Debugging Parallel DEVS

Simon Van Mierlo
Yentl Van Tendeloo
Sadaf Mustafiz
Bruno Barroca

November 10, 2014

Abstract

In this technical report, a detailed explanation of a visual debugger for the Parallel DEVS formalism is presented. The debugging environment and simulator are explicitly modelled. We achieve this by deconstructing the coded PythonPDEVS simulator and subsequently reconstructing it. The modal part of the PythonPDEVS simulator is explicitly modelled as a Statechart, after which it is enhanced with debugging operations. We take inspiration from the code debugging world, as well as from the simulation world for the debugging operations: we support as-fast-as-possible and (scaled) real-time simulation, as well as manually stepping through the simulation, breakpointing, injecting events, and manually changing the state. We develop a design, runtime, and debug formalism in AToMPM to visually model, simulate, and debug Parallel DEVS models. A toolbar is provided in the simulation environment which allows users to perform debugging operations on the running simulation.

Contents

1 Introduction

The systems we analyse, design, and develop today are characterized by an ever growing complexity. Modelling and Simulation (*M&S*) [1] 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 (software-intensive) system being developed and subsequently simulate them having a set of "goals" or desired properties in mind. Every aspect of the system is modelled, at the most appropriate level of abstraction, using the most appropriate formalism [2]. The M&S approach can only be successful if there is sufficient tool support, *i.e.*, if the modeller has access to tools which sufficiently support each phase in the M&S approach. 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 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 [3].

Support for simulation debugging is currently limited. Some early work has been performed in this direction; for instance, Mannadiar and Vangheluwe [4] survey the state-of-the-art in debugging and explore how these concepts can be transposed to the realm of Domain-Specific Modelling. Model debugging has received some attention in the literature on the Modelica [5] language. In [6, 7, 8, 9], the authors develop techniques for debugging equation-based models, which differ greatly from sequential programs, where each statement is executed one after the other. They look at static and dynamic debugging, as well as how to make the debugging techniques scalable for large models. In [10], Mustafiz and Vangheluwe explicitly model a debugger for Statecharts [11] by embedding the modelled Statechart in the Statechart describing the modal behaviour of the simulator, and instrumenting it with debugging-specific operations. Most recently, Vangheluwe et al. have explicitly modelled a visual debugging environment for Causal-Block Diagram (CBD) [12] models in [13].

In this technical report, we present a visual debugger for Parallel DEVS models [1]. In the following sections, we introduce the Parallel DEVS formalism, as well as the challenges related to simulation debugging.

1.1 Parallel DEVS

The Parallel DEVS formalism consists of atomic models, which are connected in a coupled model. The atomic models are behavioural (they exclusively model behaviour), while the coupled models are structural (they exclusively model structure).

Atomic DEVS

An *atomic DEVS* model is a structure:

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

The *input set X* denotes the set of admissible inputs of the model. The input set may have multiple ports, denoted by m in this definition. X is a structured set

$$X = \times_{i=1}^{m} X_i$$

[1] The implementation can be downloaded from `http://msdl.cs.mcgill.ca/people/simonvm/devsdebugger.zip`

where each of the X_i denotes the admissible inputs on port i. The *output set* Y denotes the set of admissible outputs of the model. The output set may have multiple ports, denoted by l in this definition. Y is a structured set

$$Y = \times_{i=1}^{l} Y_i$$

where each of the Y_i denotes the admissible outputs on port i. The *state set* S is the set of admissible sequential states. Typically, S is a structured set

$$S = \times_{i=1}^{n} S_i$$

The *internal transition function* δ_{int} defines the next sequential state, depending on the current state. It is triggered after the time returned by the *time advance function* has passed (in the simulation, not in wall-clock time). Note that this function does not require the elapsed simulation time as an argument, since it will always be equal to the *time advance function*.

$$\delta_{int} : S \to S$$

The *output function* λ maps the sequential state set onto an output bag. Output events are only generated by a DEVS model at the time of an *internal/confluent transition*. This function is called *before* the transition function is called, so the state that is used will be the state before the transition happens.

$$\lambda : S \to Y^b$$

The *external transition function* δ_{ext} gets called whenever an *external input* ($\in X$) is received in the model. The signature of this transition function is

$$\delta_{ext} : Q \times X^b \to S$$
$$\text{with } Q = \{(s,e) | s \in S, 0 \leq e \leq ta(s)\}$$

with e the elapsed time since the last transition.

When the *external transition function* is called, the *time advance function* is called again and the previously scheduled internal event is rescheduled with the new value. The *time advance function* ta defines the simulation time the system remains in the current state before triggering the *output functions* and *internal transition functions*.

$$ta : S \to \mathbb{R}^{+}_{0,+\infty}$$

Note that $+\infty$ is included, since it is possible for a model to *passivate* in a certain state, meaning that it will never have an internal transition in this state.

Should the internal and external transition function be called at exactly the same point in simulated time, the *confluent transition function* is called instead. It is defined as

$$\delta_{conf} : S \times X^b \to S$$

Coupled DEVS

A Coupled DEVS is a structure

$$M = < X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\} >$$

With X and Y the *input* and *output* set respectively.

D is the set of unique component references (names), the coupled model itself is not an element of D.

$\{M_i\}$ is the set of components containing the atomic DEVS structure of all subcomponents referenced by elements in D.

$$\{M_i | i \in D\}$$

The set of *influencees* $\{I_i\}$ determines the elements whose input ports are connected to output ports of component i. Note that $self$ is included in this definition, as a component can send (receive) messages to (from) the coupled model itself.

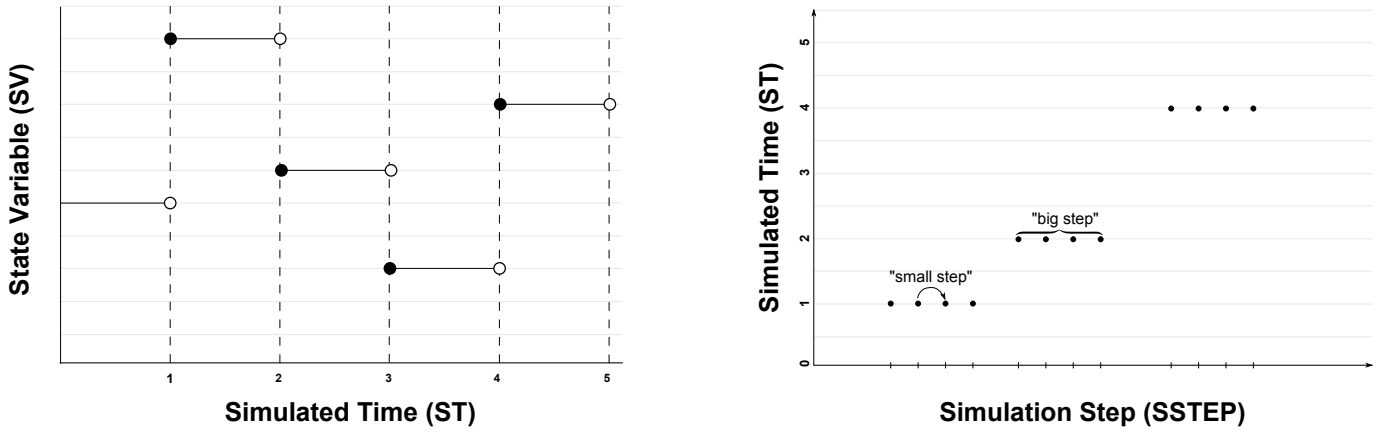
$$\{I_i | i \in D \cup \{self\}\}$$

A component cannot influence components outside of the current coupled model, nor can it influence itself directly as self-loops are forbidden.

$$\forall i \in D \cup \{self\} : I_i \subseteq D \cup \{self\}$$
$$\forall i \in D \cup \{self\} : i \notin I_i$$

The couplings are further specified by the *transfer functions* $Z_{i,j}$. These functions are applied to the messages being passed, depending on the output and input port. These functions allow for reuse, since they allow output events to be made compatible with the input set of the connected models.

$$\{Z_{i,j} | i \in D \cup \{self\}, j \in I_i\}$$



(a) Change of the state variable over time.

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

Figure 1.1: Simulation time and steps.

1.2 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 shown visually in Figure 1.1a. The state is updated by some computations, or "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 1.1b.

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, have an additional notion of time: the *simulated time*. A simulation 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 shown visually in Figure 1.2. Note that there is no linear relation in as-fast-as-possible simulation, meaning that the "current simulation time" is simply a variable in the simulator. Moreover, operations can be performed on simulated time, such as pausing, or stepping back, which are 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. Above we already mentioned that it is a valid operation on simulated time. 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 1.3 visually shows 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". For simplicity, we assume a "big step" here occurs each second. This is, of course, not necessarily the case in each DEVS model. In as-fast-as-possible mode, these computation parts are executed one after the other, without any waiting period in between. This means that if a pause is requested (denoted by the red vertical bar with "pause" next to it), the simulator is always computing the next state. Only after completing this step, will simulation be paused. The system will be in the next state, as computed by the big step.

In real-time mode, simulated time is "synchronized" with the wall-clock time. The time needed to compute the next state is of course still there, but now the simulator will wait in between these computation periods to let the simulated time synchronize with the wall-clock time. This is represented by the continuous function, which tries to follow the ideally synchronized line, represented by the grey dotted line. Of course, when a computation is performed, the wall-clock time advances, and as a result, the simulated time is desynchronized from the wall-clock time. This is represented by the horizontal parts in the function. When the computation is finished, the simulator will then synchronize time immediately, as depicted by the vertical parts. If a pause is requested during a waiting period, the simulator immediately breaks. 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 simulation time will be somewhere in between the previous transition time and the next.

As a consequence of this difference, it is impossible for an as-fast-as-possible simulation to pause at times in between 2 different simulation times. On the other hand, realtime simulation can pause at virtually every point in simulated time. The notable exception being the time at which transition functions are being computed.

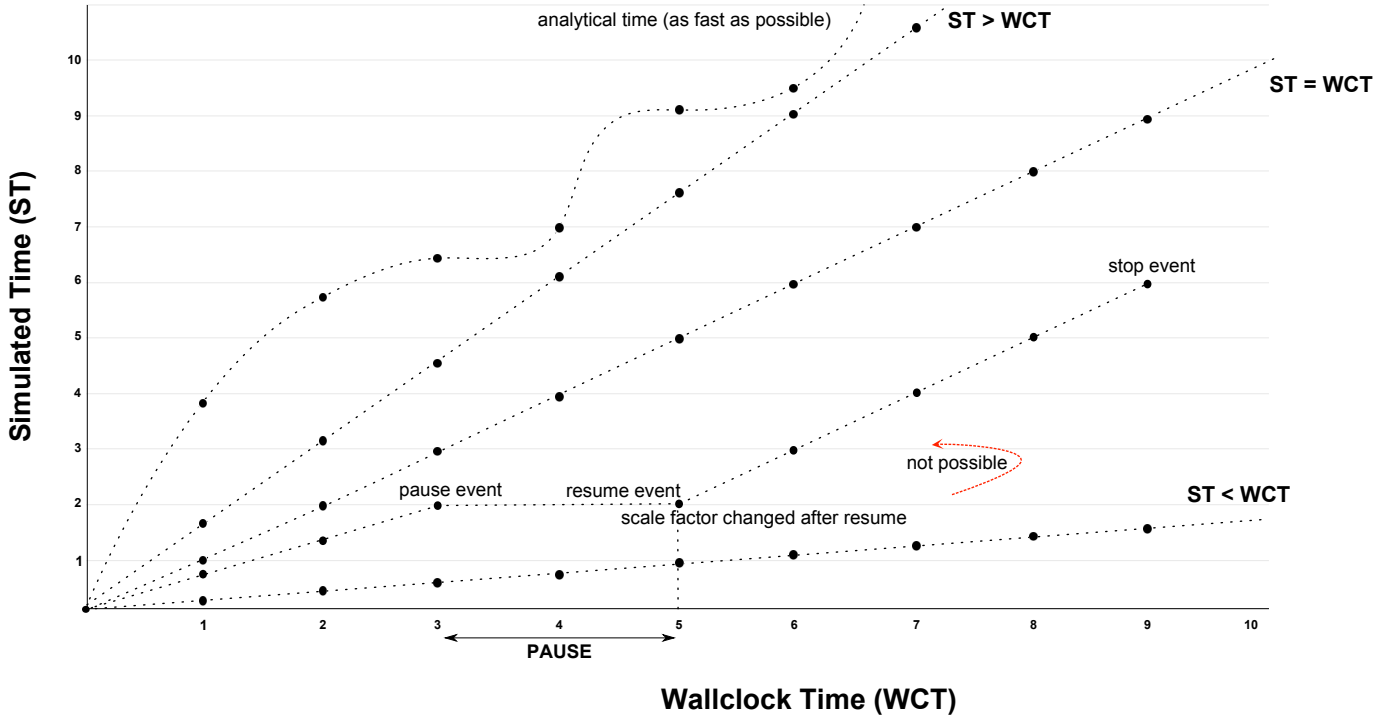


Figure 1.2: Different notions of simulated time.

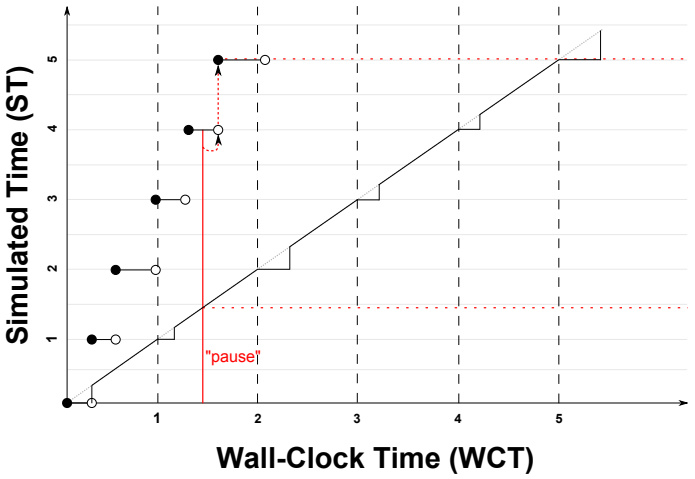


Figure 1.3: Pausing simulation: difference between as-fast-as-possible and real-time simulation.

1.3 User Interaction

Users interact with a simulation through the simulation environment. The interleaving of user events coming from the environment with the real-time, interruptible behaviour of the simulator (or interacting simulators, in the case of hybrid system simulation), is non-trivial.

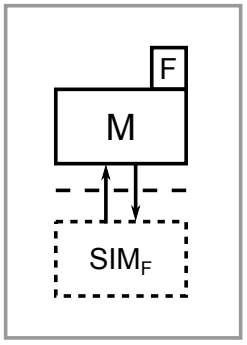
The challenge is to manage the inherent complexity of constructing these model debugging and experimentation environments. The interplay of formalism execution semantics, different notions of simulated time, and user interaction makes this a challenging task if traditional software development methods are used. While examples exist of model simulation debuggers implemented in code, it is clear that different techniques are needed to overcome this complexity.

