Modelling the DEVS-formalism

Yentl Van Tendeloo
University of Antwerp
Yentl.VanTendeloo@student.ua.ac.be

Abstract

We propose a three-step approach to allow (efficient) simulation of DEVS models, written in a graphical environment (built in $ATоM^3$). Our approach is simulator and language independent as it uses Modelica as an intermediate neutral language. To show that this approach works, we apply it to a basic example that will be simulated in the PythonDEVS simulator.

Keywords: DEVS, $ATоM^3$, PythonDEVS, Modelica, µModelica, Symbolic flattening

1. Introduction

The Discrete EVent system Specification formalism is widely used in modelling and simulation and is supported by many simulators. The formalism only presents an *abstract simulator*, so the models can be written in many different (programming) languages. This presents a problem, as it causes many different models, all of which are written for a different simulator. Depending on which simulator is used, an other programming language and API has to be used. This prevents portability and comparisons between simulators. Furthermore, the performance of the simulation is highly dependent on the used simulator, restricting the choice between simulators for many high-performance simulations.
From another side, models must be specified textually in most popular DEVS simulators (PythonDEVS, VLE, DEVS-Suite, CD++, X-S-Y, adevs, ...) which reduces understandability of the specified model and therefore introducing difficult to spot bugs. Though some simulators incorporate visual modelling to a certain extent, like VLE which allows the user to model the *coupled model* graphically. However, the *atomic model* still has to be hand-coded textually. Even if one such simulator would allow graphical modelling, it would likely be in a format tailored to that specific simulator,

as is the case in VLE. This would aggrevate the issues presented previously concerning portability, as the complete model will have to be rewritten. Textual models are still somewhat reusable to some extent (e.g. only change the used API or some language constructs), while this is not the case for graphical languages.
It is mentionable that DEVS simulators should not really have to focus on how the modelling is done. So it might be interesting to shift the modelling part out of the simulator program and make a seperate graphical modelling environment which can export it's graphical model as a textual model, compliant to the specified simulator's API and programming language. As these components are now split up, we can create a simulator-independent graphical modelling environment which generates an intermediate model that can optionally be optimized symbolically.
We will therefore use a three-step approach:

1. Model the DEVS model conform to a metamodel in $AToM^3$, to provide a user-friendly and (mostly) graphical modelling environment that generates Modelica code and is simulator independent
2. Flatten the generated Modelica code, to offer (possibly) increased performance
3. Compile the flattened Modelica code for a certain simulator

The rest of this paper is organized as follows: Section 2 quickly describes the relevant parts of the DEVS formalism, section 3 describes ways to use Modelica for DEVS models. In section 4 we will give a more global overview of where this approach fits in to the modelling process. Section 5 will present the first step of our approach and present the meta-model for the DEVS formalism in $AToM^3$. Section 6 presents the second part, which is the flattening phase. Section 7 describes the compilation phase to PythonDEVS. In section 8, a small example of a simple queueing system will be given. The performance difference of the flattening phase will be shown in section 9. Finally, section 10 gives related work, section 11 presents some possible future work and section 12 concludes.

2. DEVS Formalism

For those unfamiliar to the DEVS formalism, we refer to Bernard P. Zeigler and Kim (2000). We will mainly be using the *closure under coupling* proof, which proves that every *coupled model* can be rewritten as a behaviourally equivalent *atomic model*. The proof shows a way to encode all

actions that have to be done by the coupled model - like message passing, searching the next model to transition, ... - in the form of an atomic model and thus showing that every coupled model can be rewritten as an atomic one.
This method can be applied recursively to remove the limitation that each submodel of a coupled model must be an atomic one, because if it were a coupled one, we could rewrite it to an atomic.
This proof goes as follows:
We wish to transform a *coupled* model

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

to the *atomic* model

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

So we have to define all of the variables of the *atomic model* in function of the given *coupled model*. The input and output variables X and Y are easy, since they stay the same.
The state S now encompasses all the states from the submodels, including their elapsed times:

$$S = \times_{i \in D} Q_i$$

with Q_i defined as:

$$Q_i = \{(s_i, e_i) | s_i \in S_i, 0 \leq e_i \leq ta_i(s_i)\}, \forall i \in D$$

The time advance function ta is constructed using these values and calculates the remaining time for each submodel.

$$ta(s) = min\{\sigma_i = ta_i(s_i) - e_i | i \in D\}$$

We define the component to transition with the *select* function, which selects a component i^* from the set of *imminent children IMM* defined as:

$$IMM(s) = \{i \in D | \sigma_i = ta(s)\}$$
$$i^* = select(IMM(s))$$

The output function λ can be defined as follows:

$$\lambda(s) = \begin{cases} Z_{i^*,self}(\lambda_i^*(s_i^*)) & \text{if } self \in I_{i^*} \\ \phi & \text{otherwise} \end{cases}$$

The *internal transition function* is defined for each part of the new state seperately:

$$\delta_{int}(s) = (\dots, (s'_j, e'_j), \dots)$$

With three possibilities:

$$(s'_j, e'_j) = \begin{cases} (\delta_{int,j}(s_j), 0) & \text{for } j = i^*, \\ (\delta_{ext,j}(s_j, e_j + ta(s), Z_{i^*,j}(\lambda_{i^*}(s_{i^*}))), 0) & \text{for } j \in I_{i^*}, \\ (s_j, e_j + ta(s)) & \text{otherwise} \end{cases}$$

This internal transition function also has to include parts of the submodels external transition function, since this is a direct result of the internal transition. Note that the resultant external transition function will only be called for input *external to the coupled model* and no longer for every submodel. Therefore, we have to call the external transition function immediately.
The *external transition* function is similar to the internal transition function:

$$\delta_{ext}(s, e, x) = (\dots, (s'_i, e'_i), \dots)$$

Now with two possibilities:

$$(s'_i, e'_i) = \begin{cases} (\delta_{ext,i}(s_i, e_i + e, Z_{self,i}(x)), 0) & \text{for } i \in I_{self} \\ (s_i, e_i + e) & \text{otherwise} \end{cases}$$

3. Modelica

Modelica was chosen as an intermediate language for 4 main reasons:

1. Modelica is a relatively mature language
2. Modelica has language features that are suitable for describing both continuous and discrete models
3. Modelica has a standard library for special constructs: the *Modelica Standard Library*
4. Some Modelica constructs can be used to increase the re-usability of DEVS models

As we will only use Modelica as an intermediate language to implement our DEVS models, we will only explain the relevant constructs.
It is relatively easy to construct DEVS models in Modelica. A simple example is a generator: it will periodically poke an integer on its output port. This can be defined as follows (comments inline):

```
class Generator
  // Show that it is an Atomic model
  extends AtomicDEVS;

  // Define the output ports
  output DevsPort p_out;

  // Define the output function
  function outputFnc
      // Define variables here
  algorithm
      // Poke 1 on the p_out port
      poke(p_out, 1);
  end outputFnc;

  function timeAdvance
    // Output the value for the time advance
    output Integer timespan;
    // Define variables here
  algorithm
      // Set to 1
      timespan := 1;
      // A return happens automatically
  end timeAdvance;
end Generator;
```

A more complete example can be found in section 8.
Since it is only an intermediate language, the users that follow the complete procedure defined here, will never have to use any Modelica code. Note that it is possible to alter the generated Modelica code before compiling and thus it is possible to include user defined Modelica models.

