Research Internship I: Efficient DEVS simulation

Yentl Van Tendeloo
Supervisor: Hans Vangheluwe

January 6, 2013

Abstract

In this report, we will direct our attention towards efficient simulation of DEVS models. Efficiency can be defined on several levels, of which the most important will be discussed. The first one being the formalism itself, since some formalisms might be more specialised for certain situations and thus allow special constructs for improved performance. For this, we will mention and compare the most popular extensions to the base formalism. The second is the simulator and the features it provides. Here we will compare different simulators and their features. The last part is where we apply many changes to the PythonDEVS simulator, in an effort to speed up simulation.

Contents

1

The DEVS formalism and its extensions

Since the conception of the DEVS formalism by Zeigler, a lot of extensions have been created to this basic formalism. Each of these extensions has its own advantages, making it more suitable in some application domains. However, these different extensions require different (abstract) simulators, thus restricting the choice of simulator when a certain formalism is required. To make a comprehensive comparison between different simulators, the formalisms that these simulators implement have to be compared first.
The two main formalisms are the Classic DEVS formalism and the Parallel DEVS formalism. Most extensions that have been conceived, are based on one of these two. In this chapter, only the most basic versions of the formalisms are mentioned, meaning that they will all relate to the Classic DEVS formalism. Where applicable, a reference is made to the parallel version.
For each formalism, we will discuss the structure of the atomic and coupled model and show that it is closed under coupling (thus allowing hierarchical structures). Furthermore, there will be a comparison with the Classic DEVS formalism. Possibly, some alternatives or close variants will briefly be mentioned.

1.1 Classic DEVS

Classic DEVS (CDEVS)[1] is the formalism that sparked the many extensions that exist today, so it is still discussed here as an introduction to the formalism.
The classic DEVS formalism basically consists of *atomic* models, which are connected to each other in a *coupled* model. The atomic models are *behavioural* (they model *only* behaviour), while the coupled models are *structural* (they model *only* structure).

1.1.1 Atomic DEVS

An atomic DEVS model is a structure:

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

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

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

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

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

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 real time). Note that this function does not require the

[1] This explanation is based on [36]

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

$$\lambda : S \to Y \cup \{\phi\}$$

With ϕ defined as the *'empty event'*.
The *external transition function* δ_{ext} gets called as soon as an *external input* ($\in X$) is received in the model. This transition function is defined as

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

with e being the elapsed time since 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 *internal transition function*.

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

Note that we included $+\infty$, 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.

1.1.2 Coupled DEVS

A Coupled DEVS is a structure

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

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.
The set of components $\{M_i\}$ is a set containing the **atomic**[2] DEVS structure of all subcomponents defined 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 we also include *self* in this definition, as it is possible for a component to send or receive messages to the coupled model itself (which will then route it further to another atomic model).

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

A component cannot influence components outside of the current coupled model[3], nor can it influence itself *directly*[4].

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

The couplings are further determined by the *output-to-input translation functions* $Z_{i,j}$. These functions are applied to the messages being passed, depending on the output and input port. These functions allow for more reuse, since it allows for a kind of *wrapper*.

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

Finally, the *tie-breaking function select* is used to solve collisions between multiple components. Two or more components collide if they want to execute their *internal* transition function at the same time. The function must return exactly one component, which will execute its internal transition function.
Note that collisions are only possible between internal events, since external events will be processed together with the corresponding internal event and the order of *external transition functions* does not influence the execution order, as they only influence *one* atomic model.

[2] We now require that these components are atomic models, though this can later be relaxed to coupled models due to closure under coupling.

[3] Should this be desired, a message must be sent to the coupled model, which will then be able to send the message to models in its own scope. This is needed for hierarchical composition.

[4] Of course, a model can influence itself, though this requires an intermediate model

1.1.3 Closure under coupling

To prove that Classic DEVS is closed under coupling, we have to show that for every coupled DEVS model, an equivalent *atomic* model can be constructed. This is needed since the definition of the coupled model states that its submodels should be atomic. So if we can prove that every coupled model can be written as an equivalent atomic model, we can relax our definition for the submodels to atomic *and coupled* models.
We wish to transform a *coupled* model

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

to the *atomic* model

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

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 elapsed time is necessary, since the elapsed time from the new atomic model will update more frequently then the submodel's elapsed time.
The time advance function ta is constructed using these values and calculates the remaining time for each submodel[5].

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

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:

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

Note that λ only outputs events if the coupled model itself receives any output. Due to the (possibly many) subcomponents, the resultant atomic model will have many internal transitions without actually sending output. This doesn't mean that nothing happened, but that the 'output' is consumed internally.
The *internal transition function* is defined for each part of the new state seperately:

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

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:

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

Now with two possibilities:

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

[5]This construction causes multiple calls to the time advance function of component i. This doesn't form a problem in the case that this function is deterministic (as it should be!). So it is illegal to e.g. return a random variable that gets regenerated each time the time advance function is called.

1.1.4 Example

An easy example is that of a very basic queue that processes all kinds of events for 1 simulation time unit. If another event comes in while the previous one is still being processed, the previous event is overwritten.

$$T = \mathbb{R}$$
$$X = \{EVENT\}$$
$$Y = \{EVENT\}$$
$$S = \{active, passive\}$$
$$\delta_{int}(active) = passive$$
$$ta(passive) = \infty$$
$$ta(active) = 1$$
$$\delta_{ext}((active, e), EVENT) = active$$
$$\delta_{ext}((passive, e), EVENT) = active$$
$$\lambda(active) = EVENT$$

Since we stated that $ta(passive) = \infty$, the state *passive* just waits infinitely long for an external input.
For completeness, we also had to mention both states in the external transition function, since it doesn't actually matter. Note that cases that are not mentioned are impossible (like $\delta_{int}(passive)$, due to the infinite time advance).

1.1.5 Problems

The main problems of Classic DEVS require some more attention, as this were the reasons for the introduction of several extensions. Those identified in [36] are:

- Conflict between internal and external transitions
  When both an external and an internal event should happen at the same moment in a model, the model will take the external event first and thus ignore the internal event. It would be the responsibility of the modeler to make the models 'remember' that they had to make an internal event. This is quite logical, since the external transition function is the direct result of a previous internal transition, which was selected above the own internal transition function.
  The external transition function changes the state of the model and thus possibly alters the *time advance* function and the result of the output function. This means that the order in which these transitions happens does have a significant impact on the simulation.
- Limited parallel implementation
  As the size of models keeps increasing, both in number of components and computationally, the speed at which the simulation runs will naturally drop. With the current shift to parallel and distributed computing, it would be beneficial to be able to calculate some transitions in parallel.
  Note though that the problem only lies in the *internal transition function*, since all other functions can easily be parallellised. The other functions can be parallellised because they have no influence on *other* models, making it unnecessary to serialise them. Only the *internal transition function* can do this, as it implies the sending of the result of the *output function*.
- *Select* function is artificial
  The select function is an artificial legacy, because in the real world nothing happens at *exactly* the same moment[6]. However, due to our limited precision in the time base, it is possible that some models (seem to) collide. How this is solved is again the responsibility of the modeler by defining a *select* function. This function is rather far from the problem domain, since it only exists in our situations.
- No variable structure
  As mentioned in [6], a lot of models will be easier to model if the structure could change. However, it was also shown that these changes can be modeled with some special attention, though this would become too complex in large models. So it might be interesting should the formalism support this kind of changes in an easy way.
  Variable structure is not limited to only changing couplings, but also enables the dynamic addition or deletion of complete subcomponents.

In [15], approximately the same problem is identified:

- Ambiguity
  In Classic DEVS, ambiguity arises when an external and an internal event happen simultaneously. This problem is caused

[6] In the case of Classic DEVS, a collision is a rather bad situation for the performance. However, as we will see later, Parallel DEVS actually profits from collisions. Some implementations of Parallel DEVS even go as far as to reduce the accuracy of the time base to increase the number of collisions.

by the artificiality of the select function, as this might not properly reflect the intent of the co-occurence of the events in the system.

1.2 Parallel DEVS

Parallel DEVS (PDEVS) was presented in [15] as a revision of Classic DEVS, to provide a formalism that has better support for parallellism. Currently, Parallel DEVS has replaced Classic DEVS as the 'basic' version of DEVS in many simulators.
The corresponding abstract simulator can be found in [14]. The main change in this formalism is the introduction of *bags*, the *confluent transition function* and the removal of the *select* function.

1.2.1 Atomic DEVS

The atomic model of Parallel DEVS is mainly the same as with Classic DEVS. The structure of an atomic model is:

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

With X, Y, S, δ_{int} and ta defined the same as in Classic DEVS:

- X is the set of input events;
- Y is the set of output events;
- S is the set of sequential states;
- δ_{int} the internal transition function;
- ta is the time advance function;

The changes are in the functions δ_{ext}, δ_{conf} and λ:

- δ_{ext} is the external transition function, but works on a *bag* X^b instead of a single input. This means that there might be multiple messages on a single port and since a bag is used, the order of the messages in the bag should *not* have an influence on the resulting state.
$$\delta_{ext} : Q \times X^b \to S$$
- δ_{conf} is the newly introduced *confluent transition function*, that gets triggered when there is a *collision* between δ_{int} and δ_{ext} (so when a model receives an external input at the same time as it would do its own internal transition). Most of the time, the user will just want to call δ_{int} or δ_{ext} in some order, though it is possible to define it in any way imaginable, maybe completely deviating from the other transition functions.
$$\delta_{conf} : S \times X^b \to S$$
- λ is the output function, but now it outputs a *bag* instead of a single output. If a single output is desired, a bag with only one element should be sent.
$$\lambda : S \to Y^b$$
Remember that in Classic DEVS, only one internal transition will be ran at a time, so there will be only one message on the output (possibly on multiple output ports) and hence no collisions between messages on a single port happen.
In Parallel DEVS however, we remove this constraint, making multiple messages on a single port possible. To allow this, bags are needed.

1.2.2 Coupled DEVS

The coupled model is similar to the one defined in Classic DEVS, with the main exception that the select function has dissappeared. Therefore, the structure becomes:

$$M = < X, Y, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\} >$$

The dissappearence of the select function means that colliding models should transition in parallel instead of sequential. This is why there is a possibility for parallellisation, since the output function and transition functions can now happen in parallel. The atomic models are responsible for collision handling (multiple input messages, colliding external and internal events, ...).
Again, the models in the coupled model should only be atomic models, but due to *closure under coupling*, it is possible to include other coupled models.

1.2.3 Closure under coupling

As with the previous proof, we have to show that an equivalent atomic model exists for every coupled model.
We wish to transform the coupled model

$$DN = < X, Y, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\} >$$

into an atomic model

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

This will be done in several steps, of which some correspond to the proof from Classic DEVS:

- X and Y stay the same;
- S is defined as the product set of all substates (that is, the states of the atomic models it contains):

$$S = \times Q_i, \text{ where } i \in D$$

- ta is the minimum of all *timeleft* values
  Note that we don't use minimum of ta, since the values the ta returns is relative to the time of the last transition of that specific atomic model, not to the last transition of the coupled model as a whole.

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

- λ outputs the bag of all outputs of the atomic models connected to the coupled model at that specific state
  Note that only the ports that influence the coupled model are used, since the other outputs remain internal to the coupled model and are used in the *transition functions*.

Some definitions of sets of components are necessary to ease the definition of the transition functions:

$$IMM(s) = \{i | \sigma_i = ta(s)\}$$
$$UN(s) = \{i | \sigma_i \neq ta(s)\}$$
$$INF(s) = \{j | j \in \cup_{i \in IMM(s)} I_i\}$$
$$CONF(s) = IMM(s) \cap INF(s)$$
$$INT(s) = IMM(s) - INF(s)$$
$$EXT(s) = INF(s) - IMM(s)$$

And some parallel definitions for the confluent transition function, where the INF' now also contains the coupled model's influencees (within the submodels). This is needed because the coupled model will have received input at this time that has to be routed to the connected subcomponents.

$$INF'(s) = \{j | j \in \cup_{i \in (IMM(s) \cup \{self\}} I_i\}$$
$$CONF'(s) = IMM(s) \cap INF'(s)$$
$$INT'(s) = IMM(s) - INF'(s)$$
$$EXT'(s) = INF'(s) - IMM(s)$$

- δ_{int} is made by performing the relevant transitions on all models, while the other models just have their time elapsed increased. Since the internal transition will generate output, which might have to be consumed internally, we will also have to trigger *external* and *confluent* transition functions.

$$\delta_{int}(s, e, x^b) = (\ldots, (s_i', e_i'), \ldots)$$

$$(s_i', e_i') = \begin{cases} (\delta_{int,i}(s_i), 0) & \text{for } i \in INT(s) \\ (\delta_{ext,i}(s_i, e_i + ta(s), x_i^b), 0) & \text{for } i \in EXT(s) \\ (\delta_{con,i}(s_i, x_i^b), 0) & \text{for } i \in CONF(s) \\ (s_i, e_i + ta(s)) & \text{otherwise } i \in UN(s) \end{cases}$$

The bag is defined as the translation of the output function of all imminent models (so both those with an *internal* and *confluent* transition function)

$$x_i^b = \{Z_{o,i}(\lambda_o(s_o)) | o \in IMM(s) \wedge i \in I_o\}$$

- δ_{ext} is made by performing the external transitions on all models that have received *external* input from the coupled model itself. All other *external* transitions of the models are treated internally at the same time as the internal transition.

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

  The bag is composed of all seperate models' inputs:

$$x_i^b = \{Z_{self,i}(x) | x \in x^b \wedge i \in I_{self}\}$$

- δ_{conf} is basically the same as the δ_{int}, but with the external input bag mixed into the internal input bag. Note that the sets (e.g. INT, EXT, ...) have now been replaced by their counterpart to take into account the models that receive input from the coupled model:

$$\delta_{con}(s,x^b) = (\ldots,(s_i',e_i'),\ldots)$$
$$(s_i',e_i') = \begin{cases} (\delta_{int,i}(s_i),0) & \text{for } i \in INT'(s) \\ (\delta_{ext,i}(s_i,e_i+ta(s),x_i^b),0) & \text{for } i \in EXT'(s) \\ (\delta_{con,i}(s_i,x_i^b),0) & \text{for } i \in CONF'(s) \\ (s_i,e_i+ta(s)) & \text{otherwise} \end{cases}$$

  and the bag is defined as the combination of both bags described above. This is because both external and internal messages are to be processed.

$$x_i^b = \{Z_{o,i}(\lambda_o(s_o)) | o \in IMM(s) \wedge i \in I_o\} \uplus \{Z_{self,i}(x) | x \in x^b \wedge i \in I_{self}\}$$

1.2.4 Example

Due to the close resemblance with Classic DEVS, the previous example can be reused. With the exception that, while in Classic DEVS a collision between an external and internal event would be solved by applying the external one and forgetting the internal one, the confluent transition function gets called to solve this conflict.
E.g.: should the queue system receive a new $EVENT$ at exactly the same time as it was about to trigger its own internal event, the model will call the *confluent transition function*. This function may contain anything, but say that it chooses to execute the internal transition function first and afterwards the external transition function. This way, the event that was in the queue is sent first and the model immediately receives its next $EVENT$ to process.
The use of bags would be interesting in much bigger systems, when multiple EVENTS can be processed concurrently. For example when two events arrive at exactly the same moment (if there were two generators), the queue would receive both of them simultaneously and decide whether or not to drop one of them.

1.2.5 Comparison

Parallel DEVS seems to solve most problems presented by Classic DEVS, though there is not yet any support for dynamic structures. The main advantage is the possibility for parallellisation and the removal of the *select* function. On the other hand, the modeller becomes responsible for the collision handling by defining the *confluent transition function*. The modeller also becomes responsible for managing the bag, since some ports may now contain multiple messages at once.
Even though this new formalism is focussed on performance, there do exist situations where the multi-threaded situation actually slows down simulation speed [17]. A slow-down is likely to happen in the case of *high interaction, low computation* situations, since there is a huge overhead in the thread management (setting up, switching, destroying, ...). In these cases, it would actually be interesting to *not* perform the parallellisation, but revert to a sequential execution.
Even though it might seem that this would nullify the performance improvement compared to Classic DEVS, it is actually still faster, as no *select* function has to be called and no tie-breaking code has to run.