1.3.1 De/Reconstructing the Simulator

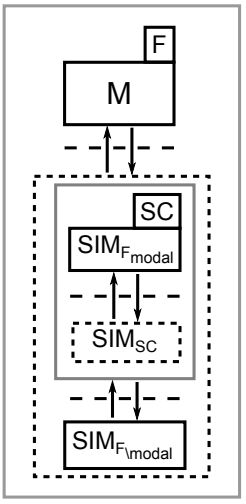
To manage the complexity inherent to building a real-time, interruptible simulator, enhanced with debugging operations, we propose to explicitly model the debugging environment using Statecharts. In general, each simulator has a "main simulator loop". This comprises a number of states and transitions between them, performing some action that updates the state of the model. In the most naive case, there is a single state which represents the main loop of the simulator and a transition going from and to that state, each time performing one simulation "step". This is the so-called "modal part" of a simulator, which is intuitively represented as a Statechart.

The process of extracting this modal part, which we call de/reconstructing the simulator, is shown in Figure 1.4. The first step, deconstrucing the simulator, extracts the modal part of the simulator in a Statechart (SC) model called $SIM_{F_{modal}}$. This model is combined with a Statechart simulator, interpreter, or compiler called SIM_{SC} to give it operational semantics. The Statechart together with its executor interface with the non-modal part of the simulator for formalism F ($\text{SIM}_{F_{\setminus modal}}$, which, in this case, consists of the coded functions to run the simulator). The combination of the modal and non-modal part of the simulator results in a behaviourally equivalent simulator to SIM_F. From the user's point of view, the black-box containing the model to be simulated and its simulator is unchanged.

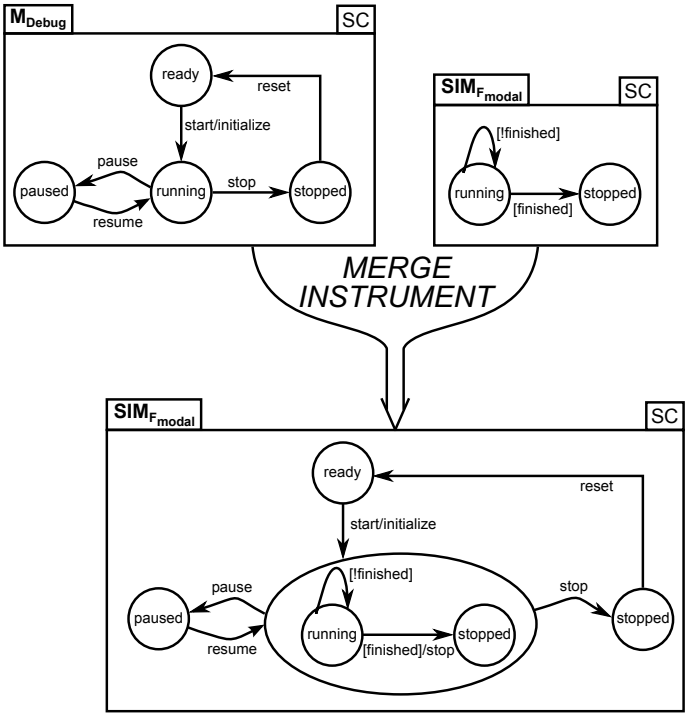
In Figure 1.4c, the last step in creating an instrumented simulator is shown. We *merge* the modal part of the simulator for F with the behavioural model of the debugger. This results in an instrumented model of the modal behaviour of the simulator. The last step is to replace $SIM_{F_{modal}}$ in Figure 1.4b with this instrumented model. Again, this should 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 (trivial, but representative) example shown, the debugger includes the concepts of *start*, *pause*, *resume*, and *stop*. The simulator only has two states: *running*, and *stopped*. It runs the main loop of the simulator until the *finished* condition satisfied, signalling that the simulation is done.



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



(b) De/Reconstructing the simulator.



(c) Merging the debugging concepts with the modal behaviour of the simulator.

Figure 1.4: The workflow for explicitly modelling the simulator's behaviour.

2 Related Work

In this chapter, we look at a number of popular DEVS simulation environments. We discuss what kind of control over the simulation they offer to the user, and compare them with our own tool PythonPDEVS.

2.1 ADEVS

ADEVS [18] is a DEVS simulator written in C++, and makes heavy use of templates. The tool focusses on performance and lightweightness, and does not offer a user interface. In fact, a simulation program has to be written by the user using the API functions ADEVS provides, and a single executable is generated for each simulation. This results in ADEVS being highly customizable, but not user-friendly, as simulating a DEVS model requires the modeller to be proficient in the C++ language, which is not necessarily the case. Furthermore, there is no built-in support for setting termination conditions, or to trace the state of the system. Everything has to be manually implemented.

2.2 MS4 Modelling Environment

The MS4 Modelling Environment [20] is an Eclipse-based environment for Parallel DEVS modelling and simulation. Models are created in a textual notation, after which they are compiled to a Java-file. In Figure 2.1, an example of an Atomic DEVS model is shown. It models a Customer in a bank system example. The customer first sends a message to the bank teller and then waits for a reply. It then requests for a withdrawal and waits until it receives the money.

In Figure 2.2, the example of the bank system is visualized. This view allows for the visual simulation of the model, and provides some debugging support. The atomic DEVS models are visualized as boxes, showing the name of the model along with its current state between brackets. The operations supported by the environment, as well as their semantics, are the following:

- **Restart** restarts the simulation from the start.
- **Pause** allows the simulation to be paused after the current step is finished.
- **Run** runs the simulation until the end.
- **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.*, compute output, execute transition functions, and compute the time advance.
- **Run to...** allows to specify a fixed number of iterations or a simulation time at which to stop the simulation.
- Inputs can be injected by right-clicking an atomic DEVS model and choosing the appropriate option.

The environment displays some simulation runtime variables: the current time, the time at which the last event was processed, the time at which the next event is scheduled, and how many iterations, as well as (internal, external, and confluent) transitions have been executed.

```
accepts input on Withdrawal !
generates output on RequestWithdrawal !

to start, hold in sendHello for time 1!
after sendHello output Hello!
from sendHello go to waitforHi!
passivate in waitforHi !
when in waitforHi and receive Hi go to sendRequestWithdrawal!
hold in sendRequestWithdrawal for time 1!
after sendRequestWithdrawal output RequestWithdrawal!
from sendRequestWithdrawal go to waitforWithdrawal!
passivate in waitforWithdrawal !
when in waitforWithdrawal and receive Withdrawal go to passive!
passivate in passive!
```

Figure 2.1: Modelling in MS4 Modelling Environment

2.3 DEVS Suite

DEVS Suite [19] offers a visual debugging environment for the interactive simulation of DEVS models. An example simulation of a DEVS models with three processor models is shown in Figure 2.3. 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, as well as their semantics, are the following:

- **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.
- 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 the Figure, messages are visualized on the output ports when they are generated, and transition to the input ports when they are sent.

2.4 VLE: The Virtual Laboratory Environment

VLE [21] supports the modelling and simulation of DEVS models. Atomic DEVS models are to be written in C++ code, but Coupled DEVS models can be created both textually or visually (using GVLE). Simulation of the created models is supported, but there is no support for any form of debugging.

2.5 X-S-Y

X-S-Y [22] is a tool 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. 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).

Figure 2.2: Debugging in MS4 Modelling Environment

Figure 2.3: Debugging in DEVS Suite

	ADEVS	DEVS Suite	MS4Me	VLE	X-S-Y	PythonPDEVS
Pause	M	Y	Y	N	Y	Y
(Scaled) Realtime	M	Y	Y	N	Y	Y
Big Step	Y	Y	Y	N	Y	Y
Small Step	M	N	N	N	N	Y
Termination condition	Y	N	N	N	N	Y
Breakpoints	N	N	N	N	N	Y
Event Injection	M	Y	Y	N	Y	Y
State Changes	N	N	N	N	N	Y
State Visualisation	M	M	M	N	M	Y
Event Visualisation	N	Y	Y	N	N	Y
Tracing	M	Y	Y	Y	Y	Y
Model Visualisation	N	Y	Y	Y	N	Y
Reset	N	Y	Y	M	Y	Y
Step back	N	N	N	N	N	N

Table 2.1: A comparison of the different tools discussed in this chapter.

  - **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** - 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.

2.6 Comparison

In Table 2.1, the functionality of the five tools are compared to PyPDEVS. For each function, we list whether or not the tool implements it (**Y** for yes, or **N** for no), or the user has to implement it manually (**M**).

3
Solution Architecture

The architecture of the DEVS debugger, with all its components visually represented, can be found in Figure 3.1. There are four main components to our solution architecture:

1. **AToMPM** [23] is a visual modelling and simulation tool developed by the Modelling, Simulation, and Design Lab (MSDL). It allows to create (domain-specific) modelling language by modelling its abstract syntax and static semantics, and model transformations for operational or denotational semantics. It is extensible, allowing functionality to be plugged into the front-end, as well as on the server side, by extending the source code.

   AToMPM will be used to visually model Parallel DEVS models in a design language, and execute them by mapping them onto a runtime formalism. Combined with a debugging tool bar, this effectively creates a visual runtime and debugging environment which shows the state of the system during simulation and allows to manipulate the simulation process by sending requests to the back-end.

2. The **Modelverse** [24] is a model repository and metamodelling framework, and will, in the future, serve as the back-end for AToMPM. 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. Parallel DEVS models are exported from AToMPM to HUTN, and stored in the Modelverse. A check is implemented to make sure that the action code (which is also modelled explicitly in the Modelverse) of the transition functions does not access anything illegally, *i.e.*, attributes that are not part of the state. If it is valid, the model is then exported to the format our simulator expects.

3. **PythonPDEVS** [25] is our simulator, which has been de/reconstructed, as described in Chapter 1. Its modal behaviour has been explicitly modelled, enhanced with debugging support. 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 of AToMPM, the Modelverse, and the PythonPDEVS simulation engine.

In the following chapters, we will explain each component in more detail.

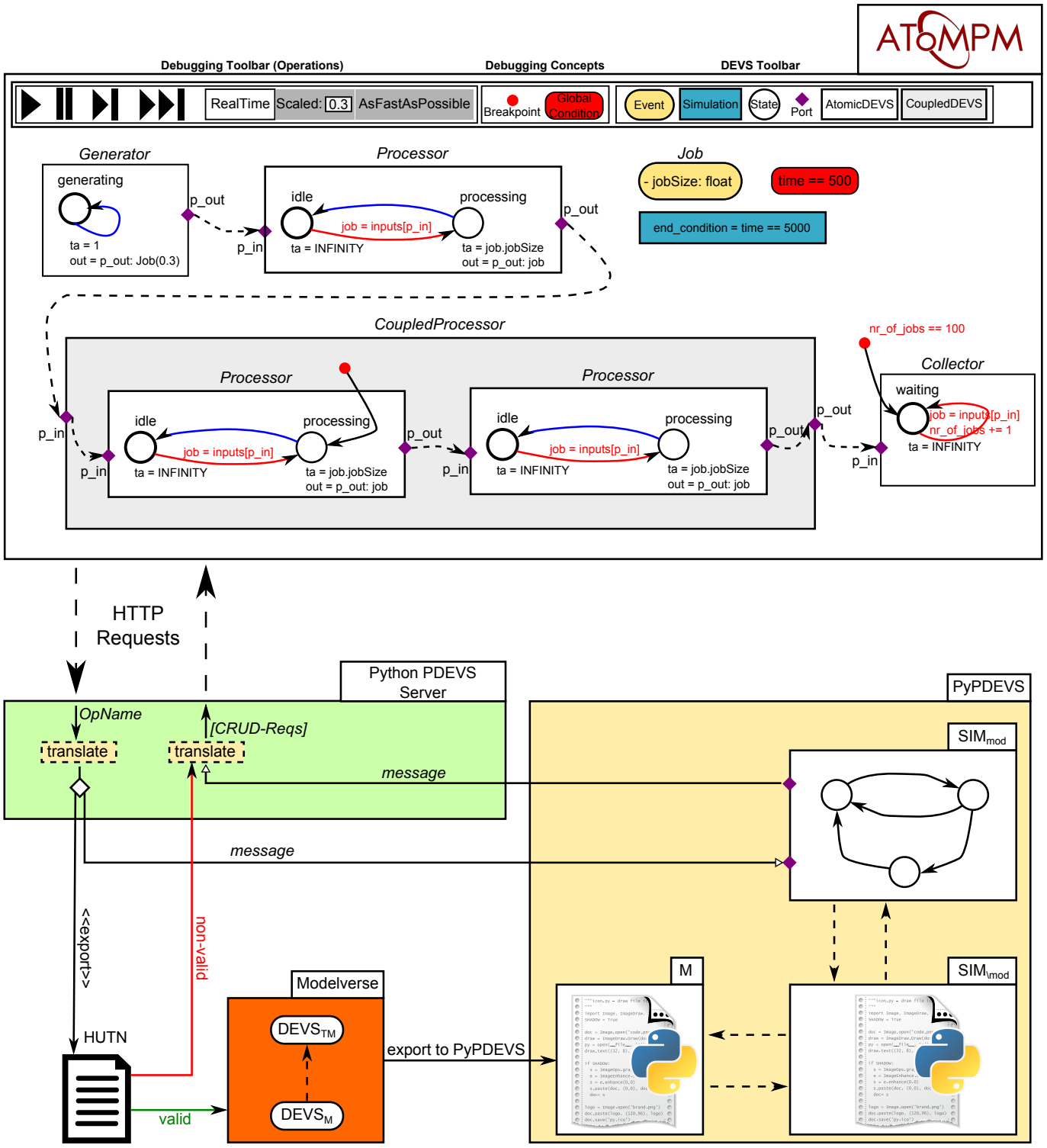


Figure 3.1: Architecture

4 AToMPM

AToMPM will act as the visual debugging/experimentation/modelling environment for Parallel DEVS models. For **modelling**, it provides a visual formalism which allows to specify Parallel DEVS models in an intuitive and user-friendly way. Models in this formalism can be exported to the Modelverse, from where they are exported to PyPDEVS to be simulated.

A number of formalisms are used for the design, simulation, and debugging of Parallel DEVS models. These are explained in the following sections.

4.1 Design Formalism

The design formalism is called "ParallelDEVS" and allows to visually model a Parallel DEVS instance.