4. Global Overview

We will start by explaining the different steps mentioned in figure 1. This report will be mainly concerned with the first two parts, namely *Modelling* and *Verification*. We will subdivide the *verification* part in two distinct parts, where we will *flatten* the model in the first step and *compile* the model in the second step.

4.1. Modelling

The modelling part is concerned with the graphical modelling of the desired DEVS model. This is done using the DEVS metamodel (presented in section 5). This graphical model can be modelled using $AToM^3$, which will

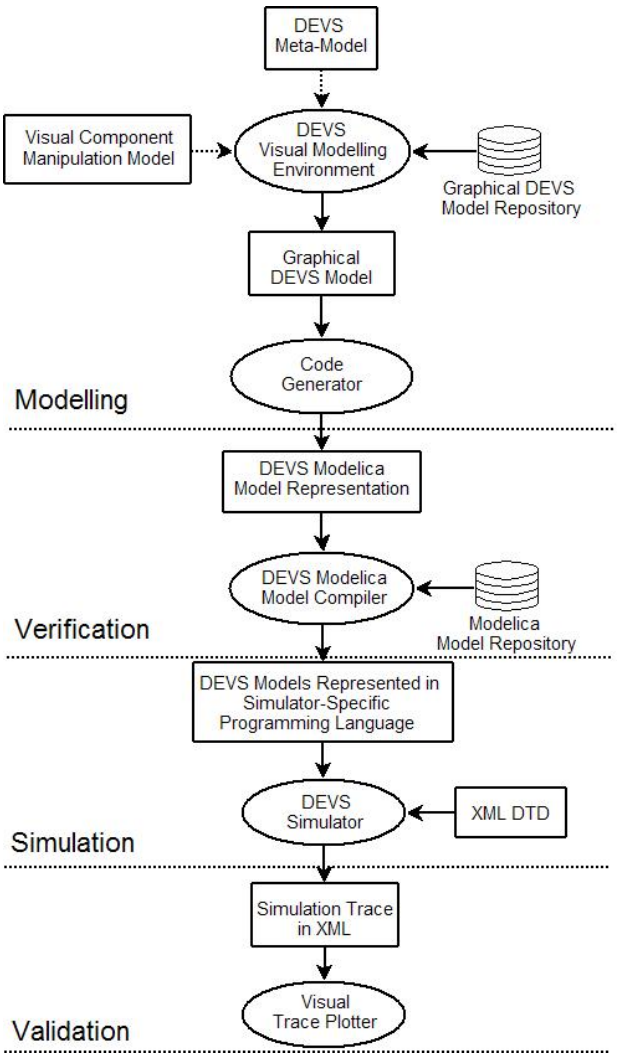

Figure 1: An overview of the translation from graphical model to simulation trace, as presented by Song (2006)

guarantee compliance to the metamodel due to the use of syntax-directed editing.
The visual modelling environment has several features:

- *The enviroment is automatically generated*
  The complete environment is modelled, from the properties and constraints of the models, to the possible operations on them. This minimizes the potential for misoperation by the user and aids in maintaining model consistency.

- *Translated to an neutral intermediate language*
  As was already previously mentioned, it would be wasteful to generate target code immediately, as the visual environment would then be bound to one simulator. When using a neutral intermediate language, we allow this visual environment to be used for many different simulators.

- *The environment can be easily updated*
  Since the environment is automatically generated, based on a DEVS metamodel (using ER diagrams) and a DEVS component manipulation model (using Statecharts), the environment is not hard-coded and can be easily updated should changes happen.

The output of this phase will be a model written in *Modelica*, which is generated from the model provided by the user.

4.2. Verification

Verification is possible by compiling the model and performing some checks on the Abstract Syntax Tree (AST). Some frequently done checks are presented by Labiche and Wainer (2005), like checking if ports are connected, their types match, no negative time advance is returned, ... Note that these are not currently present in our version, though there is some kind of syntax checking due to the compilation of the model. Also, due to the simulator-independent language, it is impossible to use simulator specific constructs which circumvent the DEVS formalism.
This phase will output a model written in *Python* that is compatible with the *PythonDEVS* simulator that we will be using.

4.3. Simulation

Simulation is concerned with using the models that were compiled in a previous step, to run a simulation. In our case, this is done with PythonDEVS (Bolduc and Vangheluwe (2001)). The simulation will generate a trace file, which can be used in the next step to visualise the simulation. Currently, PythonDEVS supports three different trace outputs: text, XML and VCD.
In this situation, XML (using a DTD) is the most versatile trace format. This is because *text* output is mainly concerned with being human readable, while *VCD* has several limitations which make it difficult to use for general trace output.
Again, the use of XML has several interesting benefits:

1. *Clear interface between simulator and analysis*
   A simulator should just be concerned with the simulation of the provided model and possibly generating trace files. Should the two be merged together, it would prevent correct comparison between the results of different simulators. Currently, the only way to compare the simulation trace of different simulators, is to manually insert output statements which is a tedious work that is prone to errors. If this trace outputting is done in the simulator itself in a standard format, both simulators can generate such a trace file which can both be compared using another tool.
2. *The XML file can be checked according to the DTD*
   XML files allow the specification of a DTD, which can be used to check the validity of a specified XML file. This functionality is not present in manual text output.
3. *XML is a standard way for data representation*
   XML files are widely used for data representation and many tools exist to perform operations on them. This makes it more easy to comprehend the information specified in the trace file and additionally allows the use of all these previously defined tools. This would not be possible with a proprietary way of data representation. On the other side, XML files tend to become relatively large in huge simulations.

The output of this phase will be a trace file written in *XML* as described by Song (2006), which is the result of simulating the provided model.

4.4. Validation

Validation means that a check will be performed to analyse whether or not a specified model meets the goal it was created for. This can be done in different ways, as it would also be possible to analyze the textual trace output. However, a visual trace is clearly more intuitive and comprehensible than wading through thousands of lines of text.
The validation phase will use the XML trace file generated in the previous phase, to make a plot of the behaviour of the system, allowing easy (graphical) validation. In our case, we will use the *TracePlotter* tool provided by Song (2006). This tool has two different levels of user interface: a simple user interface to quickly browse the simulation trace, and a more complex user interface, which allows the user to reparse the raw model state information and generated filtered traces.

4.5. Features of the overall architecture

The global architecture offers several interesting features:

1. Multiple levels of model reuse
   Due to the use of different languages which are gradually converted, it is possible to include models from a model repository defined in different languages. The first level allows inclusion of graphical models (defined in $AToM^3$), while the second level allows the inclusion of textual models (defined in *Modelica*). Even at the Python-level it is possible to include previously defined models, as long as they are compliant with PythonDEVS.
2. Clear boundaries between different steps
   The boundaries between the different phases (the dashed lines in figure 1) are clearly defined, making a strict seperation possible. Each tool is specialised in only one specific point.
3. Open structure
   The structure is completely open and allows the different steps to be created seperately. The files that are passed are mostly simulator and language independent (except the input to the simulator of course). This is also possible thanks to the clear boundaries between the different phases.

