the action code in order to continue with the simulation. For the moment, the HUTN checker implements the following checks:

- The action code is checked for basic syntantic errors according to the DEVSLang grammar definition.

- Invalid references in the DEVS models are checked against their respective definitions (errors are issued on reference resolution failures). For example, a variable can only be accessed if it is defined within the scope of that component. Also, during the evaluation of the transition preconditions, we do not allow write access on the components' state variables.

Additionally, we plan on defining a conservative analysis procedure in order to automatically decide if a given DEVSLang model is at all translatable to timed automata, so that further behavioural analysis can be provided (*e.g.*, reachability analysis) on model checking tools such as UPPAAL [1].

USER INTERFACE

This section explains the visual modelling and debugging environment for Parallel DEVS models, based on the AToMPM tool [15]. First, we look at how Parallel DEVS models are modelled in AToMPM. Then, the debug operations are explained in more detail.

Modelling

In AToMPM, we created two formalisms: a "design formalism", which allows the visual modelling of Parallel DEVS models, and a "runtime formalism", which contains the runtime information that will be updated during simulation. The runtime model of a Parallel DEVS model is automatically generated from its design model. A design model consists of a single "Root" coupled model which contains references to other (atomic or coupled) DEVS models. Each atomic model has a number of named, sequential, states, that are connected with each other using the external, internal, and confluent transition functions. A state template for each atomic model defines its state variables. Event templates, which define their attributes, are modelled as well. Finally, the termination condition is modelled as an action code string.

A separate formalism 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 of each atomic DEVS model;
  - the time at which each atomic DEVS model is scheduled.

- To make the internal structure of the "black boxes" (references to DEVS models) in the design model explicit, for debugging purposes.

- To display instances of events.

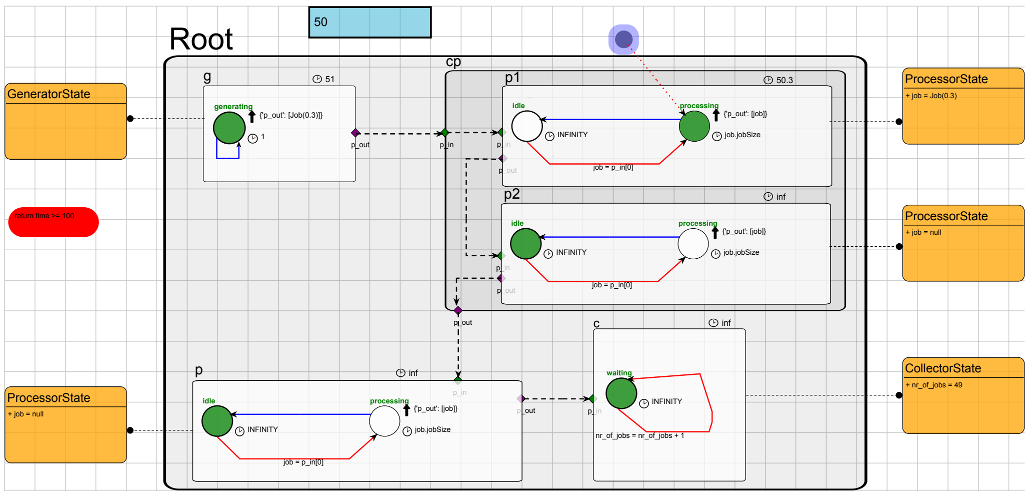


Figure 6. A breakpoint triggered in the visual debugging environment.

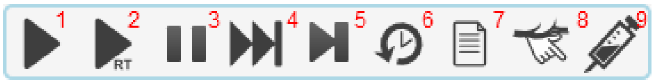


Figure 7. The debugging toolbar in AToMPM.

This boils down to expanding all references to atomic and coupled DEVS models found in the root model by replacing them with their definitions. This process stops until no more references are found. This expansion is implemented using an exogenous model transformation which transforms any valid design of a DEVS model into a runtime model.

Debugging

To support the definition of breakpoints, a third language was created: the *debugging* language. This language consists of only two concepts: a global breakpoint, which allows to specify a global condition on which the simulator should pause (similar to the termination condition, which takes into account the total state of the modelled system, accessible through calls to the PythonPDEVS simulator's interface), and a local breakpoint, which is connected to a sequential state and, optionally, declares an additional condition. The simulator will pause when the state connected to the breakpoint is entered and the condition holds. Figure 6 shows the state of the model after a breakpoint has been triggered, with the triggered breakpoint highlighted.