In Figure 4.1, the metamodel for the DEVS formalism in AToMPM is shown. It consists of the following elements:

- **BaseDEVS** is the superclass for the Coupled DEVS and Atomic DEVS classes. It has the following attributes:
  - **name** is the name of the model. It can be used in instances to refer to their class.
  - **attributes** is a list of attributes for this model.
  - **parameters** is a list of parameters passed to the constructor of this model.
  - **__init__** is the constructor for this model. Note that here, its attributes should be initialized with the correct values. For this, the code can access the parameters by name.
  - **position** and **scale** are visualization-specific attributes.
- **Port** models ports of DEVS models. There are two types of ports: input ports and output ports. Each port has a name, and they can be connected to other ports through the **channel** association. A transfer function can be defined for translating events when they are transferred through such a channel.
- **CoupledDEVS** instances can contain a number of submodels (which are instances of other atomic/coupled DEVS models). Exactly one CoupledDEVS model should be called **Root** and is the root model used for simulation.
- **DevsInstance** is an instantiation of an atomic/coupled DEVS model, which can be found in the same model. It has the following attributes:
  - **name** is the name of the instance. It has to be unique within the enclosing CoupledDEVS model.
  - **devs_type** is a reference to the atomic/coupled DEVS model this is an instance of. It needs to be defined in the same model as the reference.
  - **parameter_binding** is a mapping of parameter names onto values passed to the constructor of the atomic/coupled DEVS model this is an instance of.
  - **position** and **scale** are visualization-specific attributes.
- **AtomicDEVS** models contain a number of states, and are connected to exactly one state definition.
- **StateDefinition** models the "template" of a state for a particular AtomicDEVS model. It has the following attributes:
  - **name** is the name of this state definition.

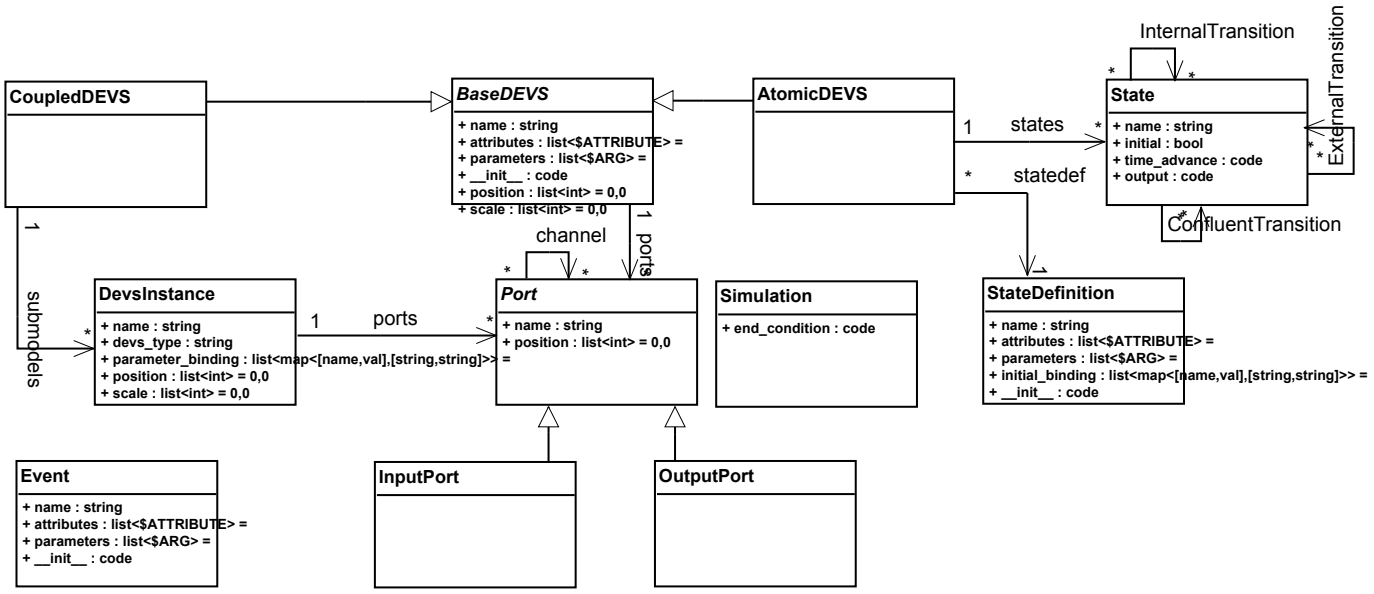


Figure 4.1: The Parallel DEVS Metamodel in AToMPM

- **attributes** is a list of attributes for this state definition. It is a list of "state variables".
- **parameters** is a list of parameters for the constructor, which is called when a state is entered.
- **__init__** is the code for the constructor.

- **State** is a state in an AtomicDEVS model. It has the following attributes:
  - **name** is the name of the state.
  - **initial** marks the initial state of the AtomicDEVS model.
  - **time_advance** is the block of code which computes the time the AtomicDEVS model should stay in this state before firing its internal transition. It should return a positive floating point number, or INFINITY.
  - **output** is the block of code which computes the output which should be generated after leaving this state. It should return a mapping between port names and a list of event instances.
- **InternalTransition** connects two states with each other, and specifies which state to transition to after the time advance of the current state has passed. It has two attributes:
  - **condition** allows to specify a condition which needs to evaluate to true in order for the internal transition to fire. Note that two internal transitions going out from the same state should have non-overlapping conditions.
  - **action** allows to specify an action when performing the transition. Generally, this action code will return the mapping between parameter names of the state it transitions to and their values. Note that a state is identified by its name, and this is an attribute of all state definitions by default. It is automatically passed as a parameter as well. The action code should only return any additional parameters.
- **ExternalTransition** connects two states with each other, and specifies which state to transition to after an input was received on some input port. It has two attributes:
  - **condition** allows to specify a condition which needs to evaluate to true in order for the external transition to fire. Note that two external transitions going out from the same state should have non-overlapping conditions.
  - **action** allows to specify an action when performing the transition. Generally, this action code will return the mapping between parameter names of the state it transitions to and their values. Note that a state is identified by its name, and this is an attribute of all state definitions by default. It is automatically passed as a parameter as well. The action code should only return any additional parameters. This action code can access the input map called "inputs". This maps all port names on the bag of inputs received. Furthermore, "self.elapsed" contains the elapsed time since the last external transition.
- **ConfluentTransition** connects two states with each other, and specifies which state to transition to when there is a conflict between internal and external transition function. By default, the internal transition function followed by the external transition is fired. It has two attributes:
  - **condition** allows to specify a condition which needs to evaluate to true in order for the confluent transition to fire. Note that two confluent transitions going out from the same state should have non-overlapping conditions.

    - **action** allows to specify an action when performing the transition. Generally, this action code will return the mapping between parameter names of the state it transitions to and their values. Note that a state is identified by its name, and this is an attribute of all state definitions by default. It is automatically passed as a parameter as well. The action code should only return any additional parameters. This action code can access the input map called "inputs". This maps all port names on the bag of inputs received.
- **Simulation** allows to specify an end condition for the simulation. The global simulation time can be accessed with the variable `time`. The current state of the model can also be accessed using the `model` variable.

In Figure 4.2, an example produce-consume model is shown in the generated AToMPM modelling environment. It consists of the following AtomicDEVS 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 (blocks) for a predetermined amount of time, and then outputs the Job on its output port.

Each AtomicDEVS model has a state definition associated with it. GeneratorState does not declare any new attributes, it only has a *name*. ProcessorState keeps track of the current job being processed: in the *idle* state this will be None, in the *processing* state it will contain the currently processing job. GeneratorState contains the number of received messages. This counter is updated each time a message arrives on the input port of the Generator.

The root coupled DEVS model connects a Generator with a CoupledProcessor (which contains two Processors), a Processor, and a Collector at the end. Note that DEVS instances are "black boxes" in the design model: we cannot see its internal states and transitions. This is to prevent duplication of the internals. In the runtime model, which will be used for simulation, these will be visible. A simple model transformation based on the names of the referenced models will fill in (duplicate) all the details.

This model can be exported to PyPDEVS using a plugin, where it can be simulated. A toolbar can be loaded from **/Toolbars/-ParallelDEVS/Export.buttons.model**. It consists of one button, as shown in Figure 4.3. Clicking the button will create two files in the **exported_to_pypdevs** folder in the root AToMPM folder, called **model.py** and **experiment.py**. Running the experiment will simulate the model.

4.2 Runtime Formalism

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 (for each Atomic DEVS model).
    - The attribute values for states and Coupled/Atomic DEVS models.
    - The time at which each Atomic DEVS model will transition next.
- 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 messages.

This boils down to expanding all references to Atomic/Coupled DEVS models found in the root model: they are replaced by their definitions. This process stops until no more references are found in the model. This expansion is implemented using an exogenous model transformation which transforms any valid design of a DEVS model into a runtime model.

Figure 4.4 shows the metamodel of the DEVS runtime formalism, as modelled in AToMPM. It is quite similar to the design formalism shown in Figure 4.1, but tailored to representing runtime, simulation-specific information. In particular, these are the changes made:

- DEVS models, events and states keep track of their attribute values in the *attribute_values* attribute.
- The *current* attribute was added to the *State* class to keep track of the current state of each Atomic DEVS model.
- The *time_next* attribute is used to signify, for each Atomic DEVS model, when its internal transition function is scheduled to execute.

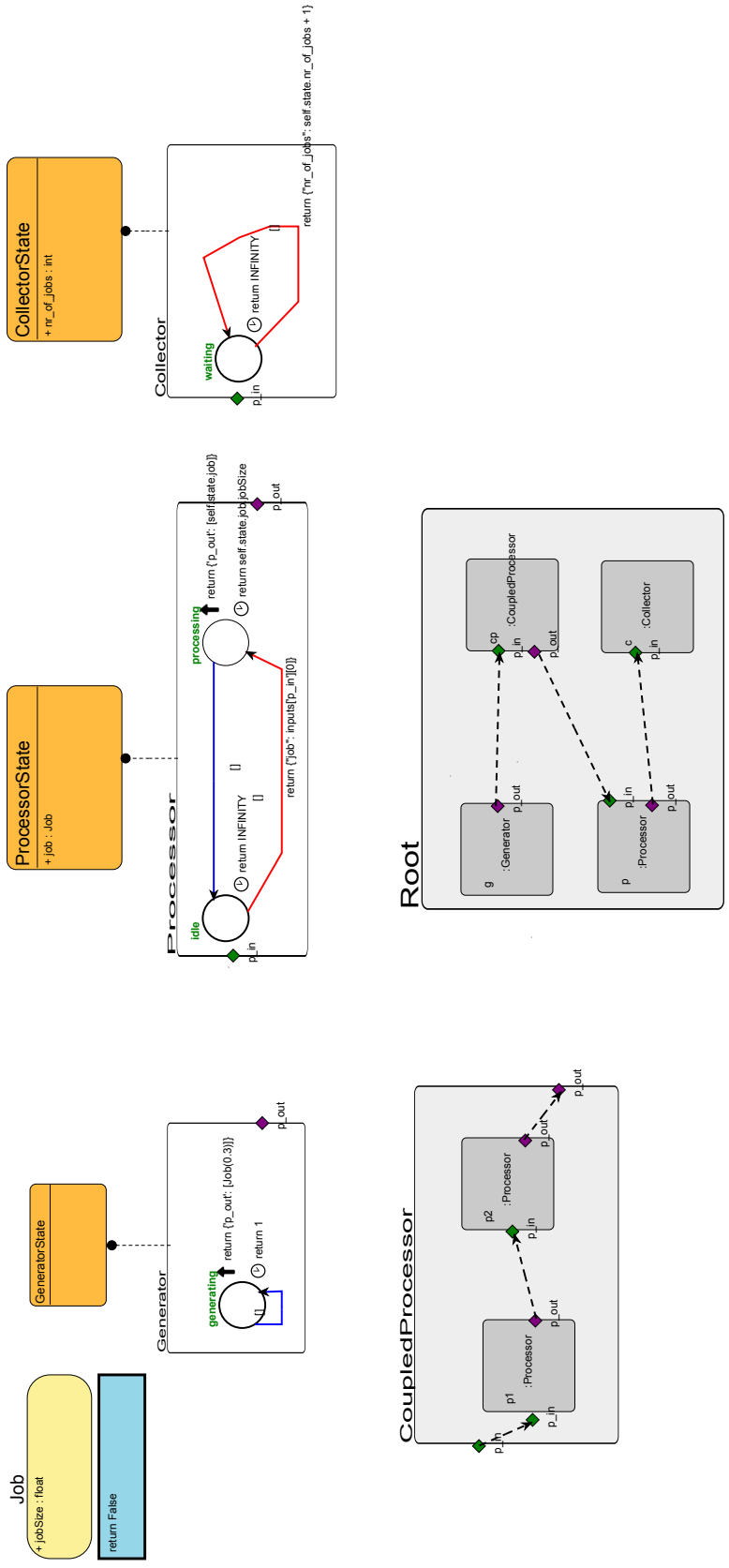


Figure 4.2: Example Produce-Consume Model in AToMPM

Figure 4.3: The button which exports a Parallel DEVS model from AToMPM to PyPDEVS.

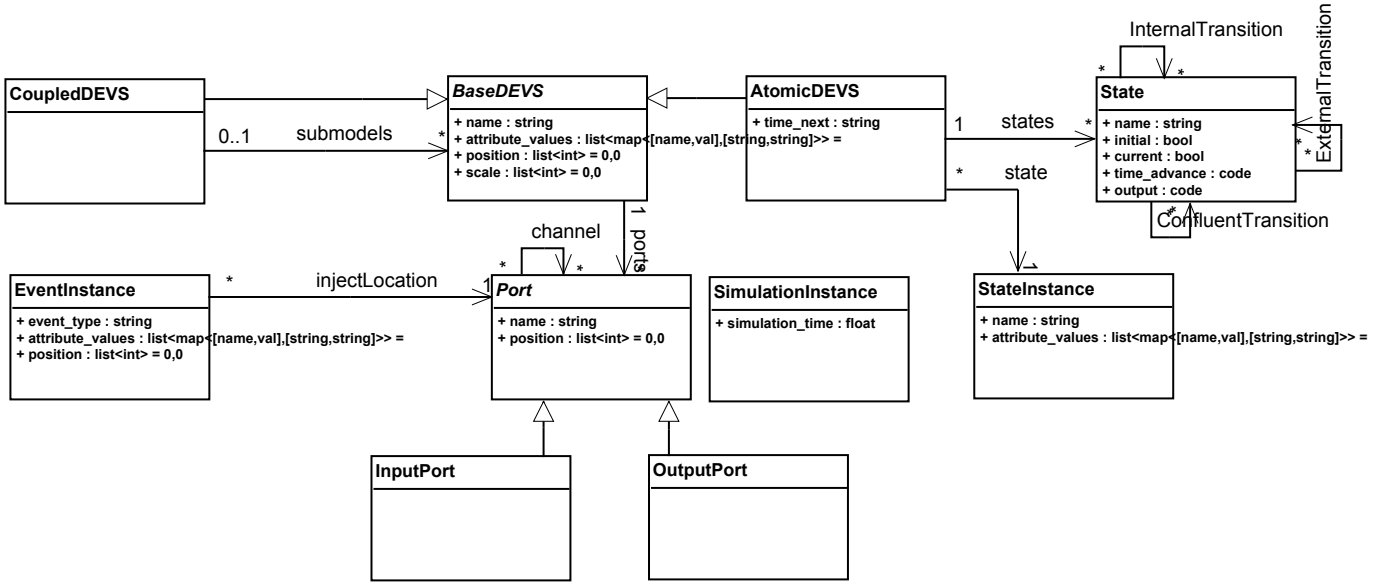


Figure 4.4: The metamodel of the Parallel DEVS runtime formalism.