5. First phase: modelling in $AToM^3$

The first step consists of drawing the atomic models in $AToM^3$, together with their desired behaviour. The atomic models still require some code to

encode the behaviour, but this is done in Modelica to provide a programming language-independent environment. Coupled models are also easily constructed by using previously defined atomic models (through *instances* instead of the atomic model itself) and connecting their ports. A possibility to encode the state, message and experiment set-up is also provided, to provide a completely simulator independent environment.
This graphical approach has some important drawbacks, like the inability to define a *select* function, not being able to define connections using loops, some limitations on the next sequential state in case of an external transition and so on.
As mentioned by Song (2006), it is possible to generate Modelica code out of these models, which can not yet be simulated by itself but needs to be compiled using e.g. the µModelica compiler.
The metamodel of this graphical representation of the DEVS formalism can be found in figure 2. Most of it is fairly straightforward as it reminds of state machines and simple connections between ports. Notable is the *instantiation* possibility. To improve reuse of atomic models, the atomic model itself is simply a definition, while the *instantiation* is the actually used model. This way, a model can be instantiated multiple times without the need to completely redefine the model. Furthermore, it provides a clear seperation between definition and use.

The metamodel itself was updated to give more accurate names to attributes, the experiment was also re-added to the metamodel and some serious bugs were fixed.

6. Second phase: flattening with µModelica

In this second phase, the generated Modelica code will be symbolically flattened to create behaviourly equivalent Modelica code which defines only a single *atomic* DEVS model. This work closely resembles the one presented by Chen and Vangheluwe (2010), though our flattening takes a slightly different direction. This flattening phase was rewritten from scratch, so it represents the biggest chunk of this report and our main contribution.
It will scan the AST for all coupled models and their associated atomic models. The following steps must happen while flattening:

1. Perform direct connection for all relevant models, which will create a hierarchy with exactly 1 coupled model;

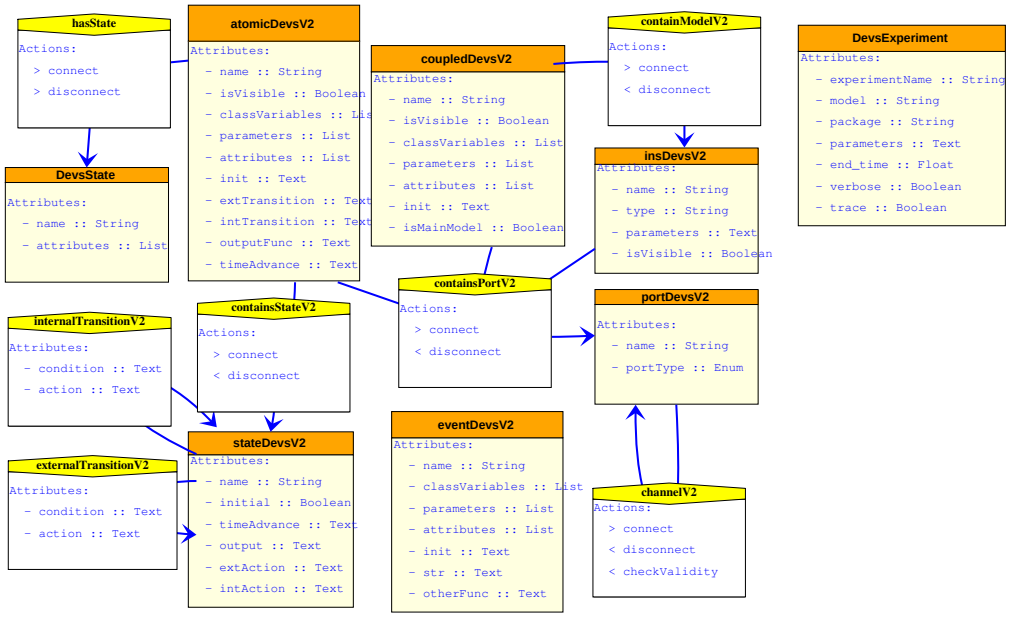

Figure 2: The metamodel of the DEVS formalism in $ATo M^3$, as defined by Song (2006) but with minor modifications

2. All atomic models must have their functions renamed to prevent naming conflicts, as they will be included in the newly created atomic model;
3. States have to be rewritten, as the original atomic model (of which it references the *sequential states*) no longer exists;
4. A new atomic model must be generated with all required functions and variables. These functions are written in Modelica and are parsed too, so we can use this AST as a template;
5. The old atomic and coupled models must be removed, as they are no longer required;
6. The modified AST must be dumped in Modelica syntax again.

It is clear that the first and fourth step are the most difficult and require the most effort. Note that the first step is not completely mandatory, as it would be possible to perform the flattening recursively from the bottom up. This would be rather slow and would still require a lot of function calls in the flattened model, resulting in a situation similar to message passing between coupled models which we try to avoid.

6.1. Direct connection

Direct connection means that the connections between different atomic models are done directly, without passing (possibly many) intermediate coupled models. This will reduce the structure of the hierarchy to only one coupled model and for the rest everything is just an atomic model.
An important result of *direct connection* is that the *select* and Z functions will have to be rewritten, since all atomic models become the direct child of the main coupled model. A possiblity for doing this is described by Chen and Vangheluwe (2010) and performed at simulator-level in PythonDEVS, though it isn't implemented in this flattening phase. While this might seem strange, it doesn't really pose a problem as the previous step (modelling in $AToM^3$) will never generate a *select* or Z function due to the previously mentioned limitations in the metamodel.

6.2. Implementation

In our implementation of the flattening phase, we introduced another step in the *devsmc.py* script, which can be told to flatten the provided Modelica code. Currently, it is only supported to *either* flatten or compile, not both sequentially.
The flattening will first do direct connection on the complete AST, thus rendering all coupled models except the root irrelevant. The root coupled model is thus extended with all atomic models.
All of these atomic models will get renamed by searching for them in the symbol table that is provided by the AST.
The new atomic model that is created, will be of a newly defined class, which will have a method *getASTElement()* that returns the new AST element that should overwrite the coupled model. This atomic model will contain all (renamed) functions and variables of all atomic models, together with new *DEVS related* functions, which are based on templates that should call all relevant atomic models.
Our templates are specified in the *lib/flatten.mo* Modelica file, which contains all necessary functions that can be used. The relevant rules of code will contain variables like *instA*, which are unspecified in the flatten.mo file, but will become specified later one. These lines, or rather their AST representation, will get deepcopied and get appended multiple times in the AST though with different variables.
Should the *flatten.mo* file be modified, the flattening phase of the compiler should also be modified, as it will contain a lot of references to these rules.

The flattening phase itself is defined in *devs/DevsFlatten.py*.

7. Third phase: PythonDEVS code generation with µModelica

The third and final phase will just compile Modelica code to PythonDEVS code. Since the main contribution of this work is in the first and (mainly) second phase, nearly no modifications are applied to this phase except for some bugs that were fixed due to the slightly different structure of the models that need to be translated. Furthermore, some of the AST nodes did not have a textual representation when printed as DEVS or Modelica code, so these were implemented too.
As previously mentioned, the generated Modelica code will use several non-default classes making these files incomplete. Therefore, it is necessary to import several DEVS library files which contain all these classes and some extra functionality, like *list* and *dict*. These library files were slightly updated, as the flattened structure needed some extra functions.
Also the code generation was slightly enhanced as the *termination_condition* was not completely modular.

8. Example and usage

To demonstrate our approach and show that it is feasible, we will create a small model using $ATоM^3$, flatten it and compile it to PythonDEVS code, which will ultimately be simulated. Since each phase needs to be done seperately, we will also include the command that was executed to obtain this output and some explanation.