1.2.6 Alternatives

Since parallellism is a rather crucial point in speeding up the simulation speed, there already existed a more parallel version of Classic DEVS, more specifically the *Extended-DEVS* (E-DEVS). This is basically Classic DEVS without the *select function* and the behaviour at collision time is reversed (first internal transition, afterwards external transition). This allows some more parallelism, though it makes some changes that are rather limitting. A more detailed comparison between E-DEVS and PDEVS can be found in [15].

1.3 Dynamic Structure DEVS

Dynamic structures are found in many different kinds of models. One of the most successful solutions to this problem is the Dynamic Structure DEVS (DSDEVS) formalism, proposed by [6]. It basically uses a *network executive* in each coupled model that can receive messages from all of its components to initiate a restructuring. This way, the complete network structure is saved in the network executive as a state. Restructuring is thus reduced to a state change of the *network executive*.

1.3.1 Atomic DEVS

Atomic DEVS models are exactly the same as those defined in the Classic DEVS formalism. This is kind of logical, since an atomic model is strictly *behavioural* and not *structural*.
Though an atomic model is now able to send messages to the network executive to initiate the structural change, so it might be needed to update the *output* list.

1.3.2 Coupled DEVS

Since all structural information is now saved in the network executive, there is no more need for all this data in the coupled model itself. This reduces the coupled model to the structure

$$DSDEVN =< \chi, M_\chi >$$

The model of the network executive is defined as the structure

$$M_\chi =< X_\chi, S_\chi, Y_\chi, \delta_{int_\chi}, \delta_{ext_\chi}, \lambda_\chi, ta_\chi >$$

And since all the *structural* information should be stored somewhere in the network executive, the state S_χ of the network executive is defined as

$$S_\chi = (X, Y, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, \Xi, \theta)$$

With all of them defined as in the coupled model of classic DEVS. θ is used to store other state variables (so the network executive is allowed to store custom data too).
There is an important constraint that is new to DSDEVS (all constraints from Classic DEVS are still valid), namely that the *network executive* should not receive a message that collides with other messages. This constraint is needed to guarantee that there is a deterministic network structure, since the order in which *external transition functions* are processed is indeterministic, making the network structure indeterministic. On the other hand, it is possible to do them at exactly the same time, as long as these messages do not collide (as the network structure is deterministic in this case). This can be done by using intermediate states to seperate these different time slices.

$$Z_{k,\chi}(y) \neq \phi \implies Z_{k,j}(y) = \phi$$
$$\text{for } k \in D \cup \{\Delta\} \text{ and } \forall j \in I_k - \{\chi\}, \text{ with } \phi \equiv \text{null event}$$

With Δ the *dynamic network structure*.
For completeness: to prevent the *network executive* from changing its own structure:

$$\chi \notin D$$

1.3.3 Closure under coupling

The proof is mostly parallel to the one for classic DEVS, with some exceptions when there is a message for the network executive. The *input*, *output*, *state set* and *time advance function* are all defined exactly like in the original proof (see section 1.1.3), only with the addition of the *network executive*:

$$X = X_{\Delta,\chi}$$
$$Y = Y_{\Delta,chi}$$
$$S = \times Q, \forall i \in D_\chi \cup \{\chi\}$$
$$\tau(s) = min\{\sigma_i | i \in D_\chi \cup \{\chi\}\}, s \in S$$

where $\sigma_i = \tau_i(s_i) - e_i$ is the remaining time of submodel i until its internal transition.
The output function is again very similar, only with the addition of the *network executive* which might also output messages (to perform structural changes at a higher level):

$$\lambda(s) = Z_{i^*,\Delta,\chi}(\lambda_{i^*}(s_{i^*})), \Delta \in I_{i^*,\chi}, s \in S$$

where $i^* = \Xi_\chi(\{i | i \in D_\chi \cup \{\chi\}, \sigma_i = \tau(s)\})$ is the component that gets selected from the imminent components to perform its internal transition.
A specific state $s \in S$ has a structure

$$s = ((s_\chi, e_\chi), \ldots, (s_i, e_i), \ldots, (s_{r_1}, e_{r_1}), \ldots, (s_{r_n}, e_{r_n}))$$

which will be used in the transition functions, with s_χ being the state of the *network executive* itself.
First we define the *external transition function*, which has two different (exclusive) possibilities, depending on whether or not the *network executive* received a message.
In the case that no message is received in the *network executive* (if $Z_{\Delta,i,\chi}(x) \neq \phi$):

$$\begin{gathered}
\delta_{ext}(x,s,e) = ((s'_\chi, e_\chi + e), \ldots, (s'_i, 0), \ldots), \forall i \in I_{\Delta,\chi} \\
s'_i = \delta_{ext_i}(Z_{\Delta,i,\chi}(x), s_i, e_i + e) \\
R = \{\} \equiv \text{removed components}
\end{gathered}$$

otherwise (if $Z_{\Delta,\chi,\chi}(x) \neq \phi$):

$$\begin{gathered}
\delta_{ext}(x,s,e) = ((s'_\chi, 0), \ldots, (s_i, e_i + e), \ldots, (s_{a_1}, 0), \ldots, (s_{a_k}, 0)), \forall i \in I_{\Delta,\chi} \\
s'_\chi = \delta_{ext_\chi}(Z_{\Delta,\chi,\chi}(x), (X_\Delta, Y_\Delta, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, \Xi, \theta), e_\chi + e) \\
= (X'_\Delta, Y'_\Delta, D', \{M'_i\}, \{I'_i\}, \{Z'_{i,j}\}, \Xi', \theta') \\
D' = D - R \cup A \\
A = \{a_1, \ldots, a_k\} \equiv \text{added components} \\
R = \{r_1, \ldots, r_n\} \equiv \text{removed components}
\end{gathered}$$

As was seen in one of the constraints, the *network executive* cannot receive a message that is colliding with another message, to guarantee determinism and avoid ambiguousness about which structure is currently in effect. Consequently, it becomes clear that both branches cannot be possible at the same time.
Now only the *internal transition function* remains:

$$\delta_{int}(s) = \delta^*_{ext}(i^*, y^*, \delta^*_{int}(i^*, s))$$

where the help functions δ^*_{ext} and δ^*_{int} were used, which reflect the effect of respectively the internal and external function of the imminent component on the total state. They are defined next.
We will also need the imminent component i^* and the output of this component y^*:

$$\begin{gathered}
i^* = \Xi_\chi(\{i | i \in D_\chi \cup \{\chi\}, \tau_i(s_i) - e_i = \tau(s)\}) \\
y^* = \lambda_{i^*}(s_{i^*})
\end{gathered}$$

We define the δ^*_{int} first. Again two (exclusive) branches are possible:
If the *network executive* doesn't make an internal transition ($i^* \neq \chi$), we just call the δ_{int} function of the imminent component:

$$\delta^*_{int}(i^*, s) = ((s_\chi, e_\chi + \tau(s)), \ldots, (\delta_{int_{i^*}(s_{i^*})}, 0), \ldots), \text{ with } R = \{\}$$

or when the *network executive* makes an internal transition ($i^* = \chi$), we update its state and thus update the network structure:

$$\begin{gathered}
\delta^*_{int}(i^*, s) = ((s'_\chi, 0), \ldots, (s_i, e_i + \tau(s)), \ldots, (s_{a_1}, 0), \ldots, (s_{a_k}, 0)) \\
s'_\chi = \delta_{int_\chi}((X_\Delta, Y_\Delta, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, \Xi, \theta)) \\
= (X'_\Delta, Y'_\Delta, D', \{M'_i\}, \{I'_i\}, \{Z'_{i,j}\}, \Xi', \theta') \\
D' = D - \{r_1, \ldots, r_n\} \cup \{a_1, \ldots, a_k\}
\end{gathered}$$

For the δ^*_{ext} we proceed likewise:
In the case that the *network executive* doesn't receive an external input from the coupled model itself (*not* from the internal submodels), ($Z_{i^*,i,\chi}(y^*) \neq \phi$):

$$\begin{gathered}
\delta^*_{ext}(i^*, y^*, s) = ((s_\chi, e_\chi + \tau(s)), \ldots, (s'_i, 0), \ldots), \forall i \in I_{i^*,\chi} \\
s'_i = \delta_{ext_i}(Z_{i^*,\chi}(y^*), s_i, e_i + \tau(s)) \\
R = \{\}
\end{gathered}$$

or when the *network executive* receives external input ($Z_{i^*,\chi,\chi}(y^*) \neq \phi$) and thus updates the network structure:

$$\delta^*_{ext}(i^*, y^*, s) = ((s_\chi, 0), \ldots, (s_i, e_i + \tau(s)), \ldots, (s_{a_1}, 0), \ldots, (s_{a_k}, 0)), \text{ if } \chi \in I_{i^*,\chi}$$
$$s'_\chi = \delta_{ext_\chi}(x, (X_\Delta, Y_\Delta, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, \Xi, \theta), e_\chi + \tau(s))$$
$$= (X'_\Delta, Y'_\Delta, D', \{M'_i\}, \{I'_i\}, \{Z'_{i,j}\}, \Xi', \theta')$$
$$x = Z_{i^*,\chi,\chi}(y^*)$$
$$D' = D - \{r_1, \ldots, r_n\} \cup \{a_1, \ldots, a_k\}$$

1.3.4 Example

Imagine a model as in figure 1.1, where a machine either sends its finished product to machine *a* or machine *b*. It will send the product to machine *a* by default, but if this gets broken, machine *b* should be used as backup.
All of this should happen transparantly to the first machine. A possible solution without DSDEVS would be to give machine *a* an output port to pass on the messages to machine *b* in case it is broken. An other possibility would be to introduce an extra model as a coordinator, by having this model communicate with machine *a* and *b*.
It is clear that these solutions might work in this case, but quickly become unmanageable in large situations. Either because of artificiality (the real-world machine *a* probably doesn't have a 'pass' port), or because of performance (introducing an extra component).
DSDEVS provides the solution, as it allows machine *a* to send a message to the *network executive*, which will relink the machine with machine *b*. The first machine doesn't get hindered by this and it has an ideal performance, since the redirection should only happen once. Furthermore, machine *a* is now completely uncoupled and it would even be possible to actually remove it from the simulation (as being 'in repair').
Note that it is not allowed for both the *network executive* and another component to receive a message at exactly the same moment (in an *external* transition), so we will need an intermediate state to wait before sending the real message (the product).

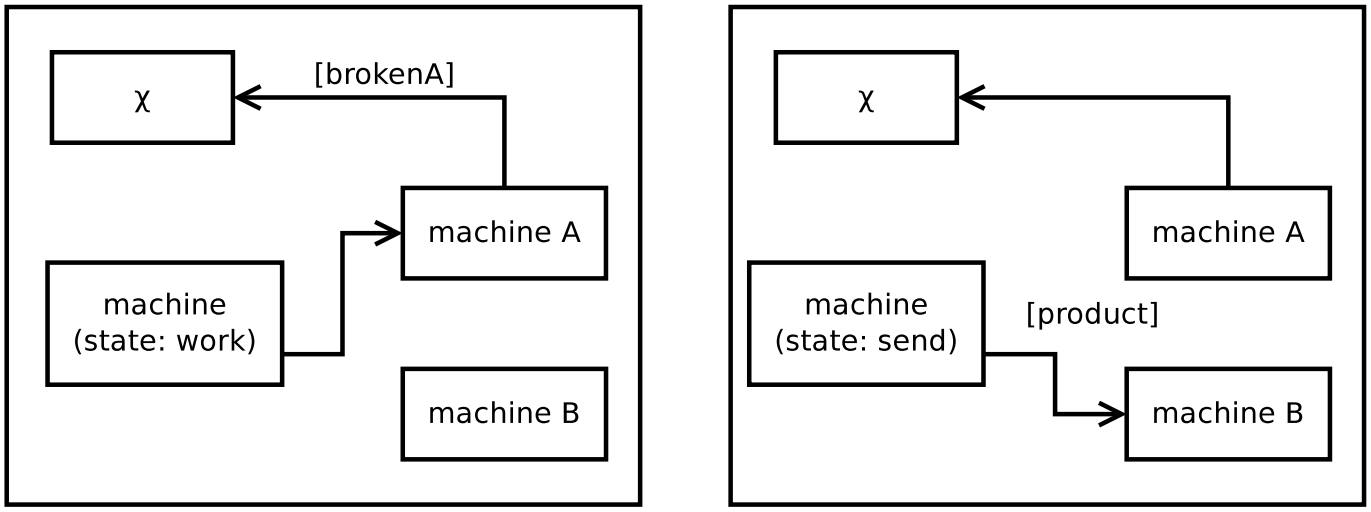

Figure 1.1: The example represented visually. First *machine A* sends a message to the *network executive* and afterwards all output from the first machine will be routed to *machine B* instead of *machine A*. Note that not all connections to the *network executive* are drawn for simplicity.

1.3.5 Comparison

The DSDEVS formalism only requires relatively small changes to a working Classic DEVS simulator, so this formalism is often supported in many simulators. Though this functionality is often offered, DSDEVS does imply some limitations on the models. For example, a model cannot send a message to both the *network executive* and another model at the same time. This is to prevent ambiguity in the execution order.
Noteworthy is that it is shown that every DSDEVS model can be translated into an equivalent Classic DEVS model. For this reason, DSDEVS is not really an extension but just a simplifying formalism to do these kind of changes.

1.3.6 Alternatives

As already mentioned, the need for dynamic structures arises in many situations, so it is not surprising that there also exist various formalisms that all support this kind of models. There are some slight differences between the different implementations, though they achieve practically the same. Those that will be mentioned later are:

- **DynDEVS** [35] offers a completely reflective approach. In DSDEVS, models cannot change themselves as they have to mention this to the network executive. Besides, the network executive cannot change itself, since it is not an element of its own D. In DynDEVS, every model can change itself and its structure. It is implemented in a.o. ADEVS;
- **DSDE** [8, 7] is a combination of Parallel DEVS with DSDEVS. The main advantage is clearly the addition of parallelism. It is implemented in a.o. VLE.
- A combination of the two above is mentioned in [31]

1.4 Cell DEVS

Cell DEVS was introduced by [38] in an attempt to combine DEVS and Cellular Automata. Cellular Automata had the downside that they worked on a discrete time base, making it very resource intensive to increase the granularity of the simulation. DEVS had the main downside that it wasn't really specialised for the case where many identical atomic models are connected to each other and therefore was rather difficult to be used. The user would have to manually connect all these atomic models together. In Cell DEVS, the DEVS formalism is used but its usage resembles the simplicity of Cellular Automata. This combines the best of two worlds: the high granularity at reasonable performance of DEVS with the ease of use (in case of cellular models) of Cellular Automata.
As will be seen in the example, the use of Cell DEVS doesn't resemble classic DEVS at all, though it is used at the core.

1.4.1 Atomic DEVS

An atomic model is a structure

$$TDC = < X, Y, S, N, delay, d, \delta_{int}, \delta_{ext}, \tau, \lambda, ta >$$

- Elements from DEVS:
  - X the set of external input events;
  - Y the set of external output events;
  - S the set of sequential states;
  - δ_{int} the internal transition function;
  - δ_{ext} the external transition function;
  - λ the output function;
  - ta the time advance function;
- New elements:
  - N the set of input events (from other cells);
  - *delay* the kind of delay to use for the cell (**inertial** or **transport**);
    - *inertial* delay means that the delay signifies the time that is being waited *before* the input is processed. This means that no new input must arrive in the delay period, or the previous one will be removed and have had no influence on the state. Should an input arrive, the previous input is ignored and no changes to the state will have occured. Sending information about the state change happens instantly.
    - *transport* delay means that it takes some time for the state information to travel to other cells, but this information is immediately processed on arrival. Every input will be able to produce a change to the state and the neighbours of this cell are guaranteed to receive this new state, so it might be that multiple state messages are 'on their way' (note that the *instant message passing* from DEVS still applies, so messages are stored internally in the cell).
  - d defines the delay length of the previously mentioned type;
  - τ the local computation function, which determines what should happen with the received values;

Since these are mostly the same as in the Classic DEVS formalism, they will not be explained seperatly. Some are new, but their meaning is very straightforward from the definition, so no further explanation should be needed.
Note that, as can be seen in the example, most the DEVS related elements are constructed based on the new elements. So there is no need for the user to define e.g. the *internal transition function*.