To support debugging, a toolbar was created in AToMPM, shown in Figure 7. By clicking on a button in this toolbar, a request is sent to the PyPDEVS server, representing a particular command. The server then forwards this request to PyPDEVS. As a result, a reply is sent back to the PyPDEVS server, where it is mapped to edit requests on the runtime model to visualize the changes.

The toolbar supports nine operations:

1. **Simulate** the model as-fast-as-possible, until either the termination 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.
2. **Realtime Simulate** the model. This means that the scheduler will try to meet all real-time deadlines, as specified in the time advance functions of the atomic DEVS models. Simulated time is interpreted in seconds. A scale factor can be specified for 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 will run. During realtime simulation, state changes are visualized in the user interface.
3. **Pause** will result in a running (as-fast-as-possible or realtime) simulation being paused as soon as possible. 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.
4. **Big Step** computes the next state of the system, which is visualized in the simulation environment.
5. A **small step** allows to visualize each phase of the next state computation. The list below lists, for each phase, what happens, and how it is visualized.
   (a) Determine the set of all atomic DEVS models whose internal transitions are scheduled to fire. These get highlighted in blue.
   (b) Execute output functions for each imminent component, which results in events being generated on their output ports. To visualize this, an event instance is visualized on the position of the output port.
   (c) Route events from output ports to input ports, while executing the transfer functions. Visually, the event instance is moved to the position of the input port.
   (d) Decide which transition to execute for every atomic DEVS model. This decision is made based on the model being imminent and/or receiving input. Models are highlighted in a color depicting the transition

	ADEVS [10]	MS4Me [13]	VLE [12]	X-S-Y [5]	PyPDEVS [18]
Pause	M	Y	N	Y	Y
(Scaled) Realtime	M	Y	N	Y	Y
Big Step	Y	Y	N	Y	Y
Small Step	M	N	N	N	Y
Termination Condition	Y	N	N	N	Y
Breakpoints	N	N	N	N	Y
Event Injection	M	Y	N	Y	Y
State Changes	N	N	N	N	Y
State Visualisation	M	M	N	M	Y
Event Visualisation	N	Y	N	N	Y
Tracing	M	Y	Y	Y	Y
Model Visualisation	N	Y	Y	N	Y
Reset	N	Y	M	Y	Y
Step Back	N	N	N	N	N
Experiment Debugging	N	N	N	N	N

Table 1. A comparison of the different tools.

function that will be executed: internal transitions are visualized in blue, external transitions in red, and confluent transitions in purple.

(e) Execute, in parallel, all enabled transition functions. The new state is shown in green, and the new values of its instance variables are displayed.

(f) Compute, for each atomic DEVS model, the time at which it gets scheduled (specified by its *time advance* function). This time is displayed next to a clock icon on top of each atomic DEVS model.

6. **Reset** the model and the simulation to their initial state.

7. **Show the Trace** (textual) of the simulation.

8. **Insert God Event** to manually change a state variable. This can only be done while the simulation is paused.

9. **Inject Event** on a specific port. This can only be done while the simulation is paused.

RELATED WORK

In Table 1, the functionality of five DEVS simulators is compared to our extended version of PythonPDEVS. For each function, we list whether the tool implements it (**Y** for yes, or **N** for no), or the user has to implement it manually (**M**).

Debuggers for some other formalisms already exist. In [9], Mustafiz and Vangheluwe construct a debugging environment with a technique similar to ours: they embed the model in the debugger, instead of the modal part of the simulator, however. In [6], Mannadiar and Vangheluwe address the need for debugging models in domain-specific languages and propose a mapping of code debugging concepts to model-based design. A debugger for Modelica was developed in [11].

CONCLUSION AND FUTURE WORK

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 and similar to that of traditional code debugging tools. The 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, we have explicitly modelled its modal part as a Statechart, as this seems to be the most natural choice of formalism to model a real-time, autonomous and reactive system. We combined the debugger with a visual environment which allows one to visually design and debug 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 was presented supporting our approach. This architecture is expected to be general enough to support debugging of models in multiple formalisms (and combinations thereof).

In the future, we want to extend this approach to other formalisms, such as Petrinets [8] and rule-based model transformations [14], which both introduce non-determinism. We will also continue enhancing the Parallel DEVS debugger with support for stepping back in time (omniscient debugging), logging (explicitly modelled in the visual modelling environment as blocks), and the debugging of (explicitly modelled) DEVS experiments, where a model is simulated a number of times with varying inputs and parameters.