8.1. Specification

Our model will be a very basic queue, which processes jobs in a time equal to the size of the job, before passing it on to the next processor. We also included a generator to make an autonomous model.
The generator will generate a new job with a fixed jobsize every second (simulation time) and send it to the connected processor.
The processor will receive a job and send it on its output ports after a time equal to the jobsize has passed. Should a new job arrive while the previous job is still being processed, the *old* event will be deleted.

8.2. Model

The model is very clear due to its graphical representation as can be seen in figure 3. The generator is always in the same state and just generates a new Job every second. The processor has two different states, one where it is *idle* and another one where it is *processing*. It can remain idle for infinitely long and if it is processing, it takes as long as the size of the job that was received.
It also includes an experiment to make simulation easier, as starting the simulation is also simulator-dependent.

8.2.1. Usage

To create the graphical model, it is necessary to start the provided $ATоM^3$ environment, which can be done by running the command

```
python atom3.py
```

In $ATоM^3$, the required metamodel (DevsV2_META.py) must be loaded. For instructions on how to use $ATоM^3$ itself, we refer to `http://atom3.cs.mcgill.ca/`. Usage of the provided metamodel should be fairly straightforward, as it is quite graphical.

8.3. Modelica code

Only the most important snippets are shown due to the massive amount of generated Modelica code.

```
package test
  import devs.*;
  import externalfunctions.*;
  import simulator.*;

  class GeneratorState
    Generator.SeqStates seqState(start=
        Generator.SeqStates.generating);
  end GeneratorState;

  class Generator
    extends AtomicDEVS;
    parameter String name;
    output DevsPort p_out;
    GeneratorState state();
    type SeqStates = enumeration(generating);
```

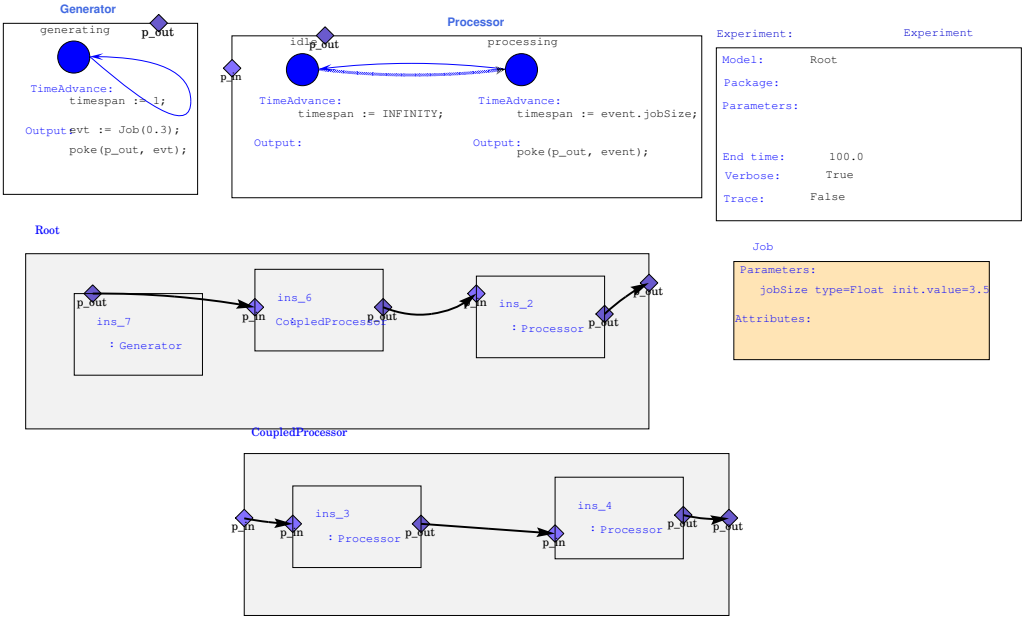

Figure 3: The model of the queue in $AToM^3$

```
function intTransition
algorithm
  if( state.seqState == SeqStates.generating ) then
    // Use Modelica syntax
    state.seqState := SeqStates.generating;
  end if;
end intTransition;

function outputFnc
  DevsEvent evt = null;
algorithm
  if( state.seqState == SeqStates.generating) then
    evt := Job(0.3);
    poke(p_out, evt);
  end if;
end outputFnc;

function timeAdvance
  output Integer timespan;
algorithm
  if( state.seqState == SeqStates.generating) then
    timespan := 1;
```

```
    end if;
  end timeAdvance;

end Generator;

class ProcessorState
  Processor.SeqStates seqState(start=Processor.SeqStates.idle
    );
end ProcessorState;

class Processor
  extends AtomicDEVS;
  parameter String name;
  input DevsPort p_in;
  output DevsPort p_out;
  ProcessorState state();
  type SeqStates = enumeration(idle, state_2);

  function extTransition
  algorithm
    if( state.seqState == SeqStates.idle ) then
      // Use Modelica syntax
      state.seqState := SeqStates.state_2;
    state.event := peek(p_in);
    end if;
  end extTransition;

  function intTransition
  algorithm
    // Use Modelica syntax
    if( state.seqState == SeqStates.state_2 ) then
      // Use Modelica syntax
      state.seqState := SeqStates.idle;
    end if;
  end intTransition;

  function outputFnc
    DevsEvent evt = null;
  algorithm
    if( state.seqState == SeqStates.state_2) then
      poke(p_out, state.event);
    end if;
  end outputFnc;

  function timeAdvance
    output Integer timespan;
```

```
    algorithm
      if( state.seqState == SeqStates.idle) then
        timespan := INFINITY;
      end if;
      if( state.seqState == SeqStates.state_2) then
        timespan := state.event.jobSize;
      end if;
    end timeAdvance;

  end Processor;

  class Root
    extends CoupledDEVS;
    parameter String name;
    Generator ins_0();
    Processor ins_1();
    Processor ins_2();
  equation
    connect( ins_0.p_out, ins_1.p_in);
    connect( ins_1.p_out, ins_2.p_in);
  end Root;

  class Job
    extends DevsEvent;
    parameter Real jobSize;
  end Job;

  class Experiment
    extends DevsExperiment;
    Root rootModel();
    Simulator sim(simModel=rootModel);
    Boolean verbose = True;
    Boolean trace = False;
    Real end_time = 100.0;
  end Experiment;

end test;
```

8.3.1. Usage

After the graphical model is constructed in $ATOM^3$, it can be exported as Modelica code using the **GEN** button in the toolbar. The generated files should contain all of the required Modelica code. Note that this code does several imports of some standard DEVS Modelica code (like the definition of the *DEVSPort*, *AtomicDEVS*, ... classes) and several small details. Though these files are not generated and are provided with the µModelica compiler,

which will be used in the next step.