1.4.2 Coupled DEVS

A coupled model is a structure

$$GCC = < X_{list}, Y_{list}, X, Y, n, \{t_i\}, N, C, B, Z >$$

- X_{list} is the input coupling list;
- Y_{list} is the output coupling list;
- X is the set of external input events;
- Y is the set of external output events;
- n is the dimension of the cell space;
- t_i is the number of cells in the i-th dimension, defined $\forall 0 < i \leq n$;
- N is the neighbourhood set;
- C is the cell space state set;
- B is the set of border cells, these will be treated seperately, since they might have other behaviour (to prevent the simulation from going out of bounds). It is also possible to define a *wrapped* simulation, where there are no borders;
- Z is the translation function;

1.4.3 Closure under coupling

Cell-DEVS doesn't extend the classic DEVS formalism, but only provides an 'interface' to DEVS as seen from Cellular Automata. This means that a proof for closure under coupling is not needed since the final result will just be a normal (classic or parallel) DEVS model.

1.4.4 Example

Since Cell DEVS doesn't allow the user to directly manipulate the transition functions, we only have to define a local computation function and a delay type. For this, we will use the notation of CD++ as this is the 'leading' implementation of Cell DEVS. More examples of this kind can be found in [40], [5], [23] and [42].
The easiest and likely most understandable example would be an implementation of *the game of life*. This is easily done in Cell DEVS as follows:

1. *Define the delay type*
   The delay type here is obviously transport, since we want to have some kind of *propagation delay* for the new values of the cells. Note though, that inertial would have also been possible, since all cells always have exactly the same time advance, so it doesn't really matter.

2. *Define the delay time*
   Time doesn't really matter here, so any value will do just fine.

3. *Define the borders*
   Since we want the cells to wrap around the borders, we choose *wrapped* for the borders.

4. *Define the input/output*
   Our game of life will not have any special input or output, it will run autonomous.

5. *Define neighbours*
   Since Cell DEVS allows a very flexible approach, the user must specify the neighbours of a given cell. All addresses are relative to the cell at the origin, the exact coordinates vary depending on the dimension. For our game of life, we will just use a two dimensional grid, so the origin would be $(0,0)$. The neighbours of a cell are all surrounding cells, so the list of neighbours would become

   $$[(-1,-1),(-1,0),(-1,1),(0,-1),(0,0),(0,1),(1,-1),(1,0),(1,1)]$$

   Note that the cell $(0,0)$ is also part of the neighbour list, this is because otherwise there would be no way to access the own value (at least in CD++).

6. *Define the local computation function*
   Now all that is left is to define the rules to be used for the cells. Note that the original specifications are somewhat harder to implement, so they were rewritten to the following:

   - the cell *remains* active if the number of active neighbours is 2 or 3
   - the cell will *become* active when the active neighbours is exactly 3
   - otherwise, the cell will *become or remain* inactive

   In CD++, a rule has the form "newstate delay {precondition}". If the *precondition* is true, the state will be changed to *newstate* with a delay of *delay*.
   This causes the following rules (the keyword *truecount* in CD++ returns the number of cells in the neighbourhood with a state equal to *true*):

- 1 10 {(0,0) = 1 and (truecount = 3 or truecount = 4)}
  So the cell will become **active** (1) with a delay of 10. The precondition is that the cell is already alive ($(0,0) = 1$) and there are 3 or 4 active neighbours ($truecount = 3$ or $truecount = 4$). Note that this number of neighbours is 1 higher than those in the specification, this is because the cell itself is also in the neighbourhood and is alive as well.
- 1 10 {(0,0) = 0 and truecount = 3}
  So the cell will become **active** (1) with a delay of 10. The precondition is that the cell is not alive ($(0,0) = 0$) and there are exactly 3 active neighbours ($truecount = 3$).
- 0 10 {t}
  So the cell will become **inactive** (0) with a delay of 10. The precondition is always true (t), since this is the catch-all case.

1.4.5 Comparison

Cell DEVS is mainly interesting in natural science, where the use of Cellullar Automata might be considered. Another advantage of Cell DEVS is that no deep knowledge of DEVS is required, since the Cell DEVS formalism doesn't require the user to implement any DEVS related transition functions manually. The user is only required to define a local computation function and a time delay, allowing non-computer scientists to use the formalism with ease. The Cell DEVS models basically get translated to equivalent classic DEVS models.
An important reason for the introduction of the Cell DEVS formalism was the easier development and maintenance of models. It can be seen in [40] that this goal has succeeded, though the simulation results are somewhat dissapointing due to the low performance compared to manual implementation.
In [23] it is compared to other approaches and a mention of *Quantized DEVS* is made.
There also exists a parallel versions of Cell DEVS, Parallel Cell DEVS, described in [34].

1.5 Conclusion

We discussed several different DEVS formalisms, starting with the original *Classic DEVS* and its revision, *Parallel DEVS*, which allows for more parallelism and removes the artificial *select* function.
Further extensions to the DEVS formalism include *Dynamic Structure DEVS* and *Cell-DEVS*, which both offer extra possibilities for increasing the speed of simulation, either by allowing the modeller to remove components from the simulation at run-time (*DSDEVS*), or by introducing some extra knowledge about the structure of the models (*Cell-DEVS*).
For all of these, we showed their formal basis for both the *Atomic model* and *Coupled model*, including a proof of *closure under coupling* which is needed to allow hierarchical structures. To illustrate the main powers of these formalisms, an easy to understand example is included. At the end, these formalisms are compared to the basic formalism.

2 Comparison of DEVS simulators

While there are many DEVS formalisms that sometimes only differ in the details, there might be even more DEVS simulators. Each DEVS simulator will have some specific feature that makes it more fit in certain problem domains, just like the formalisms they implement.
Since the DEVS formalism is described with an *abstract simulator*, there is a lot of freedom and room for performance improvement. Though the result should still conform to the result of the *abstract simulator*, otherwise the simulator does not implement that formalism but a mere *dialect*. This is still a problem in some simulators, as they deviate from the formalism in some ways, maybe even without knowing!
Classic DEVS and Parallel DEVS are the basic versions of DEVS and most simulators will at least implement one of these versions. In this chapter, several recent simulators will be compared based on some criteria like features, compliance, performance, ...

2.1 Simulators

First of all, we will start by introducing the different simulators that are compared to each others. For each of them, we will mention the basic info (name, developers, programming language, ...), the first impression of the simulator, ...
We will also include a screenshot or (part of) the output of a simulation run, for easy reference when comparing the simulators.

2.1.1 ADEVS

ADEVS[24] is a simulator primarily focussed on performance and lightweightness. The simulator is written in C++ and extensively uses templates.
There is no seperate simulation program, as everything about the simulator and the model is just built in *one* final executable. A seperate simulator would be useless, since the simulator is specialized to the models due to the extensive use of templates. The usage of templates also allows for several compile-time optimisations [33]. Sadly, the usage of templates is not completely supported by all compilers, limiting the simulator's code structure (all simulator code in header files) and portability to other compilers.
ADEVS was developed by Jim Nutaro [24] and the version that was used in this comparsion is version 2.6. An example output of the simulator is rather short, since ADEVS itself doesn't have any output whatsoever, neither does it have a *verbose* option. All output that is visible is from our own *print* statements:

```
avg_avg_v_pref_devs = 5.44437
avg_transit_times = 2.06106
```

Listing 2.1: Simulation output of ADEVS

```
N-CD++: A Tool to Implement n-Dimensional Cell-DEVS models
-----------------------------------------------------------

Version 2.0-R.45 December-1999
Daniel Rodriguez, Gabriel Wainer, Amir Barylko, Jorge Beyoglonian
Departamento de Computacion. Facultad de Ciencias Exactas y Naturales.
Universidad de Buenos Aires. Argentina.

Loading models from roads.MA
Loading events from
Message log: /dev/null
Output to: /dev/null
Tolerance set to: 1e-08
Configuration to show real numbers: Width = 12 - Precision = 5
Quantum: Not used
Evaluate Debug Mode = OFF
Flat Cell Debug Mode = OFF
Debug Cell Rules Mode = OFF
Temporary File created by Preprocessor = /tmp/fileRrmGsQ
Printing parser information = OFF

Starting simulation. Stop at time: Infinity.
Simulation ended!
```

Listing 2.2: Simulation output of CD++

2.1.2 CD++

CD++[39] is written in C++ and is focused on the Cell-DEVS formalism. Models can be written in either C++ or in a custom language (Cell-DEVS and coupled models only). CD++ comes in several versions, including a parallel version. The choice was made to use the original version, since we also discussed the classic version of Cell-DEVS previously.
This simulator was introduced together with the Cell-DEVS formalism [38], which is basically a combination of Cellular automata and DEVS. This fusion is mainly used in natural science, with several applications in [42], [30], [40], [23] and [5].
There are also many extensions to CD++, like CD++Builder (an IDE), but only the simulator itself (CD++) was tested. The tested version is version *2.0-R.45*. Note however, that this version could not be compiled using the same GCC-compiler[1] as all other simulators, so the source code was slightly altered (added includes, namespaces, ...) to make it compilable. No negative effects of this slight modification were visible during testing.
CD++ was developed by Daniel Rodriguez, Gabriel Wainer, Amir Barylko and Jorge Beyoglonian at the Universidad de Buenos Aires [39]. An example output of the simulator:

2.1.3 DEVS-Suite

DEVS-Suite[1] is written in Java, like the models, and is the successor of DEVSJava. DEVS-Suite contains an additional Simulation Viewer and a tracking environment. Since it is written in Java, there is a much higher portability for both the simulator and the generated models.
It was developed by ACIMS/ASU. The version that was used in the comparison is version 2.1.0.

2.1.4 MS4 Me

MS4 Me is based on the Eclipse framework and closed-source. This simulator is written in Java and contains many features, like a GUI, IDE, Simulation viewer, a custom natural language, ... However, several important features are still unavailable to perform a complete test on it, like hierarchical coupled models.
MS4 Me is developed by RTSync [29] and is the only commercial (and closed source) simulator in this comparison. Since the simulator is not yet *stable* at the moment of this writing, *beta* version 1.0.1 is used.

[1] CD++ requires GCC 2.95.3-5, but all other simulators are compiled using GCC 4.5.4

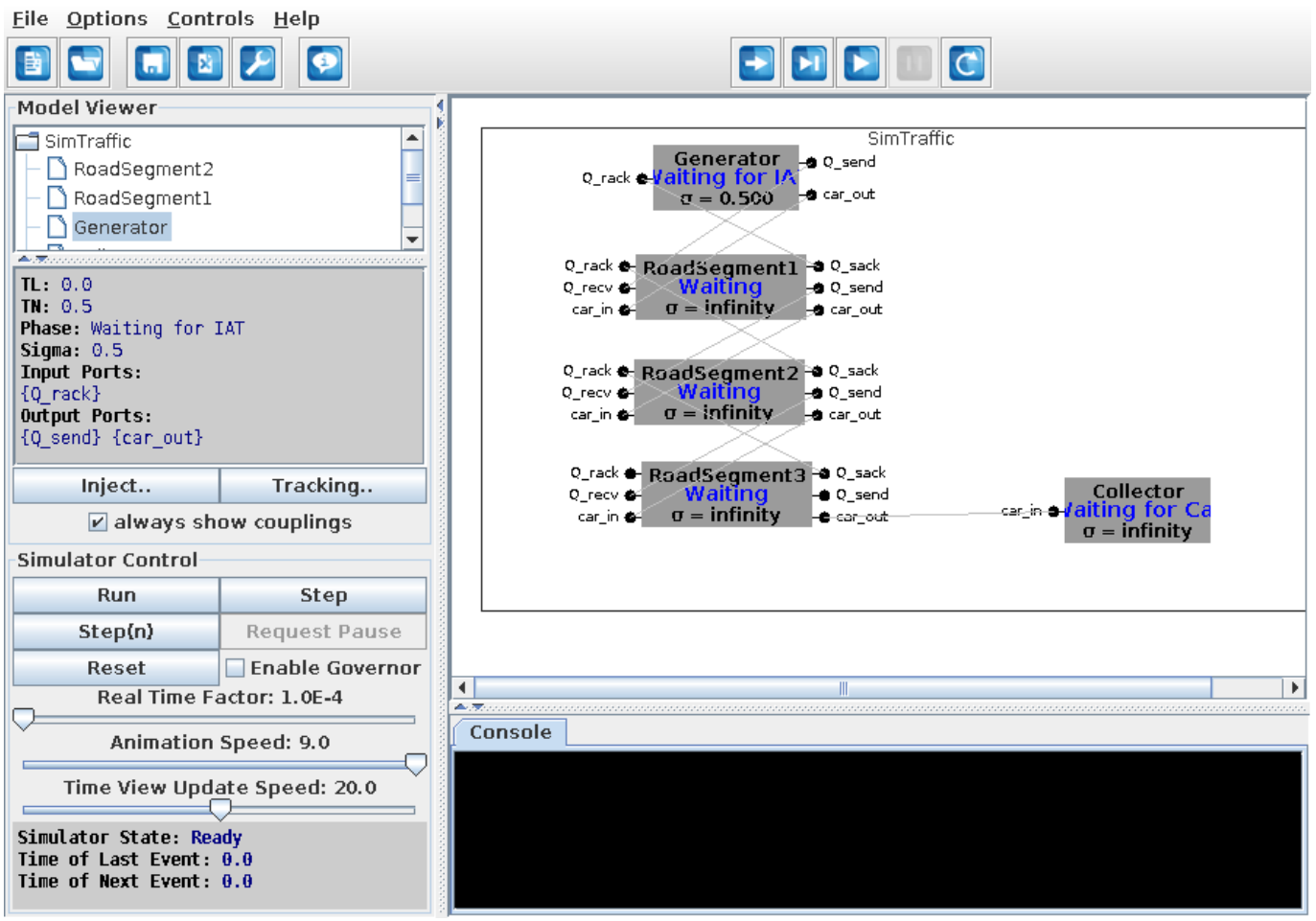

Figure 2.1: DEVS-Suite: Simviewer

2.1.5 PythonDEVS

PythonDEVS[2] is a very basic simulator written in Python (version 2), just like the models. Compared to the other simulators, PythonDEVS has a very small code base and is extremely lightweight.
This simulator is developed at MSDL. Note that this simulator is used as the basis for the optimisations described in chapter 3, though the version used for this comparison is the basic version (version 1.0.2). In some cases, there might be a reference to the modified version. A segment of the output of the simulator is given below (in *verbose* mode). Note that PythonDEVS also has the possibility to make XML traces (if supported by the models) and VCD traces (if applicable to the messages).