- The user can connect an event to a port, effectively injecting it on that port. It is possible to customize the time at which the event should be injected: either right now, or in the future. Injecting events in the past is not supported.
- The simulation keeps track of the current simulation time in an instance of the *SimulationInstance* class.

Figure 4.5 shows the runtime model of the producer-consumer example, of which the design is shown in Figure 4.2.

4.3 Debugging Formalism

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 break (similar to a termination condition), and a local breakpoint, which is connected to a state and, optionally, declares an additional condition. The simulator will break when the state connected to the breakpoint is entered and the condition holds.

Figure 4.6 shows the metamodel for the debug language. It consists of two concepts:

- **GlobalBreakpoint** allows to specify a global breakpoint. It has three attributes:
  - **condition** contains action code that returns true when the simulation should break. It has access to three variables:
    - **time**, the current simulation time.
    - **model**, the root PyPDEVS model.
    - **transitioned**, a list of Atomic DEVS models whose state has changed in the last simulation step.
  - **enabled** specifies whether the breakpoint is enabled.
  - **disable_after_trigger** specifies whether the breakpoint should be disabled (setting the enabled attribute to false) when the breakpoint has been triggered. Note that *not* checking this option results in the breakpoint evaluating to true continuously once it has been triggered, effectively disabling simulation.
- **Breakpoint** has the same attributes as *GlobalBreakpoint*, but additionally is connected to a state of an Atomic DEVS model. The breakpoint will trigger whenever this state is entered *and* its condition holds.

4.4 Debugging Toolbar

To support debugging, a toolbar was created in AToMPM, shown in Figure 4.7. By clicking on a button in this toolbar, a request will be sent to the PyPDEVS server, representing a particular command. The server will send the appropriate request to the

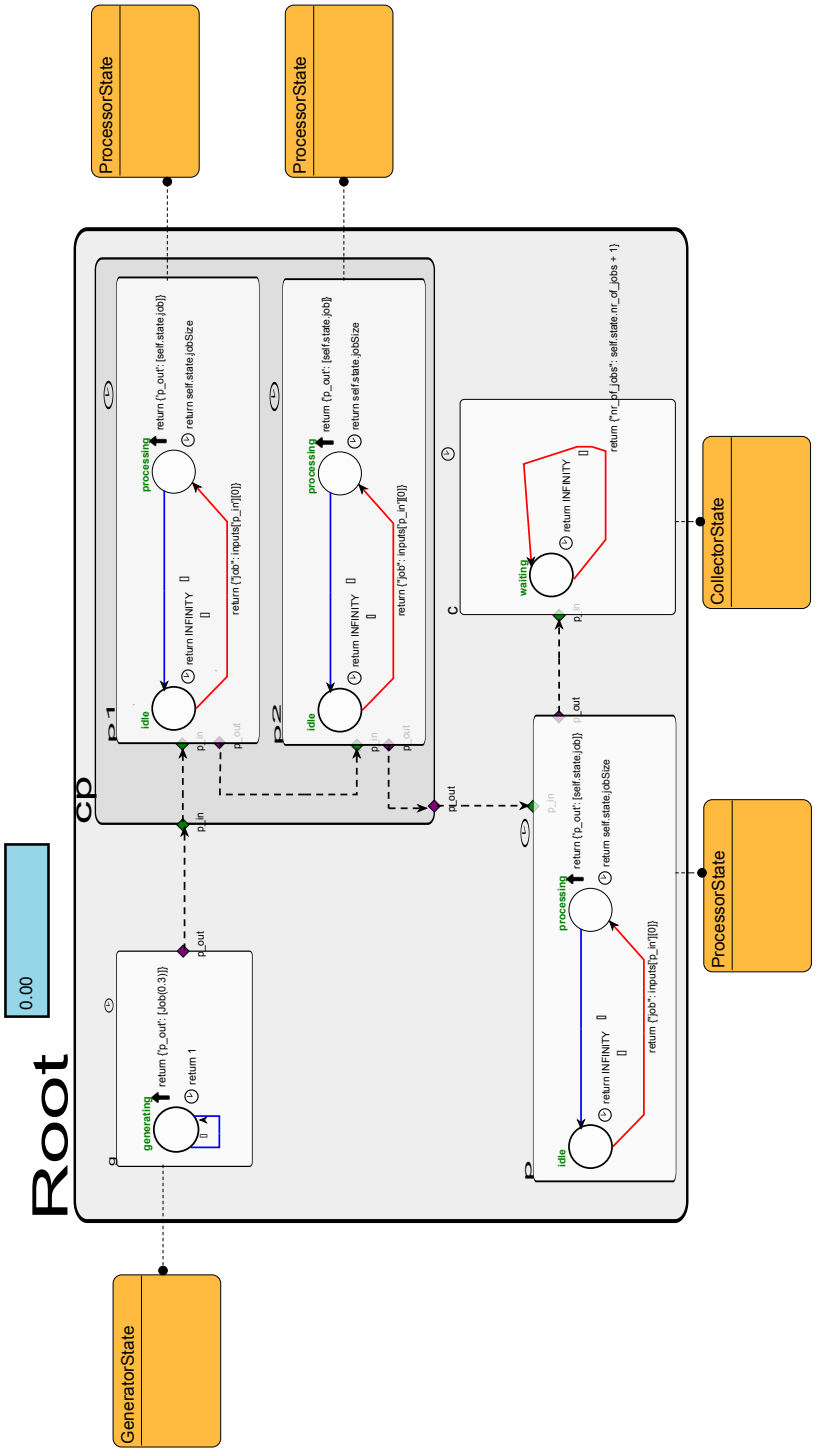


Figure 4.5: The runtime model of the producer-consumer DEVS model.

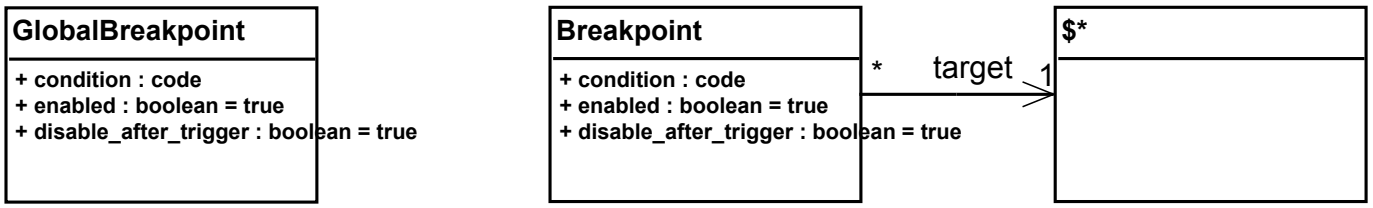


Figure 4.6: The metamodel for the debug language.

Figure 4.7: The debugging toolbar in AToMPM.

PyPDEVS server, which performs the requested action. As a result, a reply is sent back to the PyPDEVS server, where it is translated to certain edit requests of the runtime model to visualize the changes for the user.

The toolbar supports nine operations, explained in the following subsections.

4.4.1 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 PyPDEVS server 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.

4.4.2 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 is twice as fast as wall clock time, etc. During real-time simulation, the model is continuously visually updated, meaning that the user can follow the simulation process.

4.4.3 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.

4.4.4 Big Step

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

4.4.5 Small Step

A big step consists of six distinct phases. A small step allows to visualize these phases, called *small steps*. The list below lists, for each phase, what happens, and how it is visualized.

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.

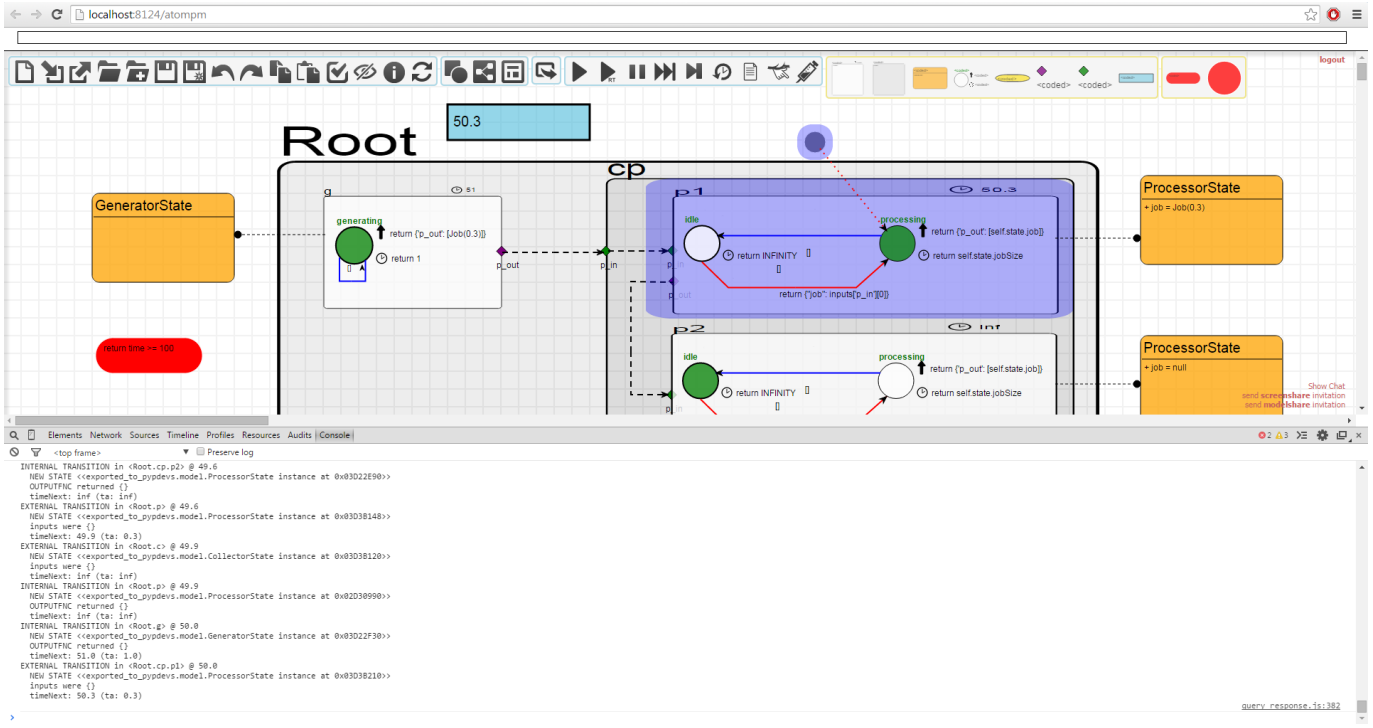


Figure 4.8: Viewing the trace in AToMPM.

To visualize this, an instance of the *EventInstance* class 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 *EventInstance* 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 which 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/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.

4.4.6 Reset

Resets the model and the simulation to their initial state.

4.4.7 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 4.8.

4.4.8 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.

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

5 Modelverse

In the Modelverse, a formalism is modelled for Parallel DEVS. This allows us to export the AToMPM model to a model in the Modelverse. There is a choice here: either we create a language which is the same as the one in AToMPM, allowing a simple 1-to-1 mapping onto models in the Modelverse. The generation of the PyPDEVS code is then more challenging. Another option is to create a language which is more close to PyPDEVS, and allows to exploit the full power of PyPDEVS. The type model in the Modelverse would then be very much like the class diagram of PyPDEVS, without the simulation-specific classes, of course. Classes such as Event and StateDefinition have to be added, as well. The modelled functions conform to the action language, and modelled using the HUTN syntax. The mapping onto PyPDEVS is then facilitated using a code generator.

Of course, the 'proper' way to model DEVS in the Modelverse would be to have this complete language (exploiting the full power of PyPDEVS), and a reduced type model (corresponding to the one in AToMPM). A model transformation can then transform models in the reduced language onto models in the full language. This would allow for other reduced languages, which make different abstractions. For now, though, we will directly generate the models in the full language from the AToMPM model.

6 HUTN for Python PDEVS

Modelverse [24] is a model repository with MvK (Modelverse Kernel) as the backend focusing on scalable and efficient model-management activities. It includes a built-in meta-modelled action language to give basic execution support to back-end operations as well as to allow symbolic transformation and analysis of equations and action code. The AToMPM models are stored in the Modelverse and can then be retrieved and edited by means of a human usable textual notation (HUTN). The HUTN [24] is a textual front-end for the Modelverse. It is a language with high reflexive powers such as the capability of dynamically (in memory) defining grammars and strong type systems (such as metamodels) upon which the syntax of models (such as the ones expressed in the user-defined AToMPM DEVS models) can be checked for syntax errors.

6.1 Modelling DEVS using HUTN

In the Modelverse, two formalisms at two levels of abstraction are used to model Parallel DEVS.

DEVSLang A user-friendly textual concrete syntax that is conceptually similar to the AToMPM language.

DEVSPro A textual notation (using action code only??) closer to the python model based on DEVS theory to allow expert users to exploit the full power of DEVS.

The AToMPM model (along with the included action code) is first exported to DEVSLang. The DEVSLang model is then transformed to a DEVSPro model. The mapping onto PyPDEVS is then facilitated using a code generator. This multi-formalism approach also allows for other reduced languages which make different abstractions. The HUTN models need to be instrumented for visualization (feedback to the AToMPM source model). This also requires the graphical details of the model elements to be stored in the Modelverse and in the Python PDEVS models.

Fig. 6.1 presents the meta-model of the DEVSLang formalism. The action code part of the meta-model is shown in Fig. 6.3. The type model for DEVSPro (shown in Fig. 6.2) is very much like the class diagram of PyPDEVS, without the simulation-specific classes. The *internal transition functions*, *external transition functions*, *confluent functions*, *time advance functions*, *output functions*, and *state definitions* are specified using HUTN action code and need to conform to the action code type model.

6.2 DEVSLang: A Textual Concrete Syntax for Python PDEVS

A textual representation in a neutral language allows models to be easily ported between different simulation environments. Models can be specified in a higher level of abstraction than the python models, and hence is more user-friendly. It is also possible to carry out model transformations at the textual syntax level since the language is meta-modelled. Formal methods can be further used on such textual specifications to analyze the models.