8.4. Flattened Modelica code

Only the most important snippets are shown due to the massive amount of generated Modelica code.

```
package test
  import devs.*;
  import externalfunctions.*;
  import simulator.*;
  class ProcessorState
    type Processor_SeqStates
      EnumType value;
      parameter StringType quantity="";
      parameter EnumType min=idle, max=state_2;
      parameter EnumType start=idle;
      parameter BooleanType fixed=false;
      parameter BooleanType enable=true;
      constant EnumType idle;
      constant EnumType state_2;
    end Processor_SeqStates;
    Processor_SeqStates seqState(start=idle);
  end ProcessorState;
  class Root
    extends AtomicDEVS;
    dict X=dict();
    dict Y=dict();
    list timeLefts;
    Real elapsed=0;
    GeneratorState ins_0_state=GeneratorState();
    Real ins_0_totalWait=timeAdvance_Generator(ins_0_state);
    Real ins_0_timeLeft=ins_0_totalWait;
    ProcessorState ins_2_state=ProcessorState();
    Real ins_2_totalWait=timeAdvance_Processor(ins_2_state);
    Real ins_2_timeLeft=ins_2_totalWait;
    ProcessorState ins_1_state=ProcessorState();
    Real ins_1_totalWait=timeAdvance_Processor(ins_1_state);
    Real ins_1_timeLeft=ins_1_totalWait;
    function extTransition_Processor
      input ProcessorState state;
      input dict X;
      input Integer elapsed;
    algorithm
      if state.seqState == ProcessorState.idle then
        state.seqState := ProcessorState.state_2;
```

```
    state.event := X.get("p_in", None);
  end if;
end extTransition_Processor;
function intTransition_Processor
  input ProcessorState state;
algorithm
  if state.seqState == ProcessorState.state_2 then
    state.seqState := ProcessorState.idle;
  end if;
end intTransition_Processor;
function outputFnc_Processor
  DevsEvent evt=null;
  input ProcessorState state;
  output dict Y=dict();
algorithm
  if state.seqState == ProcessorState.state_2 then
    Y["p_out"] := state.event;
  end if;
end outputFnc_Processor;
function timeAdvance_Processor
  output Integer timespan;
  input ProcessorState state;
algorithm
  if state.seqState == ProcessorState.idle then
    timespan := INFINITY;
  end if;
  if state.seqState == ProcessorState.state_2 then
    timespan := state.event.jobSize;
  end if;
end timeAdvance_Processor;
function intTransition
  Integer counter=0;
  list new=list();
  list transitioned=list();
  list first=list();
  list elems=list();
  Real ta=0.0;
  dict Y_inst=dict();
  list dStar;
algorithm
  new := list();
  counter := 0;
  ins_0_timeLeft := ins_0_timeLeft-elapsed;
  ins_2_timeLeft := ins_2_timeLeft-elapsed;
  ins_1_timeLeft := ins_1_timeLeft-elapsed;
  if ins_0_timeLeft < 1e-10 then
```

```
    elems.append(ins_0_state);
  end if;
  if ins_2_timeLeft < 1e-10 then
    elems.append(ins_2_state);
  end if;
  if ins_1_timeLeft < 1e-10 then
    elems.append(ins_1_state);
  end if;
  if len(elems) > 1 then
    dStar := select(elems);
  else
    dStar := elems[0];
  end if;
  if dStar == ins_0_state then
    Y := outputFnc_Generator(ins_0_state);
    ins_0_state := intTransition_Generator(ins_0_state);
    Y := renamePorts(Y,"ins_0");
    Y := mapConnections(Y);
    ins_0_timeLeft := timeAdvance_Generator(ins_0_state);
    Y_inst := getInstances(Y,"ins_1");
    elapsed := ins_1_totalWait-ins_1_timeLeft;
    ins_1_state := extTransition_Processor(ins_1_state,
        Y_inst,elapsed);
    ins_1_timeLeft := timeAdvance_Processor(ins_1_state);
    ins_1_totalWait := ins_1_timeLeft;
  end if;
  if dStar == ins_2_state then
    Y := outputFnc_Processor(ins_2_state);
    ins_2_state := intTransition_Processor(ins_2_state);
    Y := renamePorts(Y,"ins_2");
    Y := mapConnections(Y);
    ins_2_timeLeft := timeAdvance_Processor(ins_2_state);
  end if;
  if dStar == ins_1_state then
    Y := outputFnc_Processor(ins_1_state);
    ins_1_state := intTransition_Processor(ins_1_state);
    Y := renamePorts(Y,"ins_1");
    Y := mapConnections(Y);
    ins_1_timeLeft := timeAdvance_Processor(ins_1_state);
    Y_inst := getInstances(Y,"ins_2");
    elapsed := ins_2_totalWait-ins_2_timeLeft;
    ins_2_state := extTransition_Processor(ins_2_state,
        Y_inst,elapsed);
    ins_2_timeLeft := timeAdvance_Processor(ins_2_state);
    ins_2_totalWait := ins_2_timeLeft;
  end if;
```

```
end intTransition;
function extTransition
  input dict X;
  input Real elapsed;
  dict Y=dict();
  list keys=list();
  dict Y_inst=dict();
algorithm
  Y := mapConnections(X);
  keys := Y.keys();
  if keys.contains("ins_0") then
    Y_inst := getInstances(Y,"ins_0");
    elapsed := ins_0_totalWait-ins_0_timeLeft;
    ins_0_state := extTransition_Generator(ins_0_state,
        Y_inst,elapsed);
    ins_0_timeLeft := timeAdvance_Generator(ins_0_state);
    ins_0_totalWait := ins_0_timeLeft;
  end if;
  if keys.contains("ins_2") then
    Y_inst := getInstances(Y,"ins_2");
    elapsed := ins_2_totalWait-ins_2_timeLeft;
    ins_2_state := extTransition_Processor(ins_2_state,
        Y_inst,elapsed);
    ins_2_timeLeft := timeAdvance_Processor(ins_2_state);
    ins_2_totalWait := ins_2_timeLeft;
  end if;
  if keys.contains("ins_1") then
    Y_inst := getInstances(Y,"ins_1");
    elapsed := ins_1_totalWait-ins_1_timeLeft;
    ins_1_state := extTransition_Processor(ins_1_state,
        Y_inst,elapsed);
    ins_1_timeLeft := timeAdvance_Processor(ins_1_state);
    ins_1_totalWait := ins_1_timeLeft;
  end if;
end extTransition;
function timeAdvance
  output Real timeSpan;
algorithm
  timeLefts := list();
  timeLefts.append(ins_0_timeLeft);
  timeLefts.append(ins_2_timeLeft);
  timeLefts.append(ins_1_timeLeft);
  timeSpan := min(timeLefts);
end timeAdvance;
function outputFnc
  output dict Y=dict();
```

```
algorithm
  Y := mapConnections(X);
  X := dict();
end outputFnc;
function mapConnections
  input dict X;
  output dict Y;
algorithm
  Y := dict();
  if X.__contains__("ins_0_p_out") then
    Y["ins_1_p_in"] := X["ins_0_p_out"];
  end if;
  if X.__contains__("ins_1_p_out") then
    Y["ins_2_p_in"] := X["ins_1_p_out"];
  end if;
end mapConnections;
function getInstances
  input dict X;
  input String name;
  output dict Y;
  list keys;
  Integer counter;
  String key;
  String new_key;
  String searchname;
  Integer length;
algorithm
  Y := dict();
  keys := X.keys();
  counter := 0;
  length := keys.__len__();
  while counter<length loop
    key := keys[counter];
    if key.__contains__(name) then
      searchname := name+"_";
      new_key := key.split(searchname);
      new_key := new_key[1];
      Y[new_key] := X[key];
    end if;
    counter := counter+1;
  end while;
end getInstances;
function renamePorts
  input dict X;
  input String name;
  output dict Y=dict();
```

```
      list keys=list();
      String newname="";
      String key="";
      Integer counter=0;
      Integer length=0;
    algorithm
      keys := X.keys();
      counter := 0;
      length := keys.__len__();
      while counter<length loop
        key := keys[counter];
        newname := name+"_"+key;
        Y[newname] := X[key];
        counter := counter+1;
      end while;
    end renamePorts;
    function select
      input list immChildren;
      output list child;
    algorithm
      child := immChildren[0];
    end select;
  end Root;
  class Job
    extends DevsEvent;
    parameter Real jobSize;
  end Job;
  class Experiment
    extends DevsExperiment;
    Root rootModel();
    Simulator sim(simModel=rootModel);
    Boolean verbose=True;
    Boolean trace=False;
    Real end_time=100.0;
  end Experiment;
end test;
```

8.4.1. Usage

An important step before using the µModelica compiler, is to set several PATH environment variables first. These are needed to do proper imports and otherwise the compiler will crash. To set these variabeles, it suffices to use the *setPATH* file.
To flatten code and set the environment variables, the following command must be executed

```
source setPATH
python devsmc.py -f test.mo
```

This command will have generated a flattened model with extension *.flat.mo*, which is Modelica code too.