2.1.6 VLE

VLE[3] is a relatively feature-rich simulator written in C++ and developed by ULCO and INRA. It includes a GUI and an IDE (in GVLE). Its main focus lies with multi-paradigm modelling.
One of the most interesting features of GVLE is the possibility to *draw* coupled models instead of coding them.
The version used is version 1.0.3. An example output of VLE is:

2.1.7 XSY

XSY[18] is a simulator written in Python (version 2), just like the models. The main feature of this simulator is the verification engine.
The simulator was developed by Moon Ho Hwang and the used version is version 1.0.0. A segment of the output:

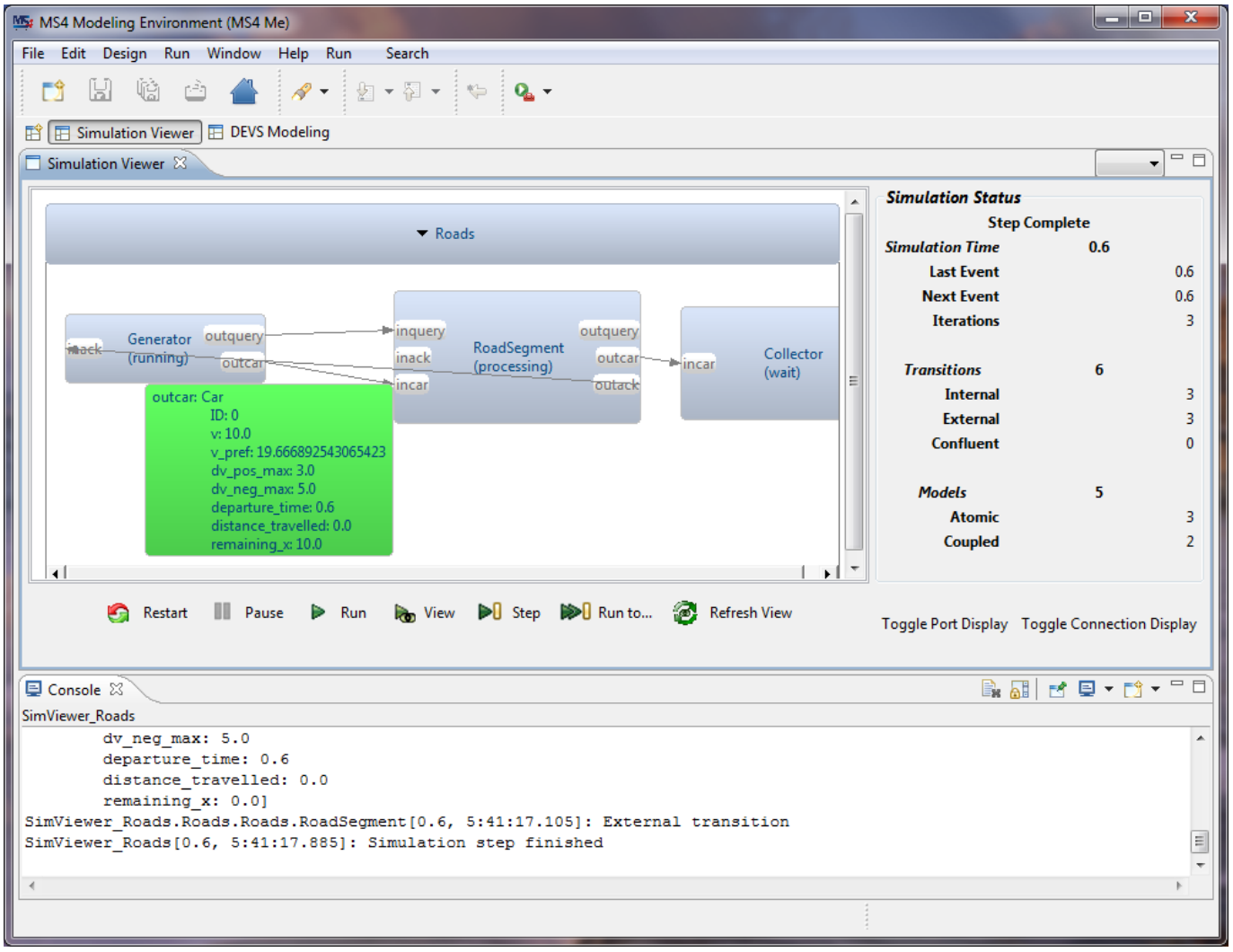

Figure 2.2: MS4 Me: Simviewer

2.2 Specification

To be able to make a decent comparison between the different simulators, a specific model was implemented in each of them. This model is described in detail in [37]. The main use of this model was to gain an understanding of the different simulators (which functions to use, how to run them, how to debug, ...), but afterwards the model was also executed to gain a basic understanding of the performance of the simulator.
The model requires several parameters, to allow easy scaling of the model. For consistency, the variables that determine the minimum and maximum values of a random range are taken to be exactly equal, effectively disabling the random component. This is necessary to prevent fluctuations in the simulation, since timings would otherwise be implementation language dependend. For clarity, the specification will be discussed here without going too deep in the technical details. The simulation time is taken to be 1000 (simulated time).
The specification consists of a connected road with a generator at the first piece of the road and a collector at the end of the road. Cars are transfered over the *car*-ports, which are checked for collisions based on a *query* and *query acknowledgment*.

2.2.1 Generator

The *generator* first waits for its interarrival time to expire, sends a *query* and waits for a response (a *queryAck*). After this response is received, the car is send based on the data in the response. The generator is also responsible to give each car a (randomly) preferred speed and an initial velocity.

```
--  Current Time:       99.98 ----------------------------------------


        INTERNAL TRANSITION in model <Root.RoadSegment2>
          New State: Road segment: cars_present = [<trafficModels.Car instance at 0
              x7f41df14b440>], send_query_delay = +INFINITY, send_ack_delay = +
              INFINITY, send_car_delay = 0.769230769231
          Output Port Configuration:
            port <Q_send>: Query: ID = 72
            port1: NoEvent
            port2: NoEvent
          Next scheduled internal transition at time 100.7461538461538

        EXTERNAL TRANSITION in model <Root.RoadSegment3>
          Input Port Configuration:
            port <Q_recv>: Query: ID = 72
            port <car_in>: None
            port <Q_rack>: None
          New State: Road segment: cars_present = [], send_query_delay = +INFINITY,
              send_ack_delay = 0.1, send_car_delay = +INFINITY
          Next scheduled internal transition at time 100.07692307692302
Average deviation from prefered velocity: 5.44436668999
Average transit time: 2.06105769231
```

Listing 2.3: Simulation output of PyDEVS

2.2.2 Road segment

The *road segment* is the main part of the model and has to reply to *queries* (with a delay *observ_delay* and sends the time it will take to clear this road segment). After it receives a *car*, it immediately sends a *query* to the next part and sends the *car* as soon as either 1) the *car* was originally scheduled to leave or 2) depending on the response received on the *query*. *Cars* can collide if more than one *car* is at the same *road segment*, though this is avoided in the test cases (as it would short-circuit the simulation).

2.2.3 Collector

The *collector* receives *cars*, but doesn't receive *queries* (and thus doesn't respond to them). After a *car* is received, the transit time and the average deviation from the prefered velocity is saved. After the simulation has ended, the average of these two values is calculated and shown[2].

2.3 Criteria

2.3.1 Formalism

For each simulator, its *main* formalism will be mentioned. This is the formalism that will be most likely used and/or is best documented in its manual. This will most likely be one of the formalisms mentioned in the previous chapters. Most simulators still support other formalisms too, like *Real-time DEVS*, *Stochastic DEVS*, ... though these are not as frequent as those discussed here.

2.3.2 Features

Some simulators offer some (optional) features that don't really alter the simulation in a big way, but might make the task of the modeller easier. Some of these features might include the presence of a GUI, IDE, package repository, ... Others include the support for distributed and/or parallel simulation, which influences the simulation performance. Many simulators support plug-ins or a built-upon GUI and/or IDE, which makes it clear that this doesn't really have an impact on the simulation itself as it is seperated. It should be noted that some features could have an interesting impact on the simulator itself, like a model repository. This might include some 'expert build' models for frequently required models that can increase the simulation speed,

[2] Some simulators don't have the possibility to perform code *after* the simulation. In these cases, the average will be recalculated each time a new *car* arrives. In the performance evaluation, all average calculations are commented to provide a fair comparison.

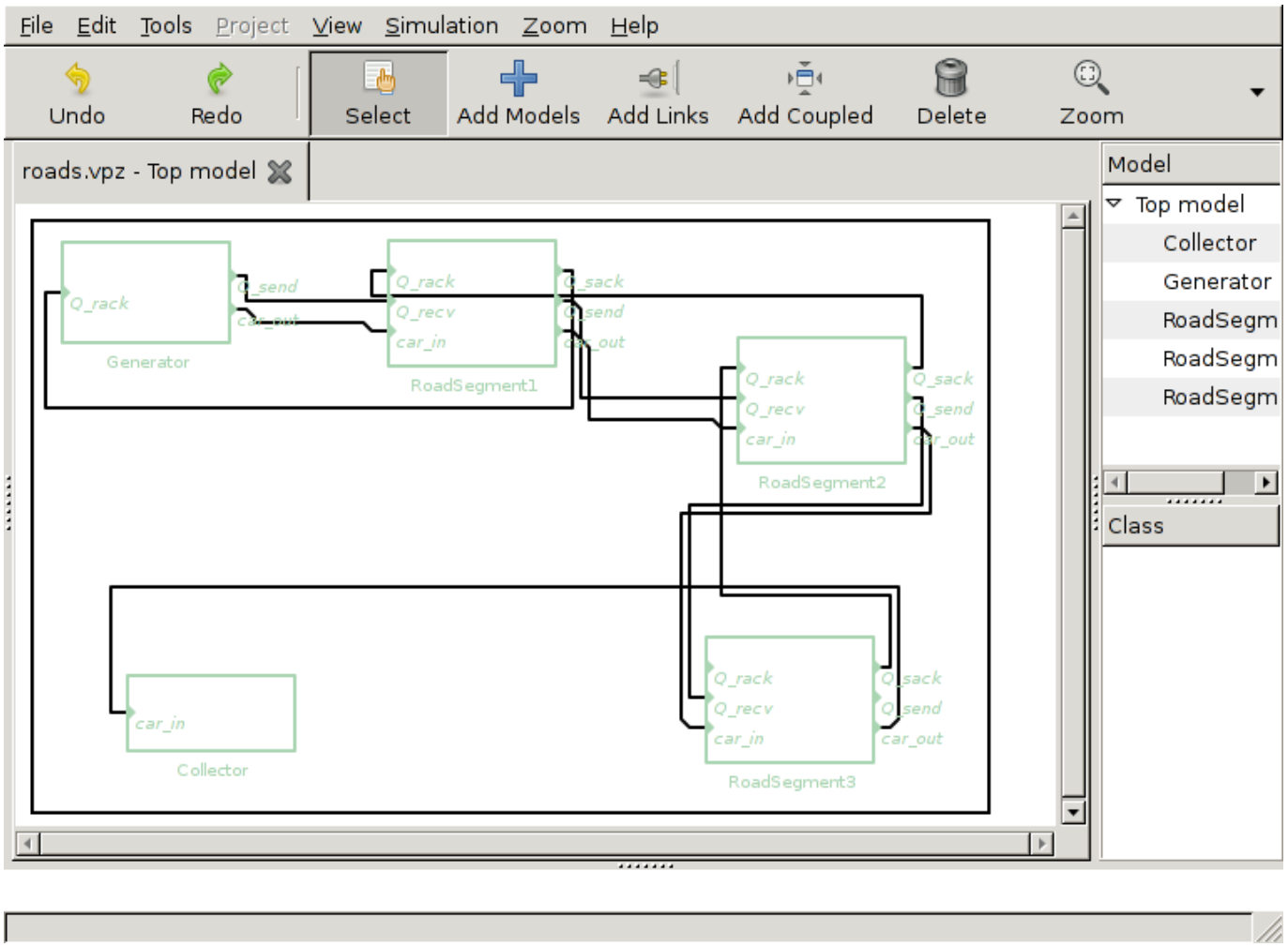

Figure 2.3: GVLE: Visual modelling

comparable to the STL in C++. Another might be the use of a GUI or custom language that can generate very efficient code for the simulator.
To avoid confusion, we will clearly mention our distinction between IDE and GUI:

- an *IDE* provides the user with a (graphical) environment which aids the user in modelling, be it with a source code editor, graphical modelling environment, ...
- a *GUI* provides a graphical user interface that *displays* the situation or at least provides an easy graphical interface to start the simulation.

2.3.3 DEVS compliance

DEVS compliance is one of the most important points in a simulator, as a non-compliant simulator would be prone to modelling bugs, most likely caused by a wrong understanding of the modelling language. However, a simulator that is non-compliant most likely contains a bug (unless stated otherwise, of course). Another problem with a non-compliant simulator is that model exchange between different simulators becomes more difficult, due to the different behaviour of the simulators.
Many of these points were already discussed in [22], but we add some extra points as to also support parallel DEVS (by removing the test about the *select* function). Furthermore, the comparison is done on other simulators.
A list of the characteristics that have to be tested is:

1. Non-negativeness of the time advance function
2. Correct output function
3. Event passing instantaneity

```
JustRun: run 1 simples simulations in one thread
[roads.vpz]
 - Open file ..................: ok
 - Coordinator load models ....: ok
 - Clean project file .........: ok
 - Coordinator initializing ...: ok
 - Simulation run .............:
   0%   10   20   30   40   50   60   70   80   90   100%
   |----|----|----|----|----|----|----|----|----|----|
   **************************************************
 - Coordinator cleaning .......: 10000         5.4443666899926 2.06105769230767
timeTop model:Collector.pref_dev          Top model:Collector.transit_time
ok
 - Time spent in kernel .......: 5.34s
```

Listing 2.4: Simulation output of VLE

```
-- the current state --
(root:((Generator:tL=999.369,tN=1000.138), (RoadSegment1:tL=999.562,tN=inf), (
    RoadSegment2:tL=999.562,tN=1000.331), (RoadSegment3:tL=999.562,tN=999.662), (
    Collector:tL=999.461,tN=inf)),tL=999.562,tN=999.662)  at time 999.562.

(root:((Generator:tL=999.369,tN=1000.138), (RoadSegment1:tL=999.562,tN=inf), (
    RoadSegment2:tL=999.562,tN=1000.331), (RoadSegment3:tL=999.562,tN=999.662), (
    Collector:tL=999.461,tN=inf)),tL=999.562,tN=999.662)  at time 999.662.
        -- (,999.662) -->
-- the current state --
(root:((Generator:tL=999.369,tN=1000.138), (RoadSegment1:tL=999.562,tN=inf), (
    RoadSegment2:tL=999.662,tN=1000.205), (RoadSegment3:tL=999.662,tN=inf), (Collector
    :tL=999.461,tN=inf)),tL=999.662,tN=1000.138)      at time 999.662.

Simulation run time is bounded by 1000.000.

-- the current state --
(root:((Generator:tL=999.369,tN=1000.138), (RoadSegment1:tL=999.562,tN=inf), (
    RoadSegment2:tL=999.662,tN=1000.205), (RoadSegment3:tL=999.662,tN=inf), (Collector
    :tL=999.461,tN=inf)),tL=999.662,tN=1000.138)      at time 1000.000.

-- simulation menu --
e(x)it, (in)jet, (m)ode, (p)ause, reset, run, stat(i)stics>
```

Listing 2.5: Simulation output of XSY

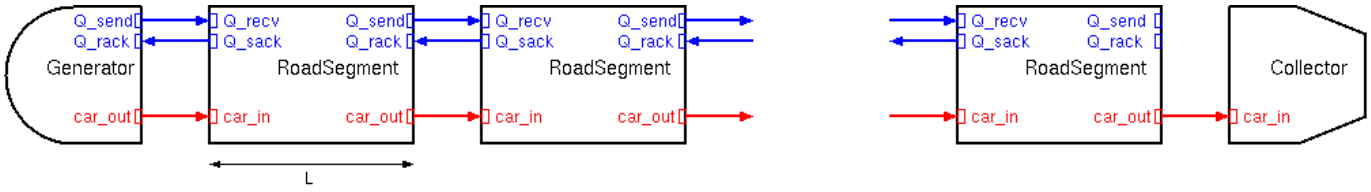

Figure 2.4: The desired result

4. Precise time granularity
5. Correct event sequence
6. Correct event dispatching
7. Event independence

And the ones added here, that are dependent on the underlying formalism:

8. Bag use (PDEVS-only)
   Do the bags get used properly. This has two sides: 1) do multiple messages come together in the bag, 2) does the bag allow the presence of multiple (identical) messages. Luckily, both of them can be tested together, since the second implies the first.
9. Confluent transition (PDEVS-only)
   Does the confluent transition function get called when an external and internal transition function collide, is it possible to modify this function?
10. Select function (CDEVS-only)
    In classic DEVS, the select function should be called when a collision occurs, so does this function really get called (e.g. is it possible for the user to define one)?

Besides these tests that must happen at run-time, some attention is given to the function interfaces. Does only the required information gets passed to the models, or do some extra parameters get passed? E.g. does the internal transition function receive messages from somewhere, does the time advance function have access to unnecessary data, ... This is partly linked with DEVS enforcement, since it might protect the user. Though it should be more with compliance, since the functions themselves should comply to the DEVS formalism.

2.3.4 DEVS enforcement

Even if the simulator is *DEVS compliant*, the modeller might (unknowingly) define some *non-compliant* models. The most easy mistake to make is changing the state in the *output function* (or *time advance*). Programming tricks to communicate with other models is another of these illegal constructs. Even though these violations are caused by the modeller, beginning DEVS-users might not have a firm understanding of the formalism and make some violations. This might cause unexpected results and possibly even a different behaviour when a new version of the simulator is released, as behaviour outside the formalism isn't guaranteed.
Due to the helpful nature of this feature, this might even be paired with debugging.
The main difference between enforcement and compliance is with the responsibilities. The author of the simulator is required to make a DEVS-compliant simulator, while the modeller is required to make his models DEVS-compliant too. To aid the modeller in this task, the simulator might use some kind of enforcement. Most parts of enforcement would be outside the scope of a simulator, though the interface that is defined might be of help. The perfect example of enforcement is of course a syntax-directed modeling environment, though this is no longer a simulator but a modelling environment.