The textual syntax should primarily allow specification of composite structures defined with coupled DEVS, specification of each atomic DEVS component along with its associated time advances, output function, transition functions, transfer functions, confluent functions, and port definitions. The events, state definitions, and simulation constructs should also be included in the textual model.

```
# Concrete syntax to define atomic DEVS components
```

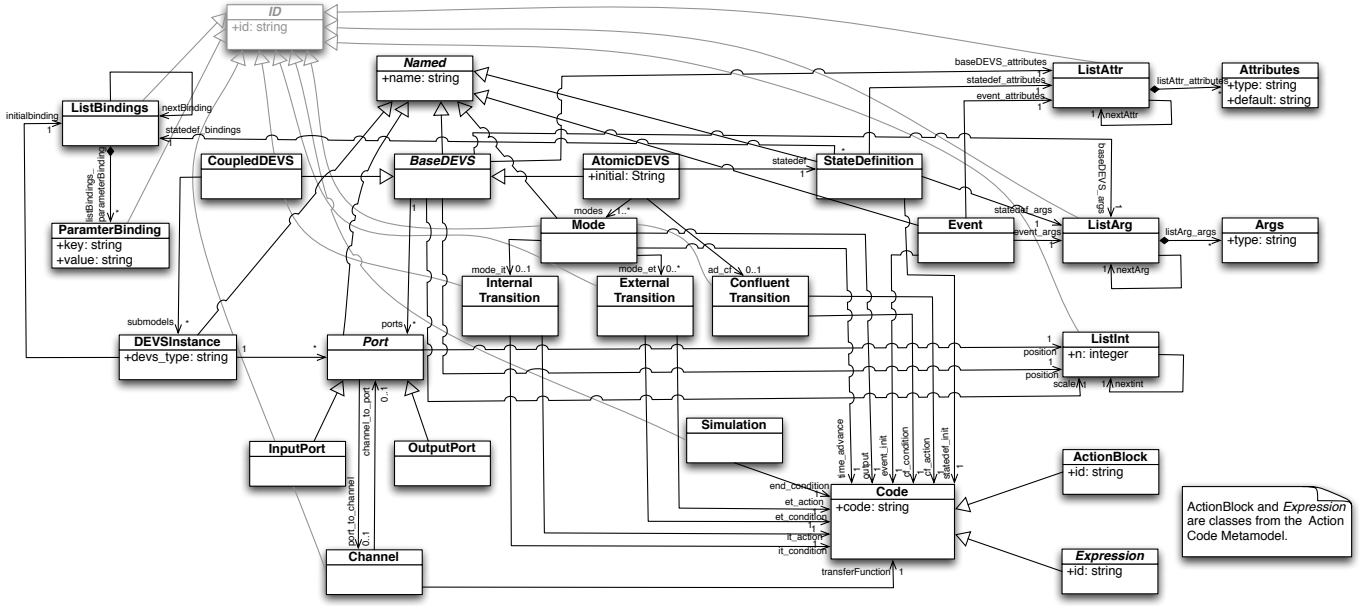


Figure 6.1: DEVSLang Metamodel

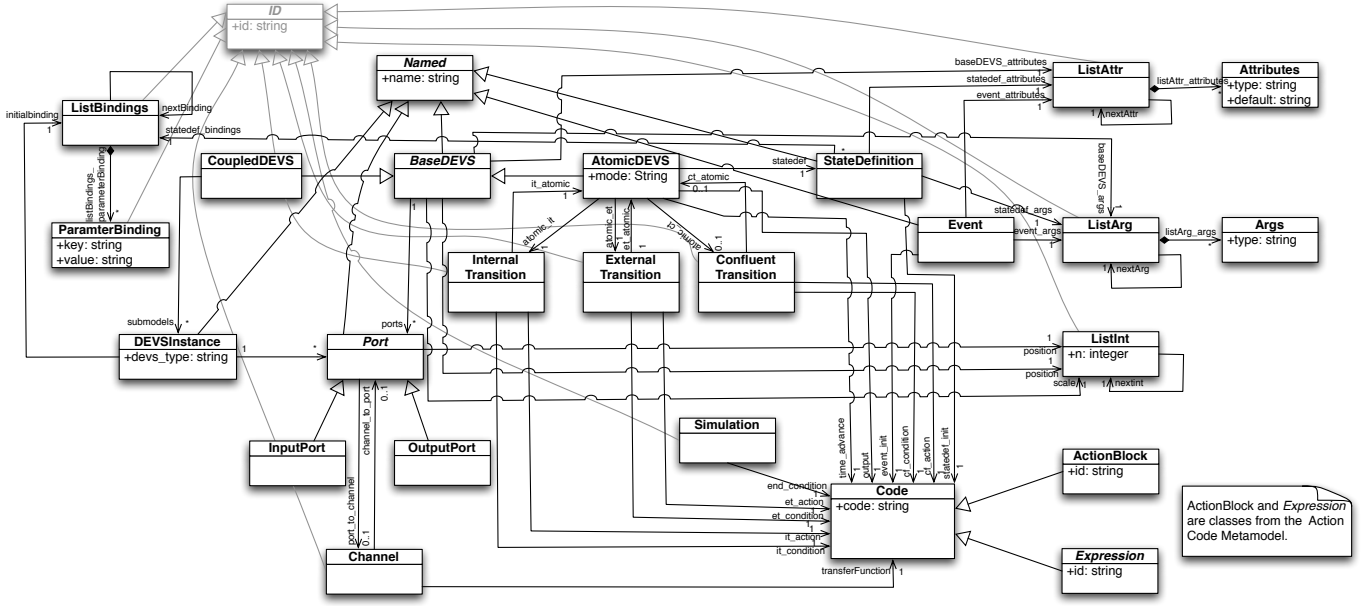


Figure 6.2: DEVSPro Metamodel

```
component AtomicDEVSName():
        #include comma separated list of ports
        inports in1, in2
        outports out1, out2
        # the transition functions associated to an atomic component are defined under ''atomic:''
        atomic:
                ...
                # the initial mode definition starts with ''initial'' followed by the mode name
                initial StateComponentName
                ...

# Concrete syntax to define internal transition functions inside an atomic DEVS
atomic:
        #mode denotes the modal part specification of an atomic component
        #mode corresponds directly to the defined states at the AToMPM level
        mode StateComponentName(arg1, arg2, ...)
                ...
                #time advance begins with the ''after'' keyword
                #the action associated with the internal transition function follows the -> symbol
```

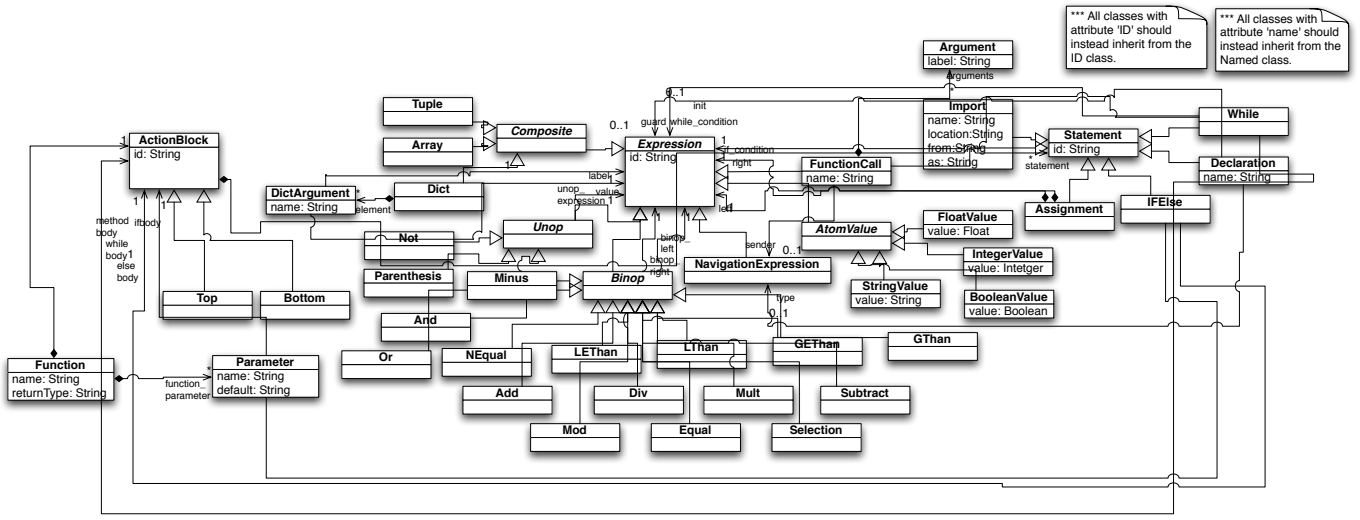


Figure 6.3: HUTN Action Code Metamodel for Python PDEVS (***UPDATE***)

```
            after ta -> TargetStateComponentName(arg1, arg2, ...)
            #output function definition begins the "out" keyword
            out {portname: OutputFunction}
            ...

# Concrete syntax to define external transition functions inside an atomic DEVS
atomic:
        mode StateComponentName(state, time)
            # specify each external transition in a new line
            # external transitions are specified with a condition followed by the action with -> as the separator
            Port[0] == 'interrupt' -> TargetStateDefinitionComponent (state, time+ELAPSED)
            any -> TargetStateComponentName(arg1, arg2, ...)
            ...

# Concrete syntax to define coupled DEVS components
component CoupledDEVSName():
        inports in1, in2, in3 ...
        outports out1, out2, ...
        # specify all DEVS instances inside the coupled DEVS
        x = AtomicDEVSName()
        y = AtomicDEVSName()
        z = AtomicDEVSName()
        # specify all connections between the DEVS instances
        connections:
            from x.in1 to y.in2
            # the action code defined in the transfer function associated to a channel follows the connection definition
            from y.out1 to z.in3 transfer {action code block}

# Concrete syntax to specify events and simulation details
component EventName(parameter1, parameter2, ...)
component Simulation()
```

An exporter in AToMPM generates the textual models from the visual DEVS models. The corresponding DEVSLang model for the Producer-Consumer model (Fig. 4.2) is presented in Listing 6.1.

```
1  ## Producer-Consumer example model in HUTN DEVS concrete syntax
2
3  # State components
4  component GeneratorState(name)
5  component ProcessorState(name, job=None)
6  component CollectorState(name, nr_of_jobs)
7
8  # Data structures
9  component Job(jobSize)
10 component Simulation()
11
12 # Functional components
13 component Generator():
14   outports p_out
15   atomic:
16     mode GeneratorState('generating'):
17       after 1 -> GeneratorState('generating')
18       out {p_out: [Job(0.3)]}
19     initial GeneratorState('generating')
```

```
20
21 component Processor():
22   inports p_in
23   outports p_out
24   atomic:
25     mode ProcessorState('idle', job):
26       any -> ProcessorState('processing', p_in[0])
27       after infinity -> any
28       out nothing
29     mode ProcessorState('processing', job):
30       after job.jobSize -> ProcessorState('idle')
31       out {p_out: [job]}
32     initial Processor('idle')
33
34 component Collector():
35   inports p_in
36   atomic:
37     mode CollectorState('waiting', nr_of_jobs):
38       any -> CollectorState('waiting', nr_of_jobs+1)
39       after infinity -> any
40       out nothing
41     initial CollectorState('waiting')
42
43 component CoupledProcessor():
44   inports p_in
45   outports p_out
46   coupled:
47     p1 = Processor()
48     p2 = Processor()
49   connections:
50     from p_in to p1.p_in
51     from p1.p_out to p2.p_in
52     from p2.p_out to p_out
53
54 component Root():
55   coupled:
56     g = Generator(a,b)
57     cp = CoupledProcessor()
58     p = Processor()
59     c = Collector()
60   connections:
61     from g.p_out to cp.p_in
62     from cp.p_out to p.p_in
63     from p.p_out to c.p_in
```

Listing 6.1: HUTN PDEVS Producer-Consumer Model

6.3 Analysis of DEVS Models

The HUTN provides the ability to define more advanced type-checking mechanisms in order to evaluate the meaning of a given type in a given evaluation context, and henceforth assert its validity (i.e., from a semantic point of view).

We have implemented a parser for the DEVSLang models. The grammar focuses on giving the HUTN users the capability to mentally render the state-machine-like structures visualized on the AToMPM in a simple textual notation. The concept *mode* for instance denotes the modal part specification of an atomic component, which corresponds directly to the defined states at the AToMPM level. The modes are connected by means of (either external or internal) transitions in a grammar-like fashion: each transition is defined by means of a rule where the left-hand side is matching some condition on the component's non-modal state variables, and the right-hand side is always creating a new mode instance while allowing the creation and modification of the non-modal state variables, and passing them through that the new mode instance. The type-checker analyses the action code defined on the left-hand side in order to verify if the non-modal state variables are being just read, which means that a match cannot change the non-modal state variables of a given component, or of any other defined component. Non-modal state variables are naturally defined as typed parameters into the new node.

Bare in mind however, that all of the above described rules in the exporter to the Python PDEVS simulator, are then get compiled into a lower-end language (say Python for instance) as the if-then-else conditions inside the component's internal or external transitions uniquely defined inside a given DEVs component, namely, the match part is compiled into the if condition, and the apply part is compiled in a return call on the *then* body which then returns a new instantiation of the specified mode class.

Notice also that these non-modal state variables can be defined as quantized intermediate states (e.g., quantized from regular continuous functions such as the Integrator) and still in this case, the strong type system check detects any attempt to write on these states. The strong assumptions enforced a-priori by the strong type system checking still allows any DEVS model to be expressed, while maintaining the analysability of the modal part (e.g., by translating it into a timed automaton, which then can be passed on through a specialized analyzer, such as Uppaal), Actually, there is already a lot of work done in automatically computing these quantized intermediate states with minimal (under some specific assumptions on the shapes of the modal part,

as for instance in a loop-connected coupled pair, unguided successive approximations will naturally lead to a complete disaster), and most interestingly by using rational time in order to relate each defined time advance from a given component with the time advance of other components, and ultimately with both the beginning and the end of the simulation. This means that in this case, we can ultimately check for the termination of the simulation (if required) or assure its convergence in a given result class, before converting it into a timed automaton in order to check more general user-defined properties (such as liveness or reachability). The Modelverse's HUTN is still in active development in order to decrease the integration of visual modelling environments, simulators and model checkers, and, as in the DEVSLang, new simulation techniques, in the most convenient fashion and following the principles of correct-by-construction while maintaining the original DEVS expressiveness.

7 PythonPDEVS

PythonPDEVS [25] (a.k.a. PyPDEVS) is a Parallel DEVS [27] language grafted on the Python language, with a matching simulator. Python is a strong typed, interpreted, object-oriented programming language. While PythonPDEVS supports a variety of features (tracing, checkpointing, realtime simulation, ...) and formalisms (Classic DEVS [15], Parallel DEVS [27], Dynamic Structure DEVS [28]), debugging was not yet supported. To simplify our approach, we have limited ourself to a subset of PythonPDEVS, supporting only the Parallel DEVS formalism, without any significant features beyond that.