8.5. PythonDEVS code

Only the most important snippets are shown due to the massive amount of generated PythonDEVS code.

```
class ProcessorState:
  idle = 'idle'
  state_2 = 'state_2'
  def __init__(self):
    self.seqState = self.idle

class Root( AtomicDEVS ):
  def __init__(self):
    AtomicDEVS.__init__(self)
    self.X = dict()
    self.Y = dict()
    self.timeLefts = None
    self.elapsed = 0
    self.ins_0_state = GeneratorState()
    self.ins_0_totalWait = self.timeAdvance_Generator(self.
        ins_0_state)
    self.ins_0_timeLeft = self.ins_0_totalWait
    self.ins_2_state = ProcessorState()
    self.ins_2_totalWait = self.timeAdvance_Processor(self.
        ins_2_state)
    self.ins_2_timeLeft = self.ins_2_totalWait
    self.ins_1_state = ProcessorState()
    self.ins_1_totalWait = self.timeAdvance_Processor(self.
        ins_1_state)
    self.ins_1_timeLeft = self.ins_1_totalWait
  def intTransition_Generator( self, state ):
    if(state.seqState == GeneratorState.generating):
      state.seqState = GeneratorState.generating
    return state
  def outputFnc_Generator( self, state ):
    Y=dict()
    evt = None
    if(state.seqState == GeneratorState.generating):
      evt = Job(0.3)
      Y["p_out"] = evt
```

```
    return Y
  def timeAdvance_Generator( self, state ):
    if(state.seqState == GeneratorState.generating):
      timespan = 1
    return timespan
  def extTransition_Processor( self, state, X, elapsed ):
    if(state.seqState == ProcessorState.idle):
      state.seqState = ProcessorState.state_2
      state.event = X.get("p_in", None)
    return state
  def intTransition_Processor( self, state ):
    if(state.seqState == ProcessorState.state_2):
      state.seqState = ProcessorState.idle
    return state
  def outputFnc_Processor( self, state ):
    Y=dict()
    evt = None
    if(state.seqState == ProcessorState.state_2):
      Y["p_out"] = state.event
    return Y
  def timeAdvance_Processor( self, state ):
    if(state.seqState == ProcessorState.idle):
      timespan = INFINITY
    if(state.seqState == ProcessorState.state_2):
      timespan = state.event.jobSize
    return timespan
  def intTransition( self ):
    counter = 0
    new = list()
    transitioned = list()
    first = list()
    elems = list()
    ta = 0.0
    Y_inst = dict()
    dStar = None
    new = list()
    counter = 0
    self.ins_0_timeLeft = self.ins_0_timeLeft - self.elapsed
    self.ins_2_timeLeft = self.ins_2_timeLeft - self.elapsed
    self.ins_1_timeLeft = self.ins_1_timeLeft - self.elapsed
    if(self.ins_0_timeLeft < 1e-10):
      elems.append(self.ins_0_state)
    if(self.ins_2_timeLeft < 1e-10):
      elems.append(self.ins_2_state)
    if(self.ins_1_timeLeft < 1e-10):
      elems.append(self.ins_1_state)
```

```
    if(len(elems) > 1):
      dStar = self.select(elems)
    else:
      dStar = elems[0]
    if(dStar == self.ins_0_state):
      self.Y = self.outputFnc_Generator(self.ins_0_state)
      self.ins_0_state = self.intTransition_Generator(self.
          ins_0_state)
      self.Y = self.renamePorts(self.Y, "ins_0")
      self.Y = self.mapConnections(self.Y)
      self.ins_0_timeLeft = self.timeAdvance_Generator(self.
          ins_0_state)
      Y_inst = self.getInstances(self.Y, "ins_1")
      self.elapsed = self.ins_1_totalWait - self.ins_1_timeLeft
      self.ins_1_state = self.extTransition_Processor(self.
          ins_1_state, Y_inst, self.elapsed)
      self.ins_1_timeLeft = self.timeAdvance_Processor(self.
          ins_1_state)
      self.ins_1_totalWait = self.ins_1_timeLeft
    if(dStar == self.ins_2_state):
      self.Y = self.outputFnc_Processor(self.ins_2_state)
      self.ins_2_state = self.intTransition_Processor(self.
          ins_2_state)
      self.Y = self.renamePorts(self.Y, "ins_2")
      self.Y = self.mapConnections(self.Y)
      self.ins_2_timeLeft = self.timeAdvance_Processor(self.
          ins_2_state)
    if(dStar == self.ins_1_state):
      self.Y = self.outputFnc_Processor(self.ins_1_state)
      self.ins_1_state = self.intTransition_Processor(self.
          ins_1_state)
      self.Y = self.renamePorts(self.Y, "ins_1")
      self.Y = self.mapConnections(self.Y)
      self.ins_1_timeLeft = self.timeAdvance_Processor(self.
          ins_1_state)
      Y_inst = self.getInstances(self.Y, "ins_2")
      self.elapsed = self.ins_2_totalWait - self.ins_2_timeLeft
      self.ins_2_state = self.extTransition_Processor(self.
          ins_2_state, Y_inst, self.elapsed)
      self.ins_2_timeLeft = self.timeAdvance_Processor(self.
          ins_2_state)
      self.ins_2_totalWait = self.ins_2_timeLeft
    return self.state
  def extTransition( self, X, elapsed ):
    Y = dict()
    keys = list()
```

```
    Y_inst = dict()
    Y = self.mapConnections(X)
    keys = self.Y.keys()
    if(keys.contains("ins_0")):
      Y_inst = self.getInstances(Y, "ins_0")
      elapsed = self.ins_0_totalWait - self.ins_0_timeLeft
      self.ins_0_state = extTransition_Generator(self.
          ins_0_state, Y_inst, elapsed)
      self.ins_0_timeLeft = self.timeAdvance_Generator(self.
          ins_0_state)
      self.ins_0_totalWait = self.ins_0_timeLeft
    if(keys.contains("ins_2")):
      Y_inst = self.getInstances(Y, "ins_2")
      elapsed = self.ins_2_totalWait - self.ins_2_timeLeft
      self.ins_2_state = self.extTransition_Processor(self.
          ins_2_state, Y_inst, elapsed)
      self.ins_2_timeLeft = self.timeAdvance_Processor(self.
          ins_2_state)
      self.ins_2_totalWait = self.ins_2_timeLeft
    if(keys.contains("ins_1")):
      Y_inst = self.getInstances(Y, "ins_1")
      elapsed = self.ins_1_totalWait - self.ins_1_timeLeft
      self.ins_1_state = self.extTransition_Processor(self.
          ins_1_state, Y_inst, elapsed)
      self.ins_1_timeLeft = self.timeAdvance_Processor(self.
          ins_1_state)
      self.ins_1_totalWait = self.ins_1_timeLeft
    return self.state
  def timeAdvance( self ):
    self.timeLefts = list()
    self.timeLefts.append(self.ins_0_timeLeft)
    self.timeLefts.append(self.ins_2_timeLeft)
    self.timeLefts.append(self.ins_1_timeLeft)
    timeSpan = min(self.timeLefts)
    return timeSpan
  def outputFnc( self ):
    Y=dict()
    Y = self.mapConnections(self.X)
    self.X = dict()
    return Y
  def mapConnections( self, X ):
    Y = dict()
    if(X.__contains__("ins_0_p_out")):
      Y["ins_1_p_in"] = X["ins_0_p_out"]
    if(X.__contains__("ins_1_p_out")):
      Y["ins_2_p_in"] = X["ins_1_p_out"]
```

```
    return Y
  def getInstances( self, X, name ):
    keys = None
    counter = 0
    key = ""
    new_key = ""
    searchname = ""
    length = 0
    Y = dict()
    keys = X.keys()
    counter = 0
    length = keys.__len__()
    while counter < length:
      key = keys[counter]
      if(key.__contains__(name)):
        searchname = name + "_"
        new_key = key.split(searchname)
        new_key = new_key[1]
        Y[new_key] = X[key]
      counter = counter + 1
    return Y
  def renamePorts( self, X, name ):
    Y=dict()
    keys = list()
    newname = ""
    key = ""
    counter = 0
    length = 0
    keys = X.keys()
    counter = 0
    length = keys.__len__()
    while counter < length:
      key = keys[counter]
      newname = name + "_" + key
      Y[newname] = X[key]
      counter = counter + 1
    return Y
  def select( self, immChildren ):
    child = immChildren[0]
    return child

class Job( object ):
  def __init__(self, jobSize):
    self.jobSize = jobSize

class Experiment( object ):
```

```
  def __init__(self):
    self.rootModel = Root()
    self.sim = Simulator(model = self.rootModel)
    self.verbose = True
    self.trace = False
    self.end_time = 100.0
  def simulate(self):
    self.sim.simulate(termination_condition=terminate_at(self.
        end_time), verbose=self.verbose, trace=self.trace)
def terminate_at(t):
 def f(model, clock):
  if clock >= t:
    return True
  else:
    return False
 return f
if __name__ == '__main__':
  experiment = Experiment()
  experiment.simulate()
```