2.3.5 Debugging

The debugging possibilities of a simulator might be affected by the amount of (optionally) verbose output. Certainly beginning modellers will find it easy to know what is going on. Besides this, it might also be interesting to have a good understanding of the model, by e.g. viewing the lay-out of the model. This could easily avoid some errors like those found in [20] and [41].
An additional feature that is closely related is the availability of a verification engine, or an integrated testing harness. Fast change/test cycles are also important, which hugely favors the scripting languages over the compiled languages.

2.3.6 Performance

Performance is a rather important part of the evaluation of a simulator. However, performance is very dependent on the type of model that is used. An extensive comparison of the performance of these simulators is therefore out of the scope of this document. The interested reader is refered to [16]. The performance will be compared on a very basic model, more specifically the *traffic roads* model as described in 2.2, this is the model that was used to get to know the simulator. This means that these results should not be taken as absolute results and should only be used as a general idea.
Besides this small model, some bigger (though artificial) models are also used to check for performance. These models have an exceptionally large characteristic, like thousands of components, hundreds of hierarchy levels, ...
Note that several features of the simulator are not used, like distributed or parallel simulation, due to the fact that this is not really necessary in our small test case. Another reason would be that there should be a comparsion between the simulation algorithms themselves (as is the topic of this research internship) and not between some supplemental features.
The tests are performed with an as lightweight interface as possible. If there is an optional GUI, this will not be used during the simulation run. However, some tools require a certain GUI, so this will have a certain impact on the simulation. Some environments don't provide an easy interface to record the (wall clock) time it takes to run the simulation, in which case the time that it takes to performn the manual operations is also taken into account. Luckily, these cases are already in the range of several seconds, rendering this unfortunate inaccuracy less worse.

2.3.7 Interactivity

Interactivity might encompass a lot of different features, like the ability to pause/continue the simulation run, to inject data (e.g. at each step or at some time), to have different halting conditions or even to view the simulation as it runs.
Interactivity of the simulation is closely linked to the debugging capability, as this might help in debugging. However, interactivity might have a negative effect on performance (due to the constant polling/drawing).

2.3.8 Scheduler

While the scheduler is a rather technical detail and should not really be seen as a feature of a simulator, the performance is very dependent on the used scheduler. This also (likely) defines the complexity of the simulator, so the scheduler should be taken into account when deciding on which simulator to use. Some examples of schedulers include a scanning list, a sorted list, a queue, a heap, a treeset, ... Many variations are possible for the scheduler, mainly because it is the most time consuming part of the simulator and it isn't explictly stated how it should be done in the abstract simulator.
It might seem that a heap is always best, since it is mostly used for scheduling and has a low complexity. But in Classic DEVS, collisions should be able to occur. This would mean that if every element from the heap would collide at once, there should occur n pops to the heap, raising the complexity to $O(n * log(n))$. The scanning list is, though mostly slower, faster in these kind of cases.
Not only time complexity is dependent, but also space complexity. A scanning list doesn't save any information, thus has a space complexity of $O(1)$, while a heap has a complexity of $O(n)$.

2.4 Comparison

The DEVS-compliance and performance are rather important parts, which is why we make a more detailed table about the different tests that were used.
For the compliance tests, the test case has a name RXTY with X being the requirement as mentioned previously and Y being the test case number for this requirement. This table has approximatly the same form as the original one in [22], but with different simulators.
The performance table contains the different tests that were used, together with the used parameters. All these results are in seconds, measured on an Intel SU3500, 1.4 GHz.

2.4.1 General

	ADEVS	CD++	DEVS-Suite	MS4 Me	PyDEVS	VLE	XSY
formalism	DynDEVS	Cell-DEVS	PDEVS	PDEVS	CDEVS	DSDE	CDEVS
IDE	no	opt	no	yes	no	opt	no
GUI	no	opt	yes	yes	no	opt	no
parallel	yes	yes	no[3]	no[4]	no	no	no
distributed	no	yes	no[3]	no	no	yes	no
stop conditions	time	time	steps	time/steps	function	time	time
scheduler	heap	list	treeset	N/A	list	heap	list
performance	fast	medium	slow	medium	slow	fast	slow
interactivity	no	file	GUI	GUI	no	no	CLI
debugging	medium	medium	medium	medium	easy	easy	medium
enforcement	no	no	no	partial	no	yes	no

Table 2.1: The main comparison

2.4.2 Compliance

	ADEVS	CD++	DEVS-Suite	MS4 Me	PyDEVS	VLE	XSY
R1T1	PASS	FAIL	FAIL	FAIL	FAIL	PASS	FAIL
R1T2	PASS	FAIL	FAIL	FAIL	FAIL	PASS	FAIL
R2T1	PASS	PASS	PASS	PASS	PASS	PASS	PASS
R3T1	PASS	PASS	PASS	PASS	PASS	PASS	PASS
R4T1	PASS	FAIL	PASS	PASS	PASS	PASS	PASS
R4T2	PASS	FAIL	PASS	PASS	FAIL	PASS	PASS
R5T1	PASS	PASS	PASS	PASS	PASS	PASS	PASS
R6T1	PASS	PASS	PASS	N/A	PASS	PASS	PASS
R6T2	PASS	PASS	PASS	N/A	PASS	PASS	PASS
R6T3	PASS	PASS	FAIL	N/A	PASS	PASS	PASS
R7T1	FAIL	PASS	FAIL	N/A	FAIL	PASS	FAIL
R7T2	FAIL	PASS	FAIL	FAIL	PASS	PASS	FAIL
R7T3	FAIL	PASS	FAIL	N/A	PASS	PASS	FAIL
R8T1	PASS	N/A	PASS	PASS	N/A	PASS	N/A
R9T1	PASS	N/A	PASS	FAIL	N/A	FAIL	N/A
R10T1	N/A	FAIL	N/A	N/A	PASS	N/A	FAIL

Table 2.2: The DEVS-compliance table

2.4.3 Performance

	ADEVS	CD++	DEVS-Suite	PyDEVS	VLE	XSY
traffic 3/1000	0.01s	0.34s	19.3s	2.15s	0.063s	7s
recurse 100/1000	0.56s	71.6s	bugged	1842s	0.54s	395s
highcomp 1000/1000	2.81s	243.8s	7015s	20822s	3.6s	4560s

Table 2.3: Performance comparison, *traffic* is the previously mentioned specification, *recurse* is a deeply hierarchical model and *highcomp* has a lot of components. Note that *MS4 Me* is not included as we were unable to construct such a model at the time being.

For a more comprehensive comparison, we also implemented a test similar to *DEVStone*[16] in the most important simulators. Note that we did not implement this test in *DEVS-Suite*, *MS4 Me* and *XSY* since it is impossible to implement completely

[3]Note that there seems to be code available to do this, though our models did not get simulated on all available cores.

[4]We were unable to create huge models for this simulator due to a lack of some features. The simulator is also closed-source, so we can't search whether or not this is supported. Experimental results showed that multiple cores were used during simulation, though this might be one for the GUI and one for the simulation (as only 2 were used).

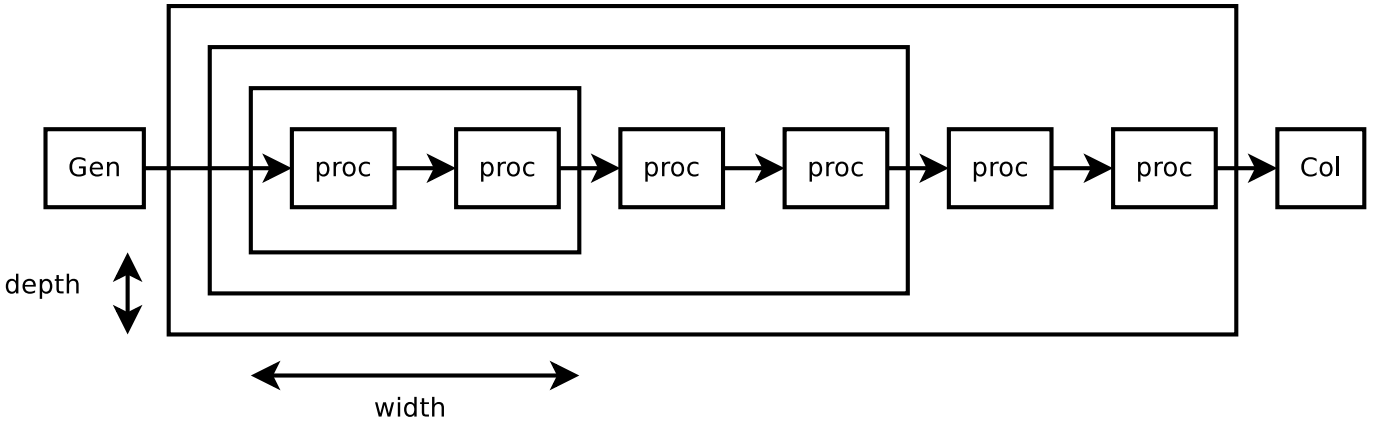

Figure 2.5: Our modified *DEVStone* test, this one has *width = 2* and *depth = 3*

automatic tests for these simulators (due to the need for user input). This would seriously influence the overall graph and since these simulators are not the fastest, they are not included. We included *PythonDEVS* as this is the simulator under study in the next chapter. The *PyDEVS-mod* simulator is the one with all optimisations enabled that are mentioned in chapter 3.
We did deviate slightly from the *DEVStone* tests, since we don't include a specific delay in the *transition functions*. The used model just passes around the received message after a certain delay. This is useful since we compare simulators in different programming languages. It would be unfair to compare them independent of this, as the models should perform (essentially) the same work and not just 'a time'.

Figure 2.6 plots the time it takes for the different simulators to complete the *DEVStone* test for a simulation time of 1000. Note that around a width of 400, the simulation seems to flatten in some simulators. This is due to the fact that any component that lies further can never be reached within 1000 simulation time units. Nevertheless, it is interesting to keep on plotting, since it becomes clear that even permanently inactive components will have an impact on *CD++* and *PyDEVS*[5], while other simulators are inaffected.
The complexity in a *normal* simulation can be determined from the first part of the plot. It is clear that *ADEVS* and *VLE* are extremely fast in these cases.

Figure 2.7 plots the same behaviour, but with a constant width and an increasing depth. Again, it becomes clear that *ADEVS* and *VLE* are the fastest. Note that *VLE* just 'stops' near the end of the plot, which is due to *VLE* denying to simulate the model as it has reached the maximum depth in its XML parser.

2.5 Evaluation

For each simulator, the most important entries in the comparison table will be discussed. For the DEVS-compliance table, each failed test will be explained for the simulator in question.

2.5.1 ADEVS

Scheduler The basic data structure for the scheduler is based on a heap, which contains only the active models (thus the elements which have a time advance value different from infinity). Deletions from the heap are done by setting the priority of the element to zero (the lowest possible), shifting the element upwards to the top of the heap and finally popping the top element.
The scheduler and related data structures are discussed in detail in [4].

Performance ADEVS is a high-performance DEVS simulator, even the fastest on the list in most cases. Only VLE comes close, not surprisingly using C++ too.
Usability is slightly decreased due to the difficult use of message passing, which requires the use of templates and such. Though this slightly decreases usability (for first-time users), this has several advantages over other implementations, since the simulator is really compiled for these specific models [33].
Besides, because templates are used instead of a base class for messages, messages can also consist of simple objects and are not limited to (inherited) classes, which also has the potential to increase performance.

Interactivity Contrary to most other simulators, ADEVS doesn't allow any interactivity at all. This might be partially because of efficiency, since it would take some extra time to allow user intervention.

[5]This experiment with PyDEVS took way to long to include in this graph, as it would mess up the scale

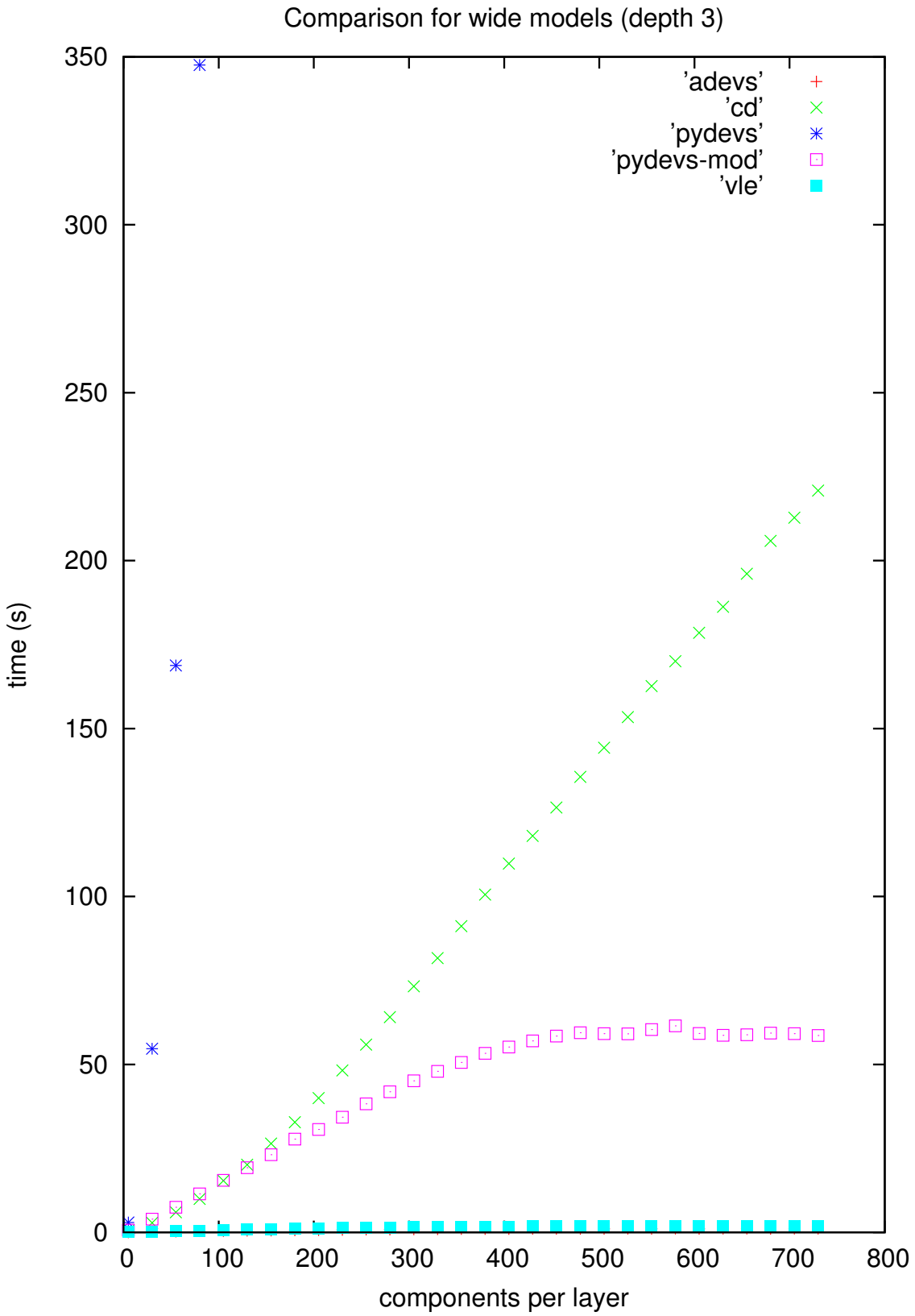

Figure 2.6: *DEVStone* test: *depth = 3*, *width* varies. Note that after about 400 components, no messages will arrive at the end. Further addition of new elements don't influence the simulation at all, this makes it clear that some simulators handle this nicely, while others don't.