Section 7.1 will start by introducing the basic constructs of PythonPDEVS with the help of the running example: a producer/consumer model. The functionality of the simulator is presented in Section 7.2. In Section 7.3, we elaborate on the design, discussing both the Class Diagram and the Statechart. Section 7.4 discusses the augmentations that were required to the Statechart to allow for debugging support, along with some implementation details. Finally, the used events (both received and emitted) are discussed in great detail in Section 7.5, which explains the API exposed to the clients.

7.1 Example

```
1  # Import code for DEVS model representation:
2  from DEVS import *
3  from infinity import INFINITY
4
5  class GeneratorState:
6      def __init__(self, name):
7          self.name = name
8
9  class ProcessorState:
10     def __init__(self, name, job=None):
11         self.name = name
12         self.job = job
13
14 class CollectorState:
15     def __init__(self, name, nr_of_jobs=0):
16         self.name = name
17         self.nr_of_jobs = nr_of_jobs
18
19 class Job:
20     def __init__(self, jobSize):
21         self.jobSize = jobSize
22
23 class Generator(AtomicDEVS):
24     def __init__(self):
25         AtomicDEVS.__init__(self, "Generator")
26         self.state = GeneratorState('generating')
27         self.outports = {'p_out': self.addOutPort("p_out")}
28
29     def timeAdvance(self):
30         if self.state.name == 'generating':
```

```
            return 1

    def outputFnc(self):
        if self.state.name == 'generating':
            return {self.outports['p_out']: [Job(0.3)]}

    def intTransition(self):
        if self.state.name == 'generating':
            return GeneratorState('generating')

    def extTransition(self, inputs):
        pass

class Processor(AtomicDEVS):
    def __init__(self):
        AtomicDEVS.__init__(self, "Processor")
        self.state = ProcessorState('idle')
        self.outports = {'p_out': self.addOutPort("p_out")}
        self.inports = {'p_in': self.addInPort("p_in")}

    def timeAdvance(self):
        if self.state.name == 'idle':
            return INFINITY
        elif self.state.name == 'processing':
            return self.state.job.jobSize

    def outputFnc(self):
        if self.state.name == 'idle':
            return {}
        elif self.state.name == 'processing':
            return {self.outports['p_out']: [self.state.job]}

    def intTransition(self):
        if self.state.name == 'processing':
            return ProcessorState('idle')

    def extTransition(self, inputs):
        if self.state.name == 'idle':
            return ProcessorState('processing', job=inputs[self.inports['p_in']][0])

class Collector(AtomicDEVS):
    def __init__(self):
        AtomicDEVS.__init__(self, "Collector")
        self.state = CollectorState('waiting')
        self.inports = {'p_in': self.addInPort("p_in")}

    def timeAdvance(self):
        if self.state.name == 'waiting':
            return INFINITY

    def outputFnc(self):
        if self.state.name == 'waiting':
            return {}

    def intTransition(self):
        pass

    def extTransition(self, inputs):
        if self.state.name == 'waiting':
            return CollectorState('waiting', nr_of_jobs=self.state.nr_of_jobs + 1)

class CoupledProcessor(CoupledDEVS):
    def __init__(self):
        CoupledDEVS.__init__(self, "CoupledProcessor")
        self.outports = {'p_out': self.addOutPort("p_out")}
        self.inports = {'p_in': self.addInPort("p_in")}
        self.submodels = []
        self.submodels.append(self.addSubModel(Processor()))
        self.submodels.append(self.addSubModel(Processor()))
        self.connectPorts(self.inports['p_in'], self.submodels[0].inports['p_in'])
        self.connectPorts(self.submodels[1].outports['p_out'], self.outports['p_out'])
        self.connectPorts(self.submodels[0].outports['p_out'], self.submodels[1].inports['p_in'])
```

```
103
104 class ProduceConsume(CoupledDEVS):
105     def __init__(self):
106         CoupledDEVS.__init__(self, "ProduceConsume")
107         self.outports = {}
108         self.inports = {}
109         self.submodels = []
110         self.submodels.append(self.addSubModel(Generator()))
111         self.submodels.append(self.addSubModel(Processor()))
112         self.submodels.append(self.addSubModel(CoupledProcessor()))
113         self.submodels.append(self.addSubModel(Collector()))
114         self.connectPorts(self.submodels[0].outports['p_out'], self.submodels[1].inports['p_in'])
115         self.connectPorts(self.submodels[1].outports['p_out'], self.submodels[2].inports['p_in'])
116         self.connectPorts(self.submodels[2].outports['p_out'], self.submodels[3].inports['p_in'])
```

Listing 7.1: PyPDEVS Producer/Consumer Model

In Listing 7.1, the producer/consumer model of Figure 4.2 is modelled in the PyPDEVS language. The code generated by the AToMPM formalism will thus look largely similar to this code. Note that it is not exactly equal, as the generated code is a lot more verbose to simplify the exporter code.

As both the original, exclusively coded in Python, PythonPDEVS and the PythonPDEVS modelled in SCCD support the same language, models are interchangable. Therefore, models can be modelled in the AToMPM front-end, and exported to a file on which the user manually invokes PythonPDEVS. The other way around, importing PythonPDEVS code and reconstructing a model in the AToMPM Parallel DEVS formalism is not supported.

7.2 Functionality

Many of the earlier features of PythonPDEVS, as presented in [25], are missing in this simplified version. This section will present the main features that are still supported.

7.2.1 Parallel DEVS

The supported formalism is still Parallel DEVS, in the sense that no shortcuts were taken. For example, transfer functions and user-configurable confluent transitions are fully supported. In this sense, PythonPDEVS is still fully compliant to the Parallel DEVS formalism.

While all required functions can be overridden, all functions have a default defined. These defaults are for a model that passivates immediately, does never output anything, and ignores all external events. For the confluent transition, the internal transition function is invoked before the external transition function. The default functions are thus defined as shown in Listing 7.2. For transfer functions, the default is the identity function.

```
1  def timeAdvance(self):
2      return INFINITY
3
4  def intTransition(self):
5      return self.state
6
7  def extTransition(self, inputs):
8      return self.state
9
10 def outputFnc(self):
11     return {}
12
13 def confTransition(self, inputs):
14     self.state = self.intTransition()
15     self.state = self.extTransition(inputs)
16     return self.state
```

Listing 7.2: PyPDEVS Defaults

Internally, simulation does not happen in parallel, as suggested by the abstract simulator [17]. Even though a parallel implementation is not required by the abstract simulator, we find it important to mention this detail. Simulation is done sequentially for the following reasons:

1. Introducing parallelism introduces more complexity, certainly with the use of debugging.
2. Sequential simulation does not necessarily mean significantly reduced simulation performance, depending on the used programming language and model being simulated [29].

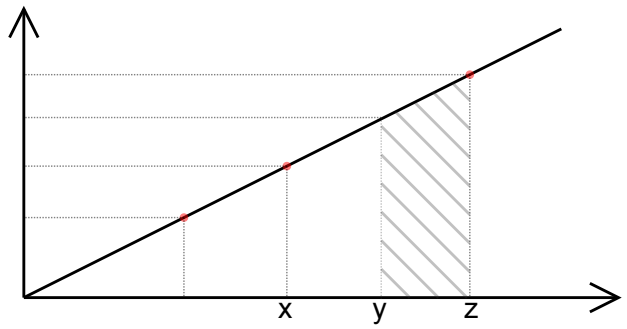


Figure 7.1: Realtime simulation stopping late

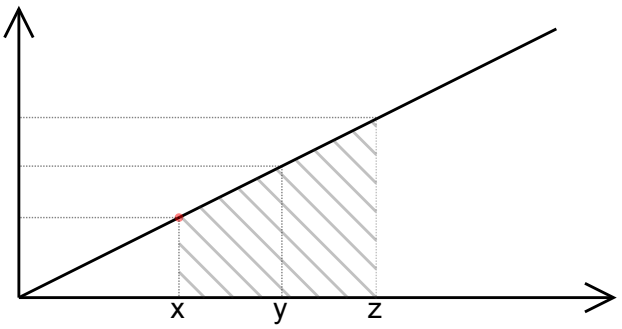


Figure 7.2: Realtime simulation stopping early

7.2.2 Termination Conditions

PythonPDEVS again supports the use of termination conditions. Instead of terminating at a specified time in simulated time, or after a fixed amount of simulation steps, the user has more freedom this way. This freedom comes at the cost of reduced performance, though the gained flexibility is of higher importance in a debugging setting.

Note that the other methods can be encoded in a termination condition, as this has access to the simulated time, the simulated model, and the models that transitioned. Even though it is possible to define a termination condition that never stops the simulation, simulation will always stop as soon as simulation time has reached infinity.

7.2.3 (Scaled) Realtime Simulation

Scaled realtime simulation is supported by PythonPDEVS. While its implementation is fairly trivial using Statecharts, some care has to be taken during the implementation.

Internally, PythonPDEVS keeps the wall clock time at which simulation has commenced. During simulation, the current simulation time is always updated, so that the linear relation remains. The scale is simply used as a factor in this equation.

Similar to the remark in Chapter 1, simulated time progresses differently during a realtime simulation, than during an as-fast-as-possible simulation. The points at which the termination condition is invoked thus becomes a concern. In as-fast-as-possible simulation, the termination condition is only invoked at the start of a simulation step, with the time being the time at which the transitions should happen. Realtime simulation can not take this same approach, as shown in Figure 7.1 and Figure 7.2.

In Figure 7.1, the actual termination of the simulation only happens at wall clock time y, whereas it was requested at time z. The cause for this is simple: termination conditions are only executed right before the start of a simulation step. As the function contains arbitrary Python code, the simulator has no way of knowing at which time it should terminate. The result is that simulation will first wait up to wall clock time y, where it is detected that $y > z$ and that simulation should terminate. However, simulation now took y seconds, instead of z (as requested). This was not a problem in as-fast-as-possible simulation, as the jump from simulated time x to y happens instantaneously. There was (almost) no wall clock time in between these two points in simulated time, making the difference unnoticable. Computing the termination condition at the end of a simulation step is of no help either, as this would finish simulation at time x instead of time z.

Figure 7.2 presents a different problem, with the same problem at the root. The last transition of this model happens at time x, whereas simulation is requested to terminate at time z. As the last transition of the model happened at time x, the model passivates afterwards, meaning that the next point in simulated time would be infinity. Detecting this, an as-fast-as-possible simulation would terminate, as simulation of the next step is impossible. While no simulation results are lost, simulation does not take as long as requested (x seconds instead of z). But more importantly, the user did not have the possibility to produce interrupts in the interval $]x,z]$, for example at time y. If z were to be infinity, the model would simply have to wait for input from the user infinitely long, instead of stopping immediately. The cause is thus again the gap between transitions, in which the user can generate input, which was not present in as-fast-as-possible simulation.

Our solution in PythonPDEVS for this problem is simple: the termination condition is invoked at the start of every simulation step. However, the time to wait will be cut of by using a minimum with a specific threshold (e.g. $100ms$). After that time has passed, simulation time is updated and the termination condition is invoked again. If simulated time has progressed sufficiently far, a simulation step will be performed.

This method is not foolproof (e.g. it is possible to terminate in an interval $[x,y]$ with $x - y < 100ms$, causing termination to be indeterministically detected). In case this would be a problem to the user, the user is encouraged to decrease the threshold to allow more frequent termination check invocations. This approach more accurately resembles the behaviour that the user expects: termination requested at time x will occur somewhere in the interval $]x, x + 100ms]$.

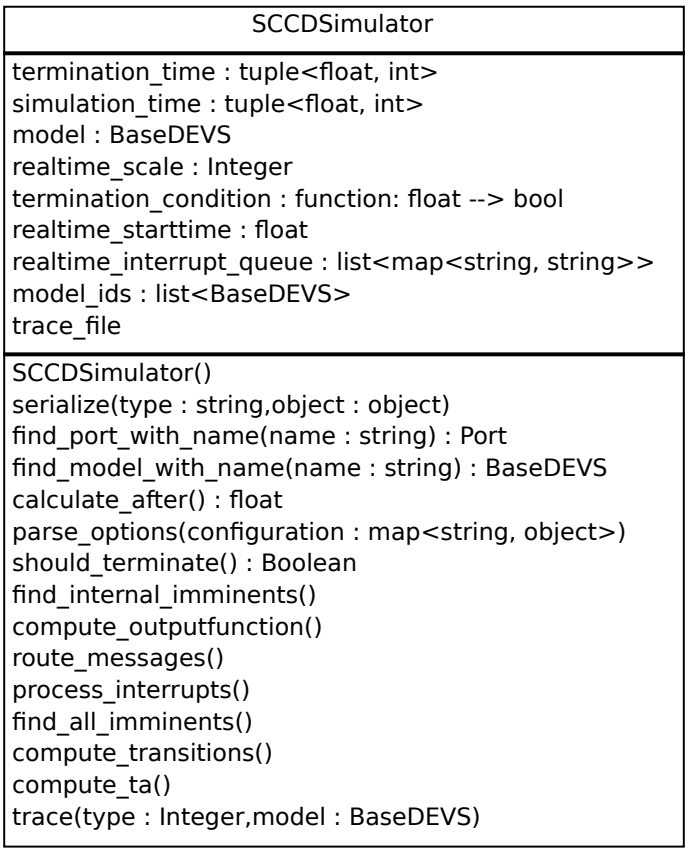


Figure 7.3: PythonPDEVS Simulator Class

7.3 Design

To facilitate the addition of debugging functionality later on, the modal part of PythonPDEVS is explicitly modelled in the SCCD [30] formalism. SCCD is a combination of Class Diagrams and Statecharts, where the Statechart part can dynamically instantiate models, each modelled with a Statechart. The additional functionality introduced by the Class Diagrams part was not necessary for our goalds, and will not be elaborated on.

Afterwards, the modal part of the simulator was lifted out and modelled in Statecharts, with future debugging extensions in mind. The Class Diagram part was already present in the original PythonPDEVS and was therefore copied and slightly updated.

This section will first present the Class Diagram and Statechart of the original simulator, thus without debugging functionality. Afterwards, debugging features are added to the statechart.

7.3.1 Class Diagram

Simulator

The class diagram of the PythonPDEVS simulator is shown in Figure 7.3. The PythonPDEVS simulator is responsible for the configuration and management of the simulation. All behaviour of the simulation is therefore present in this class. Consequently, only the Simulator has an associated statechart.