8.5.1. Usage

Compiling Modelica code to PythonDEVS code is similar to flattening, only now the *-f* flag must be ommitted:

```
python devsmc.py test.flat.mo
```

Which will generate the code in the file *test.flat.py*, though it requires some slight modifications due to the way PythonDEVS models import the simulator. So it is required to either put the *pydevs* folder in the same directory as the models, or modify the import path in the model itself which should now point to a non existing location.

8.6. Simulation

Only the most important snippets are shown due to the massive amount of output. Note that the trace output will not be very clear, since the complete structure was reduced to a single atomic model, which means that the simulator will not be able to differentiate between the different atomic models that are represented in this newly created atomic model. Should approximately the same output be desired, it might be possible to define a similar *print* function, though this would be relatively unmaintainable as it should have the same format as the output of the simulator.

```
--  Current Time:       0.00
    ----------------------------------------


    INITIAL CONDITIONS in model <A1>
      Initial State: None
      Next scheduled internal transition at time 1

--  Current Time:       1.00
    ----------------------------------------


    INTERNAL TRANSITION in model <A1>
      New State: None
      Output Port Configuration:
      Next scheduled internal transition at time 1.3

--  Current Time:       1.30
    ----------------------------------------


    INTERNAL TRANSITION in model <A1>
      New State: None
      Output Port Configuration:
      Next scheduled internal transition at time 1.6

--  Current Time:       1.60
    ----------------------------------------


    INTERNAL TRANSITION in model <A1>
      New State: None
      Output Port Configuration:
      Next scheduled internal transition at time 2.0

--  Current Time:       2.00
    ----------------------------------------


    INTERNAL TRANSITION in model <A1>
      New State: None
      Output Port Configuration:
      Next scheduled internal transition at time 2.3
```

```
--  Current Time:        2.30
    ------------------------------------------

    INTERNAL TRANSITION in model <A1>
      New State: None
      Output Port Configuration:
      Next scheduled internal transition at time
          2.5999999999999996
```

8.6.1. Usage

Simulating the model itself is no longer part of our approach, but we mention the command for completeness.

```
python test.flat.py
```

9. Performance evaluation

Symbolic flattening is often mentioned as a way to achieve improved performance. While this may be possible in case the simulator has a very inefficient way to handle hierarchy, most performance oriented simulators will have more efficient algorithms, which are often language dependent (e.g. difference between the available function in the Python libraries and C++ standard libraries).

For example, the original PyDEVS simulator as described by Bolduc and Vangheluwe (2001) will sort the event list to find the first element to transition. This is clearly $O(n \cdot log(n))$, so our flattening will reduce this to an $O(n)$ algorithm, which should clearly be more efficient in big models. The newer version of the PyDEVS simulator will do a lot more and will make use of heaps, invalidation of elements, ... to achieve near $O(log(n))$ complexity. While it would also be possible to implement this in our flattening phase, it would be more difficult due to the fact that all this code will have to be ported to Modelica and modifed to make sure that it can be used as a template for our compiler.

Should our current flattening approach be used on performance oriented simulators, we will therefor see a *decrease* in performance, as we disregard all optimisations of the simulator. This can be seen in figure 4, the complexities differ from those mentioned above, since the above only take into account the searching for the imminent component, while the figure shows

the complete simulation. Note that in small cases, the flattened version is still slightly faster, which is due to the lower initialisation time. However, it is clear that the complexity of the simulation will get altered. In the old version of the PyDEVS simulator, we achieve a slight improvement, as seen in figure 5. Note that if the flattening phase would contain approximately the same algorithms, it might be possible to come close in terms of performance, with the added advantage that these algorithms could be ported to *every* possible simulator, thus easing the task of simulator builders.
Even though both simulators simulate the same flattened model, which avoids the use of high-complexity code for coupled models, there is still a huge difference in execution times between the old and new PythonDEVS. This indicates that symbolic flattening alone is not able to make every simulator perform with the same complexity and certainly not with the same execution time. Which is actually quite logical, since a simulator contains a lot more than only coupled model simulation code. On the other hand, this code is often the one with the highest impact on simulation complexity as can be seen in figure 4, where a flattened model is compared to a non-flattened model.
For reference, we also included models that are overly hierarchical, with only 1 atomic model, which is nested in several coupled models. In this case, it can be seen that there indeed is a significant performance increase, mainly because in the flat version, the intermediate coupled models are removed at compile time.
Of course, such a model is very artificial and is unlikely to happen in real models. On the other hand, it nicely shows that the direct connection algorithm can effectively reduce the complexity in hierarchical models, both at compile-time (as in the flattened version) as at run-time (as is done in the new version of PyDEVS). The slight increase in simulation time due to the direct connection is often oughtweighted by the high performance increase that can be noticed due to this optimisation. For more information on the use of direct connection at run-time and the results in different situations, we refer to Van Tendeloo (2013). For the flattened version, it is clear that the execution time stays constant. The non-flattened version takes time dependent on the depth. Even though the new version of PythonDEVS includes direct connection at run-time, this whole reconnection sequence does take some time dependent on the depth, which is taken into account in the performance timings. This is the only reason why there is a slight increase in time due to the increasing depth.