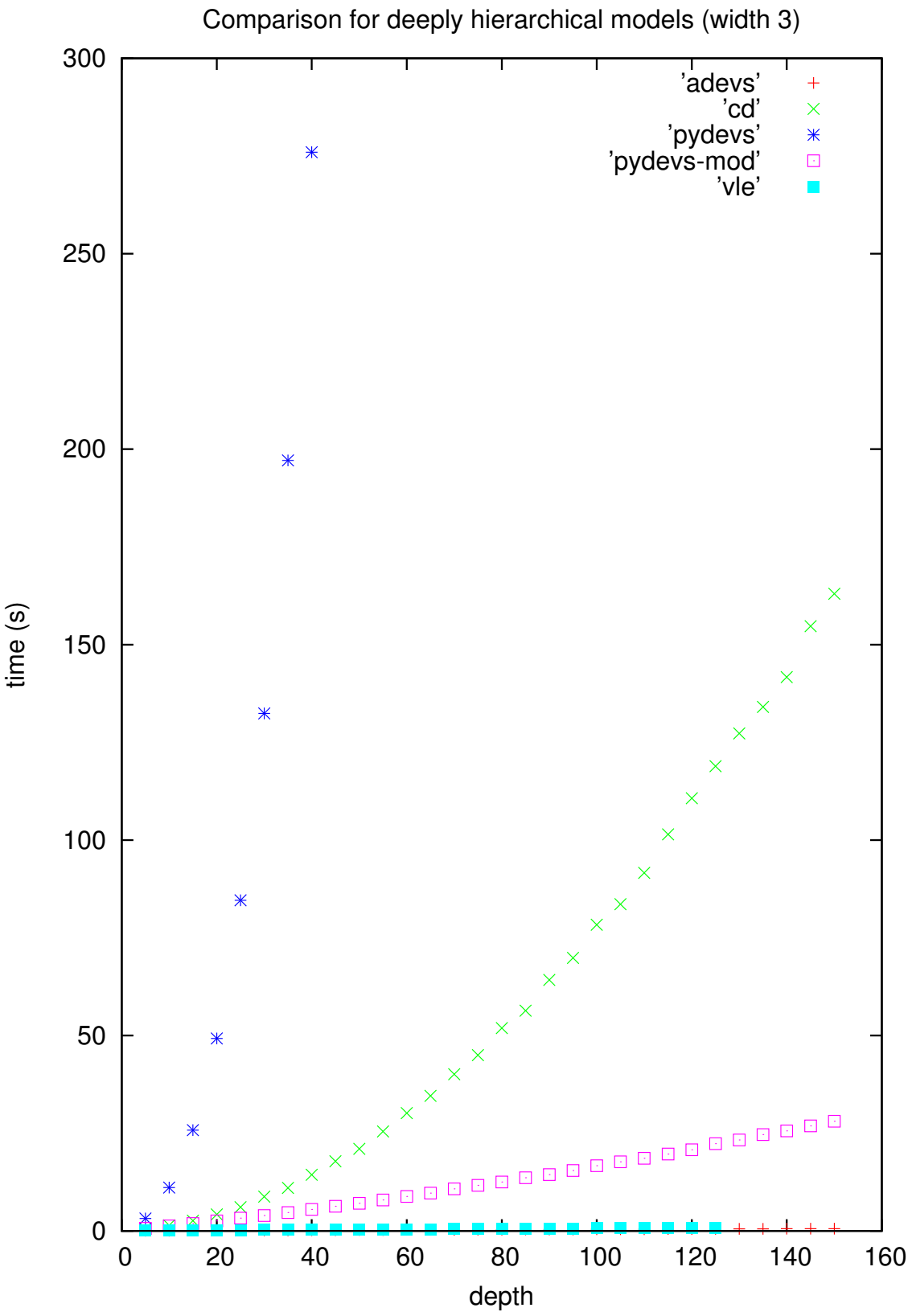

Figure 2.7: *DEVStone* test: width = 3, depth varies. VLE suddenly 'stops' because the generated XML-file became too big and VLE refused to load it.

Debugging Just like interactivity, ADEVS adds no features that make it easy to debug code. There is no possibility to toggle verbose mode or something like that. It is mentionable that the use of templates might be used to find errors while compiling, since this allows some static checks on message types, thus finding message passing errors. However, this might cause rather obscure compilation errors and actually be counterproductive in some cases.

Enforcement ADEVS doesn't make any effort to enforce the user to program correct DEVS models. This is quite logical, since the model just gets compiled with a standard C++ compiler. If any enforcement would be made, this would have to be done in *C++*-specific syntax, e.g. the use of *const* keywords (as is done in *VLE*).

Compliance The only tests that ADEVS fails for are tests R7T1, R7T2 and R7T3. These tests are concerned about the message copy functionality. It is actually quite logical that this simulator, being very performance-oriented, doesn't do unnecessary (and possibly expensive) operations and shifts these to the user instead. ADEVS just passes around the pointer to the message, which could cause problems when this message is modified in a specific model. This problem could be easily solved by either forcing the user to define a hand-made copy function that will be invoked, or by using the same approach as *VLE* and declare the incoming messages as constants.

2.5.2 CD++

IDE and GUI CD++ doesn't have any included GUI or IDE, though there do exist extensions to do this, like CD++Builder[10]. Most of these extensions are also oriented towards the Cell DEVS extension and not at the Classic DEVS part, which was tested. Therefore, these extensions are somewhat irrelevant in this comparison.

Parallel and distributed The simulator that was tested was the standard CD++ simulator, not the parallel version (P-CD++). Hence, parallel processing would be rather strange in this version. On the contrary, distributed simulation is supported in this version by running the simulator as a daemon (without any parameters).

Stop conditions Stop conditions are controlled by passing a time to the simulator at invocation. In contrast to the other simulators, the format of this time should be in HH:MM:SS format, which is not really conform to the way time is handled in the models themselves (in seconds only). This did require a small calculation to make sure the same time was used in all simulators and can be quite confusing to first time users, as the simulator also accepts ordinary numbers and will interpret them in an non-intuitive way.

Scheduler CD++ uses a scanning list approach. This means that there is no order saved, but each time there is a request to the next element, all elements will be iterated and their *timeNext* value compared to the current minimum. The main advantage is clearly that no expensive data structures are used, though a lot of iteration over elements is required in big models.
The use of a scanning list might seem strange at first, since a heap should be faster in the average case (judging from a complexity viewpoint). Due to the focus on Cell DEVS, this is not a bad choice as Cell DEVS will often cause more collisions than an average Classic DEVS model, since all cells contain exactly the same models. As can be seen later, in the complexity analysis of a heap scheduler for classic DEVS (in section 3.2.2), a heap scheduler can be rather inefficient when a lot of collisions or reschedules happen.

Performance Thanks to the implementation in C++, the efficiency is rather high. Though compared to other simulators written in the same programming language, the efficiency is actually disappointing. This might be due to the implementation of the scheduler, which uses a scanning list instead of a more efficient one.

Interactivity Interactivity is not really supported at simulation time, e.g. pausing, injecting, ... is unsupported. Though there is support for file-based interactivity, in which a time instant and a corresponding event should be written. The built-in generator will then output this event and eventually, a built-in transducer will receive the output. This output will then be checked according to the provided file. While this isn't real interactivity, there is a notion of inserting data into the simulation. Besides, this improves the possibilities for writing tests for the models, since a basic test framework is included by default.

Debugging From the debugging point of view, there are several helpful error messages in most cases. The built-in custom language for defining coupled and Cell DEVS models, provides several opportunities that most standard programming languages don't have[20, 41].
On the other hand, there is no possibility to enable verbose outputting (in the case of Classic DEVS at least), so the user has to manually write all the required print statements to be able to trace the execution. Note that there is a program *drawlog* included, to visualize the state of the cells in ASCII.

Enforcement As previously mentioned, CD++ is primarily focussed on Cell-DEVS and it is not really tweaked for the Classic DEVS part that is included. This makes some design decisions rather strange in comparison to other tools, some examples are the way of sending of messages, the use of absolute time, ... But one of the worst of all is the impossibility to send a message apart from a single float, as this would require major rewrites to models that are ported from other simulators. It seems impossible to

send complex messages, as the message is hardcoded to a float, making it impossible to e.g. inherit from this base class. While this decision might seem restricting, it is actually a relatively normal decision in the light of Cell DEVS, since this part of the simulator uses floats exclusively (it would be hard to generate complex messages in the custom language).
The Classic DEVS part of CD++ is clearly meant to be an extension to the Cell-DEVS functionality, e.g. to couple Classic DEVS and Cell-DEVS together as in [40]. CD++ is not meant to be a normal Classic DEVS simulator and, while not impossible, would not be recommended to simulate Classic DEVS models exclusively.
CD++ also has the disadvantage that the absolute time is used instead of the relative time, meaning that the modeller would be responsible for determining the elapsed time. This is clearly a (minor) violation of the DEVS specification, but an experienced user would not find this a lot of trouble. It does become worse considering that CD++ uses floats, so when small delays are used, it isn't excluded that catastrophic cancellation might occur when the simulation has progressed to a large time. Should this be managed in the simulator itself, this kind of behaviour might have been avoided. Now the modeller is responsible for all of this.
Furthermore, CD++ requires an extra parameter when sending a message, namely the time. This time is a rather strange parameter, since this must be equal to the current time. When a different value is passed, the simulator crashes or just ignores the message (depending on how big the deviation was). While this is not really a problem as soon as it is known, it seems a rather strange decision to expose this to the user.

Compliance CD++ fails for tests R1T1 and R1T2, which is caused by not checking for a negative time advance. This problem is easily solved by placing an assertion after the retrieval of the time advance value.
Tests R4T1 and R4T2 fail too. This is because CD++ uses floats as a time base, instead of doubles like the other simulators. This doesn't really pose a problem in most situations, but might cause some problems for high-accuracy simulations.
Tests R8T1, R8T2 and R8T3 should also be mentioned, though these tests succeed, this has nothing to do with the simulator design. As previously mentioned, CD++ only allows the passing of floats, which are automatically copied by the compiler (unless references are used of course...). So while the tests succeed, it is done by imposing a huge limitation to the user.
Finally, test R10T1 also fails, due to the lack of a *select* function. The user cannot define a custom *select* function, as CD++ always chooses the first element from the list.
Beside the tests that have failed, the function interfaces are also somewhat different from the ones defined in the formalism. There is no *timeAdvance* function, but the user is responsible for using a *holdIn* or *passivate* function for this goal. While this might be circumvented by calling this function with the value of a custom defined *timeAdvance* function, there is no guarantee (by simulator design) that this always happens.

2.5.3 DEVS-Suite

GUI One of the most notable features of DEVS-Suite is the GUI. This GUI shows the couplings between the different atomic/coupled models, which can be very useful in debugging various errors or just to have a good understanding about the model. While simulating, the GUI also shows the message passing and some relevant information about the models.
A tracking environment is also supported in the GUI, which can be used to easily generate trajectories of the model. The main limitation about the GUI is that modifications are not supported, so should an error be found, the underlying Java file will have to be modified, recompiled and the model reloaded. Though this is kind of logical, since the couplings are compiled in the Java code instead of a save file.

Stop conditions A problematic part of this simulator is the stop condition of the simulation. Contrary to all other simulators, DEVS-Suite only has either *simulate until infinity*, or *simulate for a certain number of steps*. Most simulators support simulation up to a certain time, which seems more natural. This made the performance testing rather difficult, as we had to figure out how many steps were required to reach the required time.

Scheduler The scheduler is based on a Java TreeSet, which refreshes all of its data at every invocation. This is extremely inefficient, since the following steps happen:

1. collect all timeNext values of all models $O(n)$
2. make a TreeSet from this list $O(n*log(n))$
3. retrieve the first element of the TreeSet $O(log(n))$

All of these steps are very compute intensive, while it would have been a lot faster just to use a scanning list. This extreme inefficiency is reflected in the performance results. All in all, the scheduler has an algorithm with complexity $O(n*log(n))$, but also with a lot of smaller components. Therefore, we expect to see a rather inefficient simulation, though the use of Java might make up for most of this in small situations. Besides, the implementation in Java makes up for a lot of this constant amount.
This algorithm is also rather inefficient in memory usage, since there are two different structures used to maintain all data: the *List* and *TreeSet*.
DEVS-Suite easily has the worst scheduler of all simulators due to these very inefficient (though easy to implement) decisions.

Performance Testing the performance was rather hard, since there is no decent way to do this with a good accuracy. The time the simulation took includes the manual operations needed to start the simulation, so the accuracy is somewhat low. Besides, the simulation didn't allow simulation up to a certain time, so we previously did several simulation runs to find the most accurate number of steps. It seems that 5240 steps were required to iterate up to simulation time 1000 in the case of the *traffic* experiment. The performance was measured with both SimView and Tracking disabled, but had to happen in the GUI, since there are no other ways to run the simulation. The actual performance was very disappointing, since the code is written in Java, which should have decent performance. Probably, a lot of time is spent in updating the GUI while running the simulation since the times and states are still updated.
As already mentioned, the scheduler is (very) sub-optimal, slowing down the algorithm even further.

Interactivity Interactivity is very well supported in DEVS-Suite. There is a possibility to pause, resume, reset[6], inject messages and even view which messages are being passed. This also greatly aids in debugging the models.

Debugging DEVS-Suite is a decent simulator which contains some very interesting features, like being able to see the simulation run, though this does seem to have negative influences on the performance. On the other hand, the linking procedures work in a strange way, using strings to indicate the name of the port, instead of objects. This is very likely to cause some bugs in the models. Sadly, this bad point is combined with another one, being the lack of feedback for errors: the simulation just continues and does never mention anything about an impossible connection. This makes DEVS-Suite a rather vague simulator when debugging the couplings.
The tracking feature is very interesting, as it allows to draw trajectories of the simulation run, which can be used in verification. This is somewhat equivalent to the traceplotter [32] available for PythonDEVS, though the tracking feature of DEVS-Suite does contain a lot of extra features and is nicely integrated (and plotted while the simulation runs) in the implementation. This makes it much easier to learn about DEVS and the associated behaviour, which has been employed in [13].
All in all, debugging is rather easy at logic level, but somewhat more difficult in errors concerning the API, e.g. illegal couplings are never mentioned.

Enforcement While the GUI makes it easy to see what happens, this severely limits the possibilities to use the simulator in scripts as it does always require user intervention. This is also the reason why no precise times can be given for the simulation. When simulating huge models, it might have a severe drop in performance, due to the constant redrawing of the GUI even when the Simulation Viewer is disabled.

Compliance DEVS-Suite fails tests R1T1 and R1T2, like most simulators. Simulation will just continue even though a negative time advance was given. This problem could be easily fixed by including a check for the time advance. Note that DEVS-Suite does not use a timeAdvance function like most other simulators, but requires the use of the *holdIn()* function.
Test R6T3 also poses a problem. We tried modelling the required model, but did not succeed in correctly linking them together. Even though the code itself didn't seem in error, the Simulation Viewer did not show the required links. When the names of the port were inverted (but not the models), the couplings were drawn, but no messages were carried over them. This shows the importance of the Simulation Viewer in debugging, though it is assumed that the impossibility to make this specific coupling is a bug in the simulator.
Furthermore, tests R7T1, R7T2 and R7T3 fail. This is most likely due to the fact that Java uses references all the time. Again, a solution might be to use constant declaration. Note that test R7T3 would have failed no matter what, since it uses the same model as R6T3, which also fails.
DEVS-Suite doesn't adhere to the DEVS specification completely, since there is sometimes access to unnecessary variables. Though this is not really a violation (as this value is just equal to the timeAdvance function), it should be mentioned. Also, there is no use of a *timeAdvance* function, so the user has to manually call the *holdIn* function. This can cause bugs like in CD++, where the call of the *timeAdvance* is not guaranteed by simulator design.

Bugs DEVS-Suite does seem to have some (severe) bugs when constructing some models. Some of the bugs that were encountered are:

1. Resetting models sometimes causes a wrong simulation afterwards
2. Coupled models sometimes prove difficult for the simulator, causing all kinds of errors. These errors go from messages that follow a (visually) wrong path, to messages that never arrive.

Most of these bugs have a serious impact on the use of the simulator, making it noteworthy to mention in a comparison between simulators.

[6]There does seem to be a bug when resetting some models

2.5.4 MS4 Me

Note that MS4 Me is still in development, so this evaluation is subject to change.