Models

Models in PythonPDEVS are modelled by subclassing the *AtomicDEVS* and *CoupledDEVS* classes. A number of abstract functions have to be implemented in these models:

- **extTransition** receives a dictionary of inputs which maps names of input ports onto bags (lists) of inputs received on that port. It returns the new state of the model after processing these inputs.
- **intTransition** calculates the new state of the model after the time has advances as specified in the time advance function.
- **confTransition** decides what happens when there is a conflict between an external and an internal transition.
- **outputFnc** returns a dictionary which maps names of output ports onto bags (lists) of outputs which need to be generated

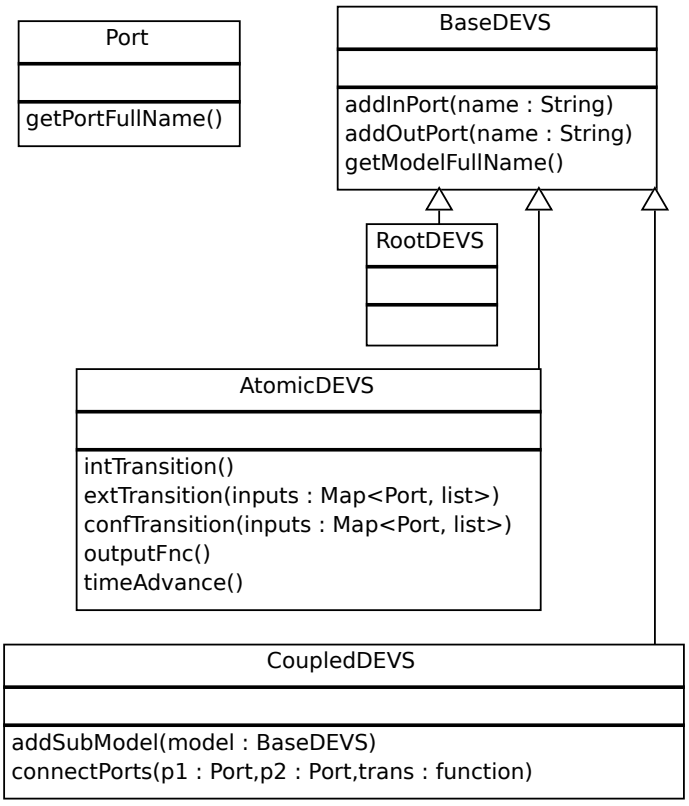


Figure 7.4: PythonPDEVS Model Classes

just before the internal transition function is executed.

- **timeAdvance** returns the amount of time the model waits in the current state before executing the internal transition function (either a float or INFINITY).

Ports (input and output) are added by the functions **addInPort** and **addOutPort**. Adding a submodel to a CoupledDEVS instance is added using the **addSubModel** function, and ports are connected using **connectPorts**.

7.3.2 Statechart

Figure 7.5 presents the Statechart of the modal part of the PythonPDEVS simulator, without any debugging functionality. This section will elaborate on the different components that build up the statechart.

Termination Check

The termination check component, identified in the Statechart by the *check_termination* composite state, is responsible for detecting whether or not to continue simulation. This is split up into a different component to make realtime simulation slightly easier. As the termination condition might be dependent on time, the termination condition will be executed frequently.

A *workaround* state is added, transitioning to the *check_termination* state, as the default semantics of the Statechart is *Take Many* [31], while we actually want *Take One*.

Simulation

Simulation itself is done by the *do_simulation* composite state. For clarity, all different stages of Parallel DEVS simulation are represented seperately. As a single simulation step is non-interruptable in the basic case, there is an always-enabled transition between the different states. There is a check for whether or not the Statechart is in simulation mode, as otherwise simulation should not happen. This is more of a safety measure, as the termination condition should prevent a simulation from starting while in the *paused* state.

Simulation State

Without support for debugging, the state of the simulation is quite simple: either it is waiting to start simulation (*paused*), it is simulating (*continuous*), or it is simulating in realtime (*realtime*). As there is no support for pausing of a simulation, the only

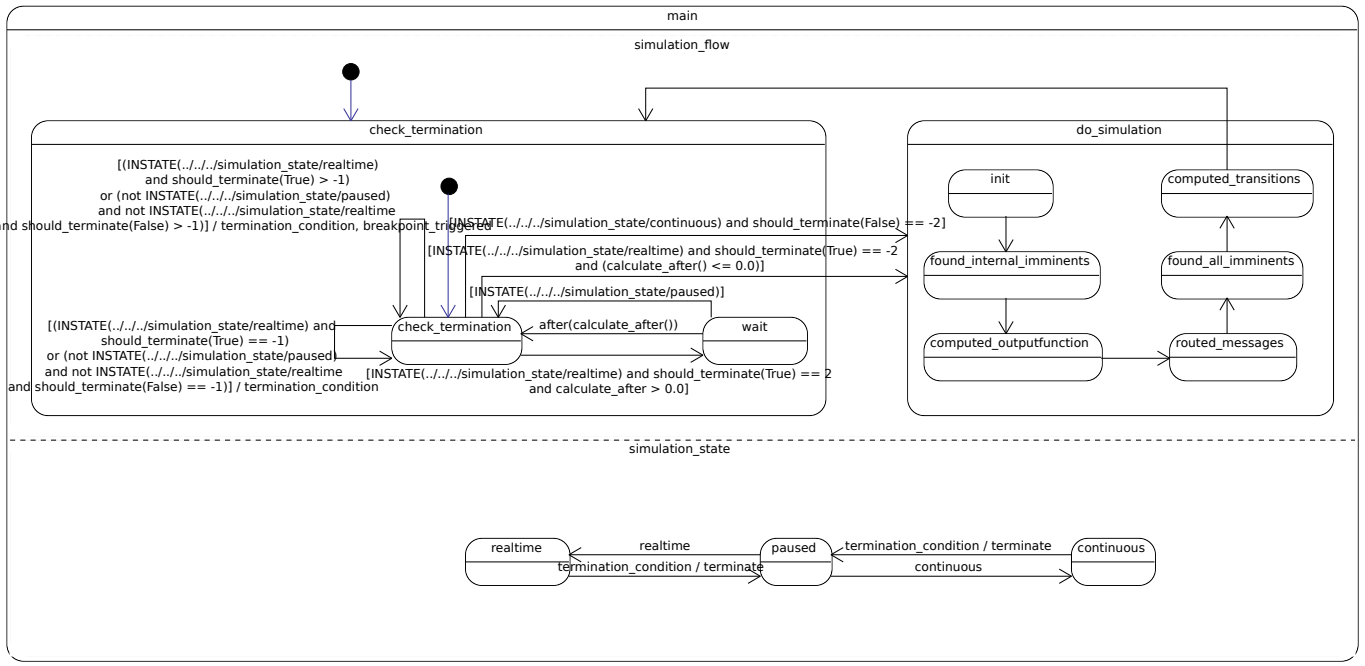


Figure 7.5: PythonPDEVS Statechart

way to return to the *paused* state is by waiting for a termination condition to trigger the end of the simulation.

7.4 Debugging Extensions

The statechart from Figure 7.5 is extended to allow for debugging operations, as seen in Figure 7.6. For each debugging function that is supported, some explanation is given about how this influenced the Statechart and our considerations. No changes are made to the Class Diagram, so we do not elaborate on this.

7.4.1 Pause

Pausing a simulation is fairly simple thanks to Statecharts. We distinguish between pausing 2 types of simulation: as-fast-as-possible and (scaled) realtime simulation.

In as-fast-as-possible simulation, pausing can be achieved by forcing the termination condition to true. That way, the current simulation step will be finishe, ensuring that the simulator is in a consistent state. After the termination condition is evaluated, simulation will automatically terminate. For as-fast-as-possible, we are limitted to pausing at simulation times at which model transitions occur. As a result, a pause issued at simulation time x will stop simulation in the interval $[x, \infty[$, where the next transition occurs.

For realtime simulation, we take mostly the same approach by forcing the termination condition to true. In combination with the polling discussed in Section 7.2.3, this will cause simulation to pause somewhere in the interval $[x, x+100ms]$. The main complexity arises when resuming simulation, possibly with a different scale. This is solved by resetting the realtime starting point, as if the complete simulation up till now was unpaused and at the requested scale.

This behaviour is visualized in Figure 7.7. At the start, the start time of the simulation is saved as $start$. Simulation runs in realtime with a scale of $scale$, up to wall clock time x. Simulated time at that point is thus given by $(x - start) * scale$. After a certain delay, at wall clock time y, realtime simulation is started with a scale of $scale$. The $start$ value now needs to be reset, as the simulator will pace itself by allowing all transitions to happen up to simulated time $(y - start) * scale$. To do this, $start$ is set to $y - (simulated\_time/scale)$.

7.4.2 Big Step

For the simulator, big steps simply means that it should simulate until the termination condition check is reached. So for the simulation state, this can become a new state of simulation (big_step). As soon as the termination condition check is invoked, an event will be raised. This event is used to signal the end of the big step, allowing the simulation state to go back to the paused state.

To allow for visualization in the front-end in AToMPM, the current state of the simulation is output at the end of the big step.

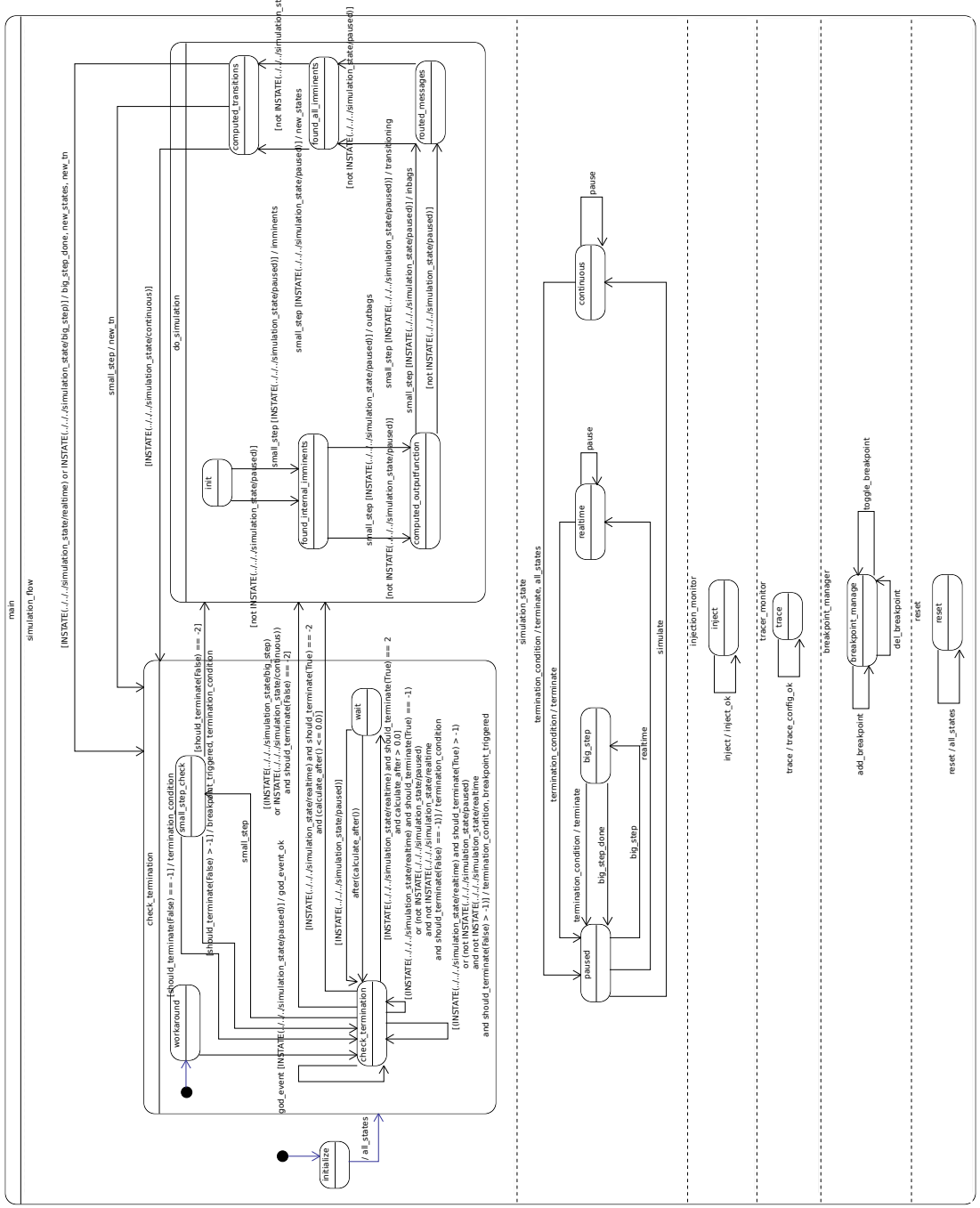


Figure 7.6: PythonPDEVS Statechart with debugging functionality

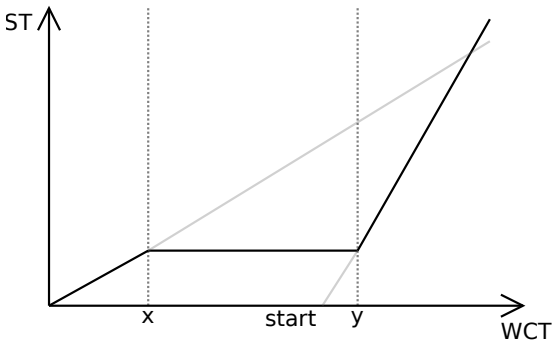


Figure 7.7: Pausing scaled realtime simulation in PythonPDEVS

7.4.3 Small Step

Small steps are very different from big steps in terms of the statechart. A small step is defined as a single step in the *do_simulation* state. Therefore, there should be no 'memory' of the small steps anywhere and consequently there is no specific simulation state for this mode. Simulation thus remains in the *paused* state.

A single step should be performed though, so every step in the *do_simulation* composite state gets another transition. The newly added transition has the same target as the transition that was already present, but is triggered by the *small_step* event. For visualization, the processed values are output at the end of the transition.

7.4.4 Breakpoints

Breakpoints are modelled as a different kind of termination condition. Whereas the termination condition signals the normal end of a simulation, breakpoints are caused by a specific cause. If a breakpoint is triggered, an event will be raised that contains the identifier of the breakpoint.

The AToMPM front-end has support for both global (model-wide) and local (entering a specific state) breakpoints. This distinction is not made in the simulator, as all can be written as a global breakpoint. The front-end therefore has to translate the breakpoint to a global one.

A breakpoint can have a condition, which has access to the following parameters:

1. **time**: the current simulation time, but before processing the transitions at that time.
2. **model**: the complete model being simulated, with all states filled in with their current values.
3. **transitioned**: a set of all models that transitioned in the previous step.

Apart from the obvious simulation time and simulation model, the set of transitioned models is required to make local breakpoints work. When a breakpoint is triggered for a model entering a state, simulation will terminate. But when simulation is resumed, the breakpoint should only trigger again when the state is *entered* again. It is therefore not possible to check the state of the model and trigger if it is in a breakpoint state. The set of transitioned models solves this problem, as local breakpoints will only check for the state if the model has recently transitioned. If the model has not transitioned, it is still in the same state, and therefore the breakpoint should never trigger.

Note that termination conditions are always evaluated before processing any breakpoints.