The comparison in figure 6 seems to have a lot of jitter, though this is mainly due to the fact that a very small time scale is used. For example in the flattened model, the duration varies between 0.250 and 0.270. However, it should be clear that the complexity is completely constant.

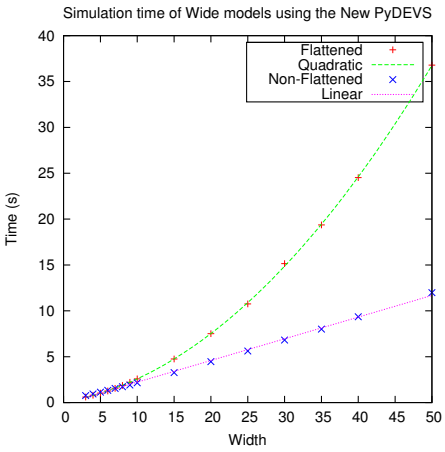

Figure 4: Performance comparison between a flattened and non-flattened model in the newly modified PythonDEVS simulator

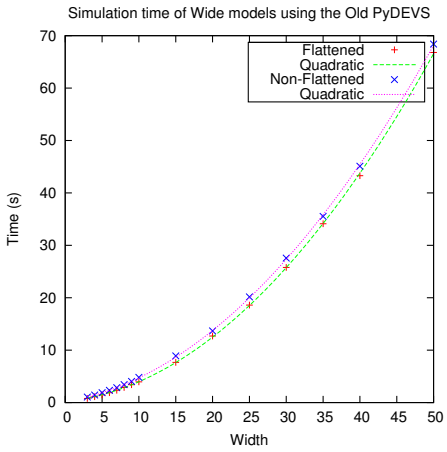

Figure 5: Performance comparison between a flattened and non-flattened model in the original PythonDEVS simulator

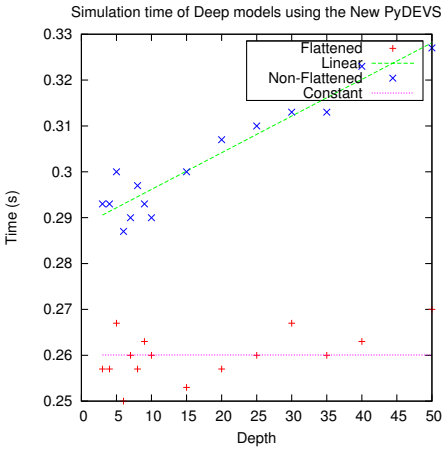

Figure 6: Performance comparison between a flattened and non-flattened hierarchical model in the newly modified PythonDEVS simulator

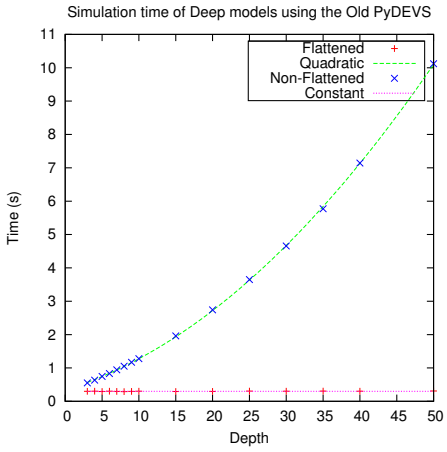

Figure 7: Performance comparison between a flattened and non-flattened hierarchical model in the original PythonDEVS simulator

10. Related work

The three different phases were already done in different levels of completeness, so our main contribution lies in combining these steps and fixing bugs or incompletenesses that arise while doing so. The main work was in the symbolic flattening step, which was rewritten from scratch.
The first step, about graphically modelling the DEVS formalism, was done by Song (2006) and Posse and Bolduc (2003). The symbolic flattening phase was mentioned by Chen and Vangheluwe (2010), though a slightly different approach was taken. Whereas their approach focusses on performance improvements, our approach focusses on fitting it in a more global overview with performance being a secondary goal.
The final step, which compiles the Modelica code to PythonDEVS code, is a combination of the original µModelica compiler by Xu (2005) and the PythonDEVS extension to this compiler by Song (2006).

11. Future work

There are a lot of opportunities for future work, mainly to the µModelica compiler. The most imporant work that could be done is concerned with extending the compiler in several dimensions:

1. *Implementing different code generators*
   Currently only PythonDEVS is supported, some possible extensions might be ADEVS, VLE, CD++, ... This way the approach allows for truely simulator independent modelling.
2. *Implementing further verification possibilities*
   Currently nearly no verification is present that checks whether or not the model is completely compliant to the DEVS formalism. Some possible directions for verifications are mentioned by Labiche and Wainer (2005).
3. *Implementing optimisations in the flattening phase*
   The flattening phase as we wrote here is still a prototype, as it closely resembles the closure under coupling proof of Classic-DEVS. Most of the optimisations that are possible for simulators can also be implemented in the generated models, since the models will actually contain parts of the simulation algorithm. This allows for an extremely light-weight simulator, as most parts of the simulator are already implemented in the model itself. Furthermore, these optimisations will

be kept in every different simulator as they happen at a simulator-independent level.

4. *Select and Z function*
   As previously mentioned, the *select* and Z functions are not yet supported. Implementing these two features would make the flattening phase of the compiler completely conform to the DEVS formalism.

12. Conclusion

We have shown that it is possible to use a three-phase approach to a) graphically model a DEVS model, b) generate a simulator-neutral, intermediate model out of it, c) perform several symbolic optimisations, d) compile this intermediate model to simulator-specific code.
Furthermore, this has given us some insight in the performance improvements that flattening might offer, but also in some simumlator parts that cannot be accelerated using flattening.
Three different tools were combined to make this possible, allowing for maximum reuse and openness between each step.

References

Bernard P. Zeigler, H. P., Kim, T. G., 2000. Theory of Modeling and Simulation, 2nd Edition. Academic Press.

Bolduc, J.-S., Vangheluwe, H., 2001. The modelling and simulation package pythondevs for classical hierarchical devs. Tech. rep., MSDL Technical Report.

Chen, B., Vangheluwe, H., 2010. Symbolic flattening of devs models. In: 2010 Summer Simulation Multiconference. SummerSim '10. Society for Computer Simulation International, San Diego, CA, USA, pp. 209–218.
URL `http://dl.acm.org/citation.cfm?id=1999416.1999442`

Labiche, Y., Wainer, G., 2005. Towards the verification and validation of devs models. In: in Proceedings of 1st Open International Conference on Modeling & Simulation, 2005. pp. 295–305.

Posse, E., Bolduc, J.-S., 2003. Generation of devs modelling and simulation environments. In: Summer Computer Simulation Conference. Student Workshop. pp. S139–S146.

Song, H., 2006. Infrastructure for devs modelling and experimentation. Master's thesis, McGill University.

Traoré, M. K., 2009. A graphical notation for devs. In: Proceedings of the 2009 Spring Simulation Multiconference. SpringSim '09. Society for Computer Simulation International, San Diego, CA, USA, pp. 162:1–162:7.
URL `http://dl.acm.org/citation.cfm?id=1639809.1655391`

Van Tendeloo, Y., 2013. Research internship 1: Optimizing pydevs. Tech. rep., MSDL.

Xu, W., 2005. The design and implementation of the modelica compiler. Master's thesis, McGill University.