IDE The use of the provided IDE is somewhat mandatory, since MS4 Me uses a custom natural language. Of course, it is possible to manually code Java files and simulate these, though this isn't really meant to happen. While the natural language seems promising, the use of Java is still strongly encouraged since only very simple models can be modeled using the natural language alone. This natural language does actually limit reuse of models that are written in other simulators.
The use of a natural language, which stands rather far from the Java code that gets generated, does provide some problems for first time users so it is still easy to be able to reference to the generated Java files. For example to see where the Java tags actually go in the code, so you know which variables are available.
On the other hand, the IDE is very complete and due to the natural language a lot of information can be shown in a sidebar (like output ports, states, type of messages, ...) and errors can be quickly marked.

GUI Simulation can only be done using the GUI, making the GUI a very important element. The SimView component is complete and contains even more features than DEVS-Suite. Several additions are message expansion (see the actual values of the message, instead of just the type), model compaction and more flexible message injection.
Nonetheless, the GUI and IDE are rather resource-hungry, probably mostly because it is based on the Eclipse environment, but this has its own advantages like being familiar to Eclipse users.

Stop conditions The simulator allows for two types of stop conditions, being the simulation time or the number of steps. These stop conditions are exclusive, so both cannot be defined at the same time to stop when one of the two triggers a stop.

Scheduler Since MS4 Me is closed source, no information about the scheduler is known.

Performance The performance tests are rather difficult to do, especially since MS4 Me is still in development and several features severely limit the possibility to take a decent performance measurement. There is no possibility to simulate without a graphical simulation view, nor is there a way to exactly time the simulation. Besides, since the simulator is still in development, performance updates still keep being issued from time to time. Furthermore, the simulator is currently only supported on Windows. All in all, the use of Java and the GUI is expected to limit the performance.
Only some very basic usage performance is visible, though this is not an accurate timing. All in all, the simulation runs relatively slow in some very small cases. We understand that this is not an accurate way of determining performance and complexity. However, this is the only model that can be modelled and ran for the time being.

Interactivity As already mentioned, the GUI allows for a lot of interactivity.

Debugging Debugging can get very easy due to the very helpful GUI and the use of the natural language. However, there is sometimes a lack of feedback. Sometimes code within Java tags doesn't even get included without giving (at least) a warning. This seems rather strange, since the user doesn't know whether there is an error in logic, in the Java code, in the natural language or even somehwere else.

Compliance Most tests that we would like to do require a certain way of hierarchical coupling, which is not yet provided by MS4 Me at the time of writing. To circumvent this problem, we devised a stripped-down version of most tests, to make make them usable (but they still test the essence). Some tests do test the hierarchical coupling, so these cannot be replaced.
MS4 Me fails test R1T1 and R1T2, noteworthy about this is that the natural language does prevent this kind of situation as it will do a static check on the positiveness of the time advance. In case the time advance is just a function call, this return will *not* be checked and thus it is possible to make this test fail.
Test R7T2 fails, as no copy of the message is made. This is the only part of this series of tests that could be tested, since it is the only one where no hierarchy is necessary.
Furthermore, test R9T1 fails due to the lack of a confluent transition function that can be described in the natural language. In case of a confluent transition, a standard function will be called that handles this. The reference manual showed no sign of a confluent transition function, so we believe that it is currently not possible to implement one.
Another point that is worth pointing out is the fact that the *time advance* function gets called *before* the internal/external transition function by default. The only solution would be to call the *holdIn* function manually at the end of the transition functions.

Enforcement There is a possibility for a lot of enforcement with the natural language, since a custom environment and language is used. This functionality is employed, though not completely. For example the code between the Java tags is not verified (nor syntax highlighted), so violations are still possible. It is actually kind of logical that Java code doesn't get verified, due to the generality of Java.
For example for the negative time advance test, simply using the custom language to wait for a negative amount of time gives a helpful error. If the custom language just calls a Java function defined elsewhere, no check is being done on this code and it is possible to execute a model with a negative *time advance value*. MS4 Me doesn't define a *time advance* function either, but each

state must have a *wait* duration (defined in the custom language). Due to the inclusion of Java code, it is possible to call a *time advance* function manually, though this might get bothersome and is prone to errors. Luckily, the natural language will do a lot of checks to make sure that such a call is defined.

2.5.5 PythonDEVS

Stop conditions A very interesting feature of PythonDEVS is the ability to pass the termination function as a parameter to the simulator. This allows the stopping at virtually anything, which is a lot more versatile than just passing an integer. Examples for the most common functions are provided (stop at specific time, don't stop and stop in specific state) in the examples.

Scheduler The scheduler is implemented in the form of a sorted list, which is rather wasteful as only the first event(s) are necessary for the scheduler. PythonDEVS makes the second worst scheduler decisions of all the compared simulators (with DEVS-Suite being the worst).

Performance The performance is rather slow in small cases, though that shouldn't be too surprising knowing that the scheduler uses a sorted list. In huge simulations, the performance is disastrous due to the sorted list implementation, which has a $O(n * log(n))$ complexity, combined with the use of Python. Also the message passing function can go to $O(n^2)$ in the worst case.
The verbose output has a serious disadvantage, since all performance optimisations should try to keep the same output, which reduces the possibilities for optimisation, besides, there happen a lot of checks whether to print or not. Some more information about these performance limitations and solutions can be found in 3. Note that the low performance of PythonDEVS is not (completely) due to the choice for Python as implementation language, but mainly due to the choice of algorithms, which have a rather high complexity in most cases. These algorithms have been modified in chapter 3, to show that this slow performance is mainly due to the bad algorithms.

Interactivity PythonDEVS doesn't support any kind of interactivity and while the simulation can be continued by rerunning the simulation in the same Python file, this doesn't seem to be advised as it might cause a broken simulation [9].

Debugging Concerning debugging, PythonDEVS becomes an excellent choice due to the very verbose output that can be enabled, which contains all the information about the model that is of importance, like the current state, which transition is called, the messages on all ports, ...
Besides this, the use of Python also allows for fast modification/execution cycles which can be very easy for small fixes to the models.

Enforcement A rather serious limitation to the use of Python is the impossibility to add some constraints to allow DEVS enforcement. While most other languages, like C and Java, allow the definition of *const* variables or functions, this isn't even supported by Python. On the other hand, Python allows for very easy message passing due to its dynamic typing. This might be easy and practical in some cases, but has some serious disadvantages which many simulators try to avoid [33], like being more prone to typing bugs.
The function interfaces are also rather badly designed, since they don't take any parameters. The model must make a call to the simulator object, which will then return the requested message. Though this makes it somewhat easier to retrieve simple messages, it has several disadvantages, like being unable to access the 'raw' input, which might allow more performant user code to be written. Also, the user has to know about which functions can be called in the simulator object and when it is appropriate to call them. It might be inappropriate to perform a *peek* or *poke* while inside the internal transition function, which doesn't really protect the user.

Compliance Since PythonDEVS was also one of the simulators tested in [22], the results of this test run is the same. For completeness, some explanation will be given about the tests:
R1T1 and R1T2 fail, like with most simulators. Again a simple check would solve this issue.
Furthermore, test R4T2 fails, this is because PythonDEVS uses an *epsilon* to determine whether or not two events collide. This might be necessary to compensate the accumulated error that might be present due to the floating point calculations, though it might cause some (unnecessary) collisions too.
Also test R7T1 fails, which is somewhat surprising, since R7T2 and R7T3 succeed. The reason for this is that a *deepcopy* is used to make a copy of the message, but looking at the code, it is seemingly forgotten to do this at the coupled model level.
The modified version of PythonDEVS, that was made for this research internship, is an attempt to speed it up, and also contains fixes to be able to pass all these tests.

2.5.6 VLE

IDE and GUI VLE has several extensions that act like a GUI or an IDE. Though this is optional, it is highly recommended to use the GVLE extension, as the coupled models cannot be generated with a programming language. Consequently, manual creation and editting of the save file (in XML) would be necessary. The GVLE extension makes this process a lot more comfortable, since

couplings can be drawn from a GUI and code can be edited in an IDE.
This approach does have some disadvantages, like not being able to use programming language constructs (e.g. *for* and *while*) to make repetitive tasks easier. There does exist a translator to perform these cases [28].
On the other hand, users that don't know anything about DEVS would be able to construct models using this GUI, while using a standard programming language would be a lot more difficult for the user that doesn't know anything about the programming language.
Since the GUI is (practically) necessary to develop the coupled models, this GUI will be taken into account in the other points. For performance, only the command-line simulator is used, without any form of GUI.

Scheduler The scheduler used by VLE is a special kind of heap. The heap contains all the events (even the passive ones), but instead of a special operation to delete an event (for a reschedule), the event just becomes invalid. Invalid elements will afterwards just trinkle up in the heap until they are due, when they are deleted using the standard pop of the heap. This approach is not really that memory saving, though it does allow for cleaner code (no heap reimplementation) and often more performance (no expensive deletes).
A rather important bug was found when analyzing this heap implementation. It seems that the heap never gets cleaned up if too much invalid elements are present. To verify this assumption, we tried creating a very simple model, with only a few atomic models, which makes vle consume all memory present on the machine (before being killed by the kernel). The model consists of a generator that kept sending messages to a processor. This processor will reschedule every time in the far future, while the generator rescheduled at exactly the same time. Of course, any schedule would suffice as long as the processor never gets a turn. This model made the eventlist explode in size, with millions of events, while there were only a few models.
It is true that this example is rather artificial, but it might be possible in larger experiments that this situation occurs without the user knowing. Or it might also cause a high performance penalty as soon as all this memory has to be freed with only heap pops.

Performance VLE has a very good performance, almost on par with ADEVS in most cases. Combined with the distributed functionality, VLE will be faster for huge projects that can be run distributed. This performance is not only due to the usage of C++, a fast compiled language, but also due to the employed algorithms.
Note that the previously mentioned problem with the scheduler might cause severe slowdowns (when the whole eventlist gets popped) or huge memory consumptions (when the aformentioned situation arises).

Interactivity While running, there is no interactivity possible with the simulator. An interesting feature though, is the percentage display of progress, though this isn't usable for interaction.

Debugging A very big disadvantage is that GVLE doesn't seem to be able to automatically extract parameters or input/output ports of the generated models. This might introduce some typo's and cause hard to detect errors. Besides, it becomes bothersome when connecting multiple elements that have the same Dynamics (and thus the same ports/parameters) and all of these have to be redefined each time.
While the GUI can be used for graphical modelling, it can't be used for graphical simulation.
VLE isn't very helpful in finding logic errors (no traces visible), but more in the debugging of DEVS compliance of the models due to many *const* keywords.

Enforcement As already mentioned, VLE is really helpful concerning the enforcement of the DEVS formalism. Though this can sometimes become rather bothersome since a lot of *const* keywords have to be set in the user code to provide the necessary keywords to the C-compiler which checks if the required conditions are required.
Nevertheless, this feature doesn't require a lot of work from the simulator itself (nearly no code) and nearly all checking is done by the compiler, making this a very interesting feature that might be interesting to all other simulators with this possibility.
On the other hand VLE makes use of absolute time instead of relative time. This is somewhat logical, since most uses of VLE are in the natural sciences, which probably would require this value. Though this does make porting models to VLE a little bit more difficult, especially since this might cause unexpected behaviour if the modeller is not informed about this.
Because of this choice, the function interfaces are also a little different from the ones defined in the formalism.

Compliance It is noteworthy that VLE passes most tests easily. This is partially thanks to the use of the *const* keyword as much as possible. For example the message passing: the user doesn't receive a copy, but the message is just a const, so the user can't modify the message in any way (the compiler will check this!). The only way to modify this data (which is often required), is to make a copy yourself. For this reason, each message should have a *copy* function declared. Thanks to these diverse checks, some bugs have even been found in the implementation that was used to compare the simulators.
Test R9T1 did actually cause some trouble since the confluent transition function didn't get called in some occassions, which is clearly a severe violation of the DSDE formalism. As a sidenote: the documentation about this functionality is not really up-to-date.

Bugs While testing the VLE environment, two (severe) bugs were found:

1. Memory and performance bug: The scheduler of VLE is, like previously mentioned, a heap that invalidates items to reschedule them. As soon as an invalid item is seen as the first, the next one is used immediately until a valid one is found. The main problem with this approach is that there might be some models that exhibit behaviour that exploits this fact and makes the heap explode in size. It doesn't seem as if a clean-up method is defined in this case, so it is possible for a very small model (4 atomic models in our case) to make VLE consume 2 GB of memory, before getting killed by the kernel due to a lack of free memory.
   From the performance perspective, it might be possible that a heap with a lot of invalids gets generated, but right before memory is full, simulation progresses to right after all these invalids. VLE would then have to pop all these invalid elements from the heap. Due to the implementation of a heap, a *pop* is not cheap, as the whole heap has to be rebuild. This might cause immense delays between two transitions.

2. Confluent transition: As seen in test R9T1, the confluent transition doesn't get called in some situations. The test framework that we provided will trigger both the Generator's internal transition (thus its output function too) and the two Processor's internal transition. This should (normally) be resolved by first computing the different outputs and the influenced elements of this. The result should be:

   - Internal transition in Generator (generate new events)
   - Confluent transition in Processor (pass on events and receive generated events)
   - External transition in Collector (receive passed events)

   From a trace in the models themselves, it seems that this isn't respected and that the internal functions get called one after another, with the outputs and transitions interleaved at exactly the same time. Therefore the confluent transition functino is not called at all.

2.5.7 XSY

Scheduler XSY uses a scanning list as a scheduler. This seems to be a common approach for Classic DEVS simulators.

Performance The performance of the simulator is rather dissapointing, since it mostly uses the same methods as PythonDEVS and the low performance isn't really explainable (as it was with DEVS-Suite).
Since XSY is open-source we tried to search for the main reason of inefficiency. It seems that Python only uses about 25% of the CPU in some cases, which is caused by the many *sleep* calls that happen within a thread, mainly the call at SimEngine.py:123. If this sleep call was commented, simulation was many times faster. Luckily, other cases use about 80% of the CPU.

Debugging One of the main features of XSY is the verification engine, which could be helpful when debugging. Since the same implementation language is used, approximately the same advantages and disadvantages are seen as with PythonDEVS. Contrary to PythonDEVS, XSY is more of an *interactive simulator* as it includes an interactive menu with the possibility to inject messages, pause, statistics, ... On the other hand, the verbose output is not very clear, since it contains a lot of useless information which would become very messy when large models are simulated. After each time step, all the models and their timeNext value are printed, even though only a few of them should have changed and be relevant in this context.

Enforcement XSY doesn't use the same function interfaces as the formalism suggests, since the elapsed time must be asked from the simulator object that is linked to the model, instead of being passed as a parameter.

Compliance XSY fails tests R1T1 and R1T2, like most simulators. Again, a simple check would suffice to prevent this.
Also tests R7T1, R7T2 and R7T3 fail. In Python, this problem can be easily circumvented by using the *deepcopy* function, though it will drastically reduce performance in some cases.
Test R10T1 fails too, since XSY doesn't allow the implementation of a select function. Instead, the first model that is found is used.

2.6 Conclusion

Overall, it is clear that most simulators have different goals and they clearly succeed in their goal. Even though the existence of many simulators would imply that there is a lot of choice, reality is different. Most (bigger) models that have to be written will have a simulator that closely supports this kind of models. This makes some simulator - model combinations exceedingly difficult.
As a very summarized comparison, the ideal situations for the following simulators are:

- ADEVS: big models that require a high performance.
- CD++: for use with the Cell DEVS formalism.
- DEVS-Suite: users new to DEVS that require a graphical simulation.

- MS4 Me: users that don't know a lot about DEVS or programming languages.
- PythonDEVS: fast prototype models or for those learning about DEVS.
- VLE: for use in natural science, where the end-user might not know about DEVS too much.
- XSY: mainly for the verification engine or the increased interactivity versus PythonDEVS.