7.4.5 Event Injection

Event injection is the artificial creation of events, with the purpose of checking the behaviour of a model upon receiving a certain event.

Event injection is only allowed to happen at certain points in the simulation. Because events might happen during any point in the simulation though, an orthogonal component was added that receives these events. When it receives the event, it puts it in a queue. That queue is checked at the start of a big step and all messages in it are processed at once.

Processing event injection is simple, as the event only has to be added to the set of bags that are already kept in between little steps. For maximal flexibility, events can be injected at every input port, even internal ports. While this breaks modularity, it allows for more precise control over the model.

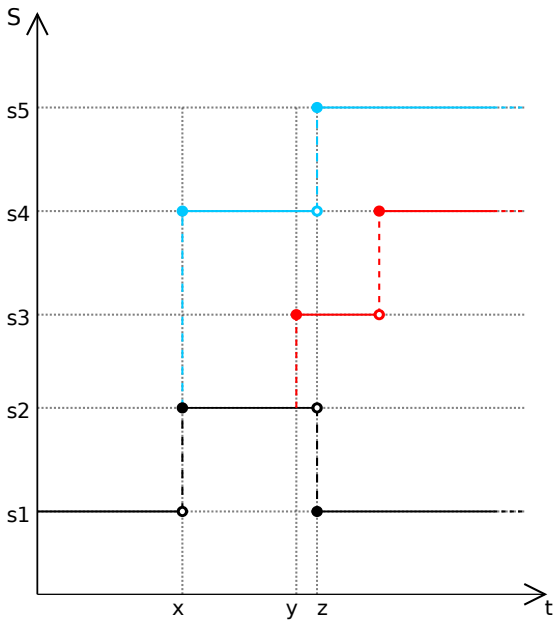


Figure 7.8: God event processing in PythonPDEVS

7.4.6 God Events

While event injection tests the behaviour of a model upon receiving an event, god events make it possible to alter the state of a model. For the simulator, such god events are very simple to process, as the state only has to be updated. However, state changes might have drastic changes to the simulation. For example with timing: when the god event is processed, the state changes, but consequently the *time advance* is altered too.

There are several different paths that can be taken, as to how the model will respond to this:

1. The god event is processed just like an external transition, but with a *modification* message. This approach has the advantage that it fits nicely in the semantics of DEVS, as it is mapped to an external transition. It can become problematic though, as the transition will also cause a new time advance to be computed, starting from the time of modification. If the time of modification is not as desired, the resulting time next will be wrong too, possibly resulting in problematic simulation. Another important disadvantage is that the user does not have the chance to perform any updates himself. It is common practice in DEVS models to keep the total simulation time locally (e.g. for performance metrics). As DEVS models do not have access to this time, some lines of code are added to the internal, external and confluent transition. By defining a new kind of transition, this approach becomes impossible, as the gap between the last transition and the god event is not taken into account. While this could be solved by altering the internal simulation time, using a god event, this does not seem elegant to us.

2. The god event is processed as if the previous state transition (either internal, external or confluent) did not result in that state, but in a different state. The past is thus modified, to retrofit the new state. Because the old state is modified, its time advance has to be recalculated. Caution is required when recalculating the state, as it needs to be based on the time of the previous transition. While this approach seems the most elegant to us, it creates a grey zone: a model might receive a god event at time x, while its last transition happened at time y ($y < x$). The god event being executed means that $x - y < ta$. But after the state is updated, the ta will have changed too. A problem would arise if $ta < x - y$, as then a transition would have happened before the god event was executed. This is clearly a problematic situation, so we require that the new state returns a time advance for which $x - y < ta$ holds.

Both methods are shown in Figure 7.8. At simulation time y, a god event is processed for this model. The first approach will transition at time y, whereas the second approach will transition at time x.

We have opted to use the second approach. To implement it, the model's `timeNext` has to be set to $timeLast + timeAdvance(state)$.

7.4.7 Tracing

Tracing support is easily implemented by writing out some information to a file after performing a model transition. To keep the front-end from being flooded by tracing information, information is written to a file that can be read on-demand by the front-end.

Because tracing has a performance impact, tracing is disabled by default in the statechart. For the AToMPM front-end, tracing will always be enabled in the background, as the file could be requested at any time.

7.4.8 Reset

Resetting a simulation requires the original model to be saved internally, because simulation will alter the model in-place. For technical reasons, no complete copy of the model can be made, which is restored during a reset. We therefore only take a copy of the states of the model, limitting our approach to static models (as required in Parallel DEVS anyway).

The simulation time, the trace file, and scheduler have to be reset too. Note that breakpoints are not reset, as they are managed by the user.

7.5 Clients

As the SCCD model of PythonPDEVS now simply listens to events, through a very small Python wrapper, a multitude of clients can be written that communicate with the statechart. The statechart listens to the following events:

- **simulate**: simulate until the termination condition is true, or the termination time is reached. A breakpoint evaluating to true will also trigger the end of the simulation.
- **big_step**: simulate until the termination check is reached.
- **realtime**: simulate until the termination condition is true, or the termination time is reached, or a breakpoint evaluates to true. Simulation is performed in realtime, after applying a certain scale factor. A scale factor of x means that simulation time will progress x times faster than the wall clock time.
- **small_step**: simulate a single internal simulation step. Termination and breakpoints are only evaluated at the termination check phase in the simulation.
- **god_event**: modify the internal state of the selected model. States can only be updated by updating their attributes. The event is only processed as soon as simulation has reached the termination check.
- **inject**: put a user-defined message on an input port anywhere in the model. The event is only processed as soon as simulation has reached the termination check.
- **trace**: write out trace information to a file.
- **pause**: pause simulation, by forcing the termination check to evaluate to true. The pause will be processed as soon as possible, but not in an inconsistent state (i.e. during a simulation big step).
- **add_breakpoint**: add a breakpoint to be checked at every simulation step. An identifier is passed too, which will be used to identify the breakpoint in case it triggers. Changes take effect at the end of the current big step.
- **del_breakpoint**: deletes a breakpoint. This has the same effect as disabling the breakpoint, but it cannot be reenabled afterwards. Changes take effect at the end of the current big step.
- **toggle_breakpoint**: sets the 'disabled' state of the breakpoint. Changes take effect at the end of the current big step.
- **reset**: reset the simulation as if no simulation has happened. This will reset the simulation time to 0.0 and reset all states to their initial state.

The following events are raised by the statechart:

- **inject_ok**: indicates that the inject event was processed.
- **trace_config_ok**: indicates that the trace file configuration was processed and all future transitions will be logged.
- **all_states**: contains a copy of the simulated model at the current point in time.
- **breakpoint_triggered**: indicates that a breakpoint has triggered. The ID of the breakpoint is contained in the event.
- **god_event_ok**: indicates that the god event was processed.
- **imminents**: contains a set of all imminent models.
- **outbags**: contains a set of bags that were output by all imminent models.
- **inbags**: contains a set of bags that were transfered and routed to their destination. All injected events are also present in this set.
- **transitioning**: contains a set of all models that are transitioning, together with the kind of transition (*internal*, *external*, or *confluent*).
- **new_states**: contains a set of all new states after performing transitions.
- **new_tn**: contains a set of all new `timeNext` values of the models that transitioned.
- **terminate**: indicates that simulation has finished, either by reaching a termination condition, triggering a breakpoint, or a pause event being processed.

7.5.1 Command line

A simple command line interface for PythonPDEVS is written. The interface is simply a read from standard input, which will transform the textual input into an event. That event is asynchronously inserted into the running statechart. It is therefore possible to insert new events, while previous events are still being processed.

The command line simply maps between textual input and events, so it is not very usable. It mainly serves as a reference implementation for the simulator, but also as an AToMPM-independent front-end to the simulator.

7.5.2 AToMPM

The AToMPM front-end, which uses this exact same interface, is discussed in detail in Section 4.

8 PythonPDEVS Server

The PythonPDEVS server is responsible for accepting HTTP requests coming from the front-end (AToMPM) and sending them to the PythonPDEVS simulator. Responses coming back from the simulator need to be translated into Create, Read, Update, and Delete (CRUD) requests which edit the runtime model to reflect the update of the simulation.

The supported HTTP requests are listed below.

- **pdevssimulate**: unhighlights all highlighted elements, and sends a *simulate* request to PyPDEVS. The termination condition is retrieved from the experiment file which was exported from the design model.
- **pdevssimulatert**: unhighlights all highlighted elements, and sends a *realtime* request to PyPDEVS. The termination condition is retrieved from the experiment file which was exported from the design model.
- **pdevspause**: unhighlights all highlighted elements, and sends a *pause* request to PyPDEVS.
- **pdevsbigstep**: unhighlights all highlighted elements, and sends a *big_step* request to PyPDEVS. The termination condition is retrieved from the experiment file which was exported from the design model.
- **pdevssmallstep**: unhighlights all highlighted elements, and sends a *small_step* request to PyPDEVS. The termination condition is retrieved from the experiment file which was exported from the design model.
- **pdevsreset**: unhighlights all highlighted elements, and sends a *reset* request to PyPDEVS.
- **pdevsshowtrace**: sends a request to the front-end to display the contents of the *trace.txt* file in the console.
- **pdevsgodevent**: if the changed attribute is *name*, sends a **UPDATE** request to the front-end to change the current state to the state with the new value for the *name* attribute, otherwise sends a **UPDATE** request to change the *attribute_values* attribute. Note that attribute values are evaluated, to allow for class instantiations. At the end, sends a *god_event* to PyPDEVS.
- **pdevsinject**: sends an *inject* request to PyPDEVS for all messages created by the user that were not injected before.

The PythonPDEVS server also listens to output events of the PythonPDEVS simulator, which denote a change in simulation state. All events, and how the server translates these into CRUD events for the front-end, is listed below.

- **all_states** or **new_states**:
  1. Removes all *EventInstance*s by sending **DELETE** requests.
  2. Sets the current simulation time by sending an **UPDATE** request.
  3. For all Atomic DEVS models, sets the value of the *time_next* attribute, as well as the current state and its attribute values by sending **UPDATE** requests.
- **imminents**:
  1. Sets the current simulation time by sending an **UPDATE** request.
  2. For all imminent Atomic DEVS models, send a request to the front-end to highlight it in blue.
- **inbags** or **outbags**:
  1. Sets the current simulation time by sending an **UPDATE** request.

2. For all port locations where a message is generated or received, send a **CREATE** request to the front-end to instantiate an *EventInstance* on the location of the port with the correct attribute values.

- **transitioning**:
  1. Sets the current simulation time by sending an **UPDATE** request.
  2. For all transitioning Atomic DEVS models, highlight them in a color depending on which transition function is used: blue for internal, red for external, and purple for confluent.
- **new_tn**:
  1. Sets the current simulation time by sending an **UPDATE** request.
  2. For all Atomic DEVS models whose time next was recalculated, set their *time_next* attribute by sending an **UPDATE** request.
- **breakpoint_triggered**: highlight the triggered breakpoint in blue by sending a request to the front-end.
- **terminate**: update the current simulation time by sending an **UPDATE** request.
- **inject_ok**: print a message on the console that the event injection was successful.
- **god_event_ok**: print a message on the console that the god event was successful.

The PythonPDEVS server also listens to CRUD events generated by the user and reacts accordingly, as listed below.

- **CREATE** a *GlobalBreakpoint*: saves the breakpoint id and sends a *add_breakpoint* event to PyPDEVS.
- **CREATE** a *Breakpoint*: saves the breakpoint id.
- **CREATE** a *target* relation: sends a *add_breakpoint* event to PyPDEVS. The breakpoint function is modified to only trigger the breakpoint when the state it is connected to was entered.
- **CREATE** a *EventInstance*: saves the event instance for later, when the *pdevsinject* request is received.
- **UPDATE** a *GlobalBreakpoint* or *Breakpoint*: if the *enabled* attribute was changed, sends a *toggle_bp* event to PyPDEVS. Otherwise, sends a *del_breakpoint* event, followed by a *add_breakpoint* event, with the adjusted attribute values.
- **DELETE** a *GlobalBreakpoint* or *Breakpoint*: sends a *del_breakpoint* event to PyPDEVS.

Bibliography

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

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

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

[4] 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.

[5] Peter Fritzson and Peter Bunus. Modelica - a general object-oriented language for continuous and discrete-event system modeling and simulation. In *Simulation Symposium*, pages 365–380, 2002.

[6] Martin Sjölund and Peter Fritzson. Debugging Symbolic Transformations in Equation Systems. In *EOOLT, Linköping Electronic Conference Proceedings vol. 56*, pages 67–74, 2011.

[7] Martin Sjölund, Francesco Casella, Adrian Pop, Adeel Asghar, Peter Fritzson, Willi Braun, Lennart Ochel, Bernhard Bachmann, and Politecnico Milano. Integrated Debugging of Equation-Based Models. In *Proceedings of the 10th International ModelicaConference*, pages 195–204, 2014.

[8] 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.

[9] Adeel Asghar, Adrian Pop, Martin Sjölund, and Peter Fritzson. Efficient Debugging of Large Algorithmic Modelica Applications. In *Proceedings of MATHMOD 2012 - 7th Vienna International Conference on Mathematical Modelling*, 2012.

[10] 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.

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

[12] François E. Cellier. *Continuous system modeling*. Springer-Verlag, New York, 1991.

[13] Hans Vangheluwe, Daniel Riegelhaupt, Sadaf Mustafiz, Joachim Denil, and Simon Van Mierlo. 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, pages 379 – 386. Society for Computer Simulation International, 2014.

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

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

[16] Hans Vangheluwe. The Discrete EVent System specification (DEVS) formalism.

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

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

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

[25] 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.

[26] Yentl Van Tendeloo and Hans Vangheluwe. The modular architecture of the python(p)devs simulation kernel: work in progress paper. In *2014 Spring Simulation Multiconference, SpringSim '14, Tampa, FL, USA, April 13-16, 2014, Proceedings of the Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium*, page 14, 2014.

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

[28] Fernando J. Barros. Dynamic structure discrete event system specification: a new formalism for dynamic structure modeling and simulation. In *Proceedings of the 27th conference on Winter simulation*, pages 781–785, 1995.

[29] Jan Himmelspach and Adelinde M Uhrmacher. Sequential processing of PDEVS models. In Agostino G. Bruzzone, Antoni Guasch, Miquel Angel Piera, and Jerzy Rozenblit, editors, *Proceedings of the 3rd EMSS*, pages 239–244, Barcelona, Spain, 2006. LogiSim. ISBN 84-690-0726-2.

[30] Glenn De Jonghe and Hans Vangheluwe. Statecharts and Class Diagram XML - A general-purpose textual modelling formalism. Technical report, University of Antwerp, 2014.

[31] Shahram Esmaeilsabzali, Nancy a. Day, Joanne M. Atlee, and Jianwei Niu. Deconstructing the semantics of big-step modelling languages. *Requirements Engineering*, 15(2):235–265, April 2010.