Acknowledgement. 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. Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., and Yi, W. UPPAAL - a tool suite for automatic verification of real-time systems. In *Hybrid Systems III*, R. Alur, T. Henzinger, and E. Sontag, Eds., vol. 1066 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1996, 232–243.

2. Chow, A. C. H., and Zeigler, B. P. Parallel DEVS: a parallel, hierarchical, modular, modeling formalism. In *Proceedings of the 26th conference on Winter simulation* (1994), 716–722.

3. Chow, A. C. H., Zeigler, B. P., and Kim, D. H. Abstract simulator for the parallel DEVS formalism. In *AI, Simulation, and Planning in High Autonomy Systems* (1994), 157–163.

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

5. Hwang, M. H. X-S-Y. `https://code.google.com/p/x-s-y/`, 2014.

6. Mannadiar, R., and Vangheluwe, H. Debugging in domain-specific modelling. In *Software Language Engineering*, B. Malloy, S. Staab, and M. Brand, Eds., vol. 6563 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011, 276–285.

7. Mosterman, P. J., and Vangheluwe, H. Computer automated multi-paradigm modeling: An introduction. *Simulation 80*, 9 (Sept. 2004), 433–450.

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

9. Mustafiz, S., and Vangheluwe, H. Explicit modelling of Statechart simulation environments. In *Summer Simulation Multiconference*, Society for Computer Simulation International (SCS) (July 2013), 445 – 452. Toronto, Canada.

10. Nutaro, J. J. adevs. `http://www.ornl.gov/~1qn/adevs/`, 2014.

11. Pop, A., Sjölund, M., Asghar, A., Fritzson, P., and Francesco, C. Static and Dynamic Debugging of Modelica Models. In *Proceedings of the 9th International Modelica Conference* (Nov. 2012), 443–454.

12. Quesnel, G., Duboz, R., Ramat, E., and Traoré, M. K. VLE: a multimodeling and simulation environment. In *Proceedings of the 2007 summer computer simulation conference* (2007), 367–374.

13. Seo, C., Zeigler, B. P., Coop, R., and Kim, D. DEVS modeling and simulation methodology with ms4me software. In *Symposium on Theory of Modeling and Simulation - DEVS (TMS/DEVS)* (2013).

14. Syriani, E., and Vangheluwe, H. A modular timed graph transformation language for simulation-based design. *Software and Systems Modeling (SoSyM) 12*, 2 (2013), 387–414.

15. Syriani, E., Vangheluwe, H., Mannadiar, R., Hansen, C., Van Mierlo, S., and Ergin, H. AToMPM: A web-based modeling environment. In *MODELS'13 Demonstrations* (2013).

16. Van Mierlo, S., Barroca, B., Vangeluwe, H., Syriani, E., and Kühne, T. Multi-level modelling in the Modelverse. In *Multi-Level Modelling Workshop (MULTI 2014) Proceedings* (2014).

17. Van Mierlo, S., Van Tendeloo, Y., Mustafiz, S., Barroca, B., and Vangheluwe, H. Debugging Parallel DEVS. Tech. rep., University of Antwerp and McGill University, 2014. `http://msdl.cs.mcgill.ca/people/simonvm/devsdebuggerreport.pdf`.

18. Van Tendeloo, Y., and Vangheluwe, H. The modular architecture of the Python(P)DEVS simulation kernel. In *Proceedings of the 2014 Symposium on Theory of Modeling and Simulation - DEVS* (2014), 387–392.

19. Vangheluwe, H. DEVS as a common denominator for multi-formalism hybrid systems modelling. *CACSD. Conference Proceedings. IEEE International Symposium on Computer-Aided Control System Design* (2000), 129–134.

20. Vangheluwe, H. Foundations of modelling and simulation of complex systems. *ECEASST 10* (2008).

21. Vangheluwe, H., Riegelhaupt, D., Mustafiz, S., Denil, J., and Van Mierlo, S. Explicit modelling of a CBD experimentation environment. In *Proceedings of the 2014 Symposium on Theory of Modeling and Simulation - DEVS*, TMS/DEVS '14, part of the Spring Simulation Multi-Conference, Society for Computer Simulation International (2014), 379 – 386.

22. Zeigler, B. P., Praehofer, H., and Kim, T. G. *Theory of Modeling and Simulation*, second ed. Academic Press, 2000.

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