3 Optimisations

The main focus of this research internship is on making the PythonDEVS simulator faster. This is needed, since the need for fast simulation increases as the models get more and more complex. In software design, an important choice that has to be done is between performance and maintainability. The original implementation of PythonDEVS was rather easy to maintain. Some parts took very long to process, but were implemented in merely one line. As this increases the complexity, this is not desired. So in this research internship, we try to decrease the complexity - or more generally the simulation time - while still trying to have a decent maintainability.
An introduction to the PythonDEVS simulator is out of the scope of this document and we refer to [9] for a detailed description.
All the changes to the original implementation can be subdivided in roughly three categories. These categories are reflected in the structure of this chapter. For each optimisation, we will explain 1) what we have implemented, 2) what was the reasoning behind it, 3) how this affects complexity and 4) show some empirical results. As for the empirical results, we will show how *only* this optimisation has affected the simulation time. Some optimisations do work together, causing some kind of *synergy*. The empirical results might show some small difference - or even an increase in simulation time -, but those optimisations together might have a huge impact on the performance.
The empirical results are simply discussed, the results can be found in section 3.6. This was done to make comparison between the different optimisations and test cases easier.
At the end of this chapter, we show the difference between the original PythonDEVS simulator, including all partial changes, and the optimised PythonDEVS simulator. At the end of each optimisation, a reference will be made to the table and a discussion of the results will take place.

3.1 Note about performance

It should be clear that the performance increase of a simulator will only affect the speed of the simulator and not of the simulated model. If the models that are being used in the simulation are the main bottleneck (e.g. 95% of the time is spent in the models), no huge speedups will be noticable simply because the simulator is not responsible for the slow performance.
We only try to minimize the *overhead* of the simulator, particularly in large models (e.g. lots of components, lots of connections, ...). So to test this *overhead* we implemented extremely minimalistic models in all of these test cases to minimize the time in the models itself. This means that a *High Computation, Low Interconnection* model will probably not speed up that much, maybe even by a few percents. On the other hand, *Low Computation, High Interconnection* models will benefit from these changes, as the main portion of the simulation time will be in transferring data, scheduling events and such.
Note that in the *DEVStone* benchmarks also *overhead* of the simulator is measured, by performing a fixed time of computation in the models. We did not go this way, as there would be a huge difference between e.g. Python and C++ as implementation language. This would give biassed information, as the C++ models would clearly be able to process a lot more information than the Python models.
As an example, only fetching the data from the ports in PyDEVS might easily take longer than fetching *and* processing all information in C++. Should we work with a fixed time, we might give the impression that one simulator is better than the other, even though it would actually work slower in every case due to this difference.
No matter which kind of models are used, the new PyDEVS kernel will scale a lot better in every dimension.

3.2 Scheduler

An important part of every simulator is the scheduler. The scheduler is the part of the simulator that has the biggest data structure and therefore, also consumes a lot of time to manage this data structure. Besides this, the scheduler will be called frequently, since it is needed to determine the next event to be processed. Due to these reasons, the scheduler receives its own section.

3.2.1 Heap approach

The main data structure that the scheduler will use is based on a heap. This choice has been made due to the excellent complexity of a heap to push an element and to pop the first element. These are the main operations that the scheduler will be processing, so the heap approach seems ideal. An important disadvantage of a heap is that it is not easy to modify the priority of an element in the heap, which would occur when an event gets rescheduled. This problem will be solved seperatly.
So for a basic scheduler, the heap approach would have a rather low complexity. The Python Standard Library does have an implementation of the most important heap features, namely the *heappop* and *heappush* operations. The heap itself is managed as a list and every list can be transformed in a heap using the *heapify* operation.
A heap is also used as the data structure for the schedulers of ADEVS and VLE.

Active models only

Like most data structures, the complexity of operations on the data structure are dependent on the number of elements in it. For this reason, it might be interesting to drop the passive elements (those with a *timeNext* equal to ∞) from the data structure, thus possibly reducing the size of the data structure and speeding up all other operations. To accomplish this, each element that gets added to the heap first gets checked for whether or not it is passive. If the event is passive, it doesn't get added. Should an event be rescheduled from passive to active, it will simply be added to the heap. In the inverse case, an event that is active and becomes passive will simply be removed from the heap.
A huge impact on performance should normally not be found from this feature, as the heap has a complexity of $O(log(n))$. So even if half of the elements were passive, it would only save a very small amount of time. However, the space complexity of the data structure would decrease significantly should there be many passive elements. This might be useful in combination with the next feature, which will handle the reschedules in an easy way.
This feature is also present in ADEVS and is mentioned in [4].

Invalidate events

To handle the rescheduling of events, several options were considered. The one that was most appealing and maintainable was the use of invalid events. This implies that instead of removing the event from the heap, a flag gets set in the event, showing that it is no longer valid. This would cause the scheduler to ignore this event and continue with the next event until a valid element is found.
This is easily the strangest part of the heap implementation, as it requires a lot of extra checks. It does have a lot of advantages, which were the reason why this approach was chosen. First of all, the Python library functions don't implement a function for random heap modification, only *heappush*, *heappop* and *heapify* are provided. This could be circumvented by completely rewriting these library functions, which isn't really a maintainable (or efficient) solution. Secondly, the size of the heap is not a very big concern, since 1) the performance doesn't change a lot since a heap has logarithmic operations, and 2) inactive elements are removed from the heap, creating some extra space.
The main disadvantages of this feature are of course the managing of the invalid events. Once an invalid event is encountered, it is just ignored and the next one is selected. However, care must be taken when using the first element of the heap to schedule the *timeNext* of the simulation, as it might be an invalid event. Besides this, care should be taken that the number of invalids doesn't raise too much, as it would make the feature inefficient due to the many *pop*'s that are needed to clean up these invalid elements (as with the bug in VLE). It would also make the memory usage of the simulation skyrocket. For these reasons, a cleanup operation is defined, which will remove all invalid elements in a linear function. This function will rarely be called, as in most scenario's the normal continuation of time will slowly clear up all old events.
Note that this *cleanup* operation will guarantee that the *space complexity* doesn't change, since the threshold is defined at e.g. two times the number of real models. This increases the space complexity with a constant, which is removed thanks to the big-Oh notation.
It is necessary to prove that this invalidation does not invalidate the heap invariant (to guarantee that the *heappop* and *heappop* have a valid heap to work on), this is necessary as the validation status is taken into account when sorting.
Note that some more optimisations would be possible, trying to lessen the number of invalidations required, though this would not speed up in the average case and might even slow down due to the extra checks that would be required.

Proof. The structure of the records is $[timeNext, ID, valid, model]$. When using *heappush* and *heappop*, the heap invariant is guaranteed to remain valid, so we assume that the invariant is valid before the invalidation.

valid gets compared with $False < True$ as a basis, while *element* gets compared on the objects adress in memory (its hash).
Since comparison of lists starts from the first element and continues until an inequality is found, the only structure that might invalidate the heap invariant is when there are 2 records $a = [time, id, v1, elem1]$ and $b = [time, id, v2, elem2]$ with one of them a child of the other.
Without loss of generality, we state that a is parent of b. As we require that the invalidation happens on a valid record, either $v1$ or $v2$ must be $True$. As the heap invariant is guaranteed to be valid at this moment, we know that $a \leq b$ and thus $v1 \leq v2$ with at least one of them being $True$. Which means that $v2 = True$ is guaranteed. The following two situations are possible:

1. $v1 = False$
   In this case $v2$ will change to $False$ after the invalidation. This means that the comparison will progress to $elem1$ and $elem2$, since $v1 = v2 = False$ after the invalidation. However, it is known that $elem1 = elem2$ is always guaranteed because the object is uniquely determined by the *id*, wich was already assumed equal previously. Therefore, the heap invariant that states that $a \leq b$ is still valid (with a parent of b).
2. $v1 = True$
   This case is impossible, since only one element can be valid at any given time. This is guaranteed by the invalidation that always happens when a new valid record is inserted.

□

The use of invalid events, though without a cleanup method, is also implemented in VLE.

3.2.2 Complexity

The complexity of the original scheduler was clearly determined by the sort of the eventlist. The Python sort function has average case complexity of $O(log(n!)) = O(n * log(n))$ [27].
The complexity of the modified scheduler is a lot trickier. It is clear that a *heappush* and *heappop* operation have a complexity of $O(log(n))$, while the *heapify* operation has a complexity of $O(n)$ [27]. Due to the threshold imposed on the number of invalids, it can be said that there will never be more than $2n$ elements in the heap. The total complexity will be $O(c * log(2n) + r * log(2n))$, assuming that the cleanup operation will never be called, which isn't a too optimistic assumption. The cleanup operation itself will have a complexity of $O(2n + a) = O(n)$. With the following definitions:

- a: the number of active elements
- n: the total number of elements
- r: the number of elements to be rescheduled to an active state
- c: the number of colliding elements

In most cases $c + r < n$ is a decent assumption (not too many interconnections and collisions), meaning that there will be a huge speedup. Should collisions be very frequent, it might actually be better to use the Parallel DEVS formalism, which greatly profits from a high number of collisions.
The worst case complexity is clearly $O(4 * n * log(2n)) = O(n * log(2n))$, while the best case complexity is $O(log(n))$.

3.2.3 Empirical results

From these results, it can be seen that the use of a heap will indeed decrease the complexity of the scheduler by a massive amount in the cases assumed in the complexity calculation. The results indicate that the performance of very deep hierarchical models really suffers from this implementation. This is not surprising, since all extra logic for the heap will run even though there are only two elements in the coupled model (the atomic model and another coupled model).
In the case where there are a lot of reschedules, like the high interconnection test, the heap will have to cope with a lot of invalid elements that have to be popped. In the worst case, a cleanup might even be required.
The most obvious case is the traffic experiment. When there are lots of models, the performance increases drastically, indicating that the complexity has decreased. This is logical since the traffic experiment has a lot of inactive elements (certainly in the beginning) and there are nearly no collisions.
It seems that this new data structure has a huge impact in cases where there are a lot of models in a single coupled model. This will be very helpful later on, when implementing *direct connection* 3.3.4.

3.2.4 Alternatives

As mentioned in the comparison of simulators, nearly every simulator uses its own kind of scheduler. Some different schedulers, together with their advantages and disadvantages will be given below.

- Sorted list
  The original scheduler of PythonDEVS uses a list of all events that gets sorted at every access. It is needless to say that this implementation is very slow, though very easy to implement. To construct the list that has to be sorted, all elements are scanned, basically reducing this version to an even slower variant of the scanning list approach.
- Scanning list
  Every time the first event is needed, every element is scanned for its *timeNext* value and if the *timeNext* is lower than the current minimum, this value is set as the new minimum. This is an optimisation of the previous method, as we are only interested in the first element, not in the relation between all events. It should be noted that this approach cannot make use of the 'active elements only' feature, as this approach doesn't save any info besides the first element. An interesting advantage of this approach is that it doesn't use a data structure to save a reference to the event and its event time.
- Heap
  This alternative is mainly an alternative due to the many possibilities to reschedule elements, the most logical alternatives are given by:
  - Change priority and heapify
    This approach is the easiest of all, as it only requires a call to the *heapify* function. Since the *heapify* function doesn't know that only one element is changed, it will try to heapify the complete list, resulting in a complexity of $O(n)$ in total. This would probably shift the complexity away from the pushing and popping to the rescheduling of events. There is one important advantage to this approach, namely that the time it takes is not as dependent from the number of reschedules as was the case with the approach that was previously chosen. In the the worst case, the approach that we implemented might have a complexity of $O(2 \cdot n \cdot log(2 \cdot n))$, which is many times slower than $O(n)$. Though it is assumed that the number of connections between elements and number of collisions is relatively low, making the worst case impossible.
  - Change priority, trinkle up, pop
    This is the approach used in most implementations of the heap [11, 24, 4] as it is the cleanest way to do so in the case of rare reschedules. It should be noted that this is not a possibility in Python using the provided heap operations. This would imply that the complete heap implementation would have to be manually rewritten (as in [24, 4]). Knowing that we are using Python, this would be an even worse decision since the provided heap operations are provided in compiled and optimised form.
    Our approach is somewhat different, as we don't require a special 'remove index' function and therefore also avoid the associated complexity. On the other hand, our implementation will have a larger heap which might cause some slowdown and a little bit of extra logic to check for invalids. The main advantage is that we don't reimplement the complete heap but just use what is available.

3.3 Major optimisations

The following optimisations have a major impact on the simulation or there is a clear reason to believe that a good improvement can be reached.

3.3.1 Consider only influenced elements

Instead of checking every model in the coupled model whether or not it has received any external event, the loop is rewritten to iterate over the output messages of the transitioned element.
This loop rewriting wasn't trivial, since it was required to have the same semantics as the original, where all messages for one component are sent at once. Forcing us to temporarily collect all messages, to be able to send them all at once.

Rationale

The rationale behind this optimisation is easy to see, as it would be redundant to do a *for* loop over all elements, while a list of all messages that actually happened is available.

Complexity

The complexity change is easy to see, as the sending of messages is no longer $O(components)$, but $O(sent\_messages)$ which is obviously smaller (or in the worst case equal).

Empirical results

By only evaluating the influenced models, the complexity decreases immensly. Both in the *high interconnection* and in the *traffic* experiments, there is a huge speedup.
The *high interconnection* experiment profits because only the output ports are followed instead of checking the linkage of all

elements step-by-step. This way, only one of all elements is actually checked even though the messages are still sent to all connected elements.
In the *traffic experiment*, approximately the same performance gain can be seen. Now only the element that needs to send a message and its connected elements are evaluated. It is logical that the performance keeps increasing if the number of models grows.

3.3.2 Message custom copy

We allow the modeller to define a custom *copy* function, which should be part of the interface of the message that is being sent. Should no *copy* function be found (an *AttributeError* in Python), the default *deepcopy* from Python is used. This is done both for legacy support and usability. The modeller might not want to take the risk (even compared to the increased performance) to define the *copy* function themself, which might be the case if rather complex messages are passed. Sometimes, it is not possible to implement the *copy* function, like when using simple data types instead of classes. Though *deepcopy* should also be able to do this in a more performant way.

Rationale

The DEVS formalism states that copies of messages should be sent, to avoid models influencing other model's messages. This is done in PythonDEVS using the *deepcopy* function, which is rather slow in most cases. Mainly because deepcopy has to do a lot of extra bookkeeping and special checks since it is *too general* for our case.
Since the modeller might possess some extra information about the kind of messages or about what is being done with the content of the message in the model, we might as well offer the possibility to do something with this knowledge.
It might be possible that the message is just a class containing only basic types, which makes it very easy for the user to do the *copy* function manually. This will naturally increase the performance. It might also be possible that the modeller is certain that the content of the message will not be modified, so the *copy* function could just return the message itself, without making a real copy.
Besides PythonDEVS, only VLE offers this kind of safety, but kind of forces the manual *copy* function approach due to the *const* keyword on the bag. VLE just makes the received message a constant, so the model can't change it without making a copy itself. All the other simulators just pass the message without taking a copy. This greatly aids performance, though it shifts all the responsibility to the modeller.
We feel that PythonDEVS offers the most complete solution, as it allows the more experienced users to define one themself, while it offers a perfect default for the inexperienced users. In both cases, safety is guaranteed *unless* the user manually states that it is not required.

Complexity

There is no change in complexity, though it might be possible that the user implements a very efficient copy function which can be performed in $O(1)$ instead of $O(attributes)$, for example if the user is sure that the model will not change the message.
In most cases will this decrease be a constant, though a huge constant.

Empirical results

There is only a performance gain if there is a *copy* function defined, which is done in the traffic model, but not in the other cases (as the message was not a class). A very slight slow-down might be noticed due to the (unsuccesful) checks.
The traffic experiment is the only one that benefits (as this one is the only one with a custom *copy* function), though the gains are relatively small. Should some more info be known, like that the message will never be modified, the copy function could have been defined in such a manner that the speedup was somewhat higher.
Note that this time was (after all other optimisations happended) responsible for a very huge amount of the simulation time. Clearly, if the simulation took e.g. 100 seconds of which 2s is *deepcopy*, this is a relatively small amount. After all other optimisations were done, the simulation time was e.g. 5 seconds, of which 2s are still *deepcopy*. Because of this, implementing this optimisation became useful.

3.3.3 String comparison optimisation

While initializing the simulator, each model is queried for its complete model name. All these names of all the models in the simulation are gathered and sorted. Now the sorted list of names is iterated and every model is assigned an increasing identifier. Due to the algorithm that assigns the identifier, these identifiers have the same ordening of the names, so all comparisons can be done using these integers instead of the strings (of course in the assumption that names cannot change). Note that these identifiers cannot always be used, since they can only be used for the comparison and not for the verbose output.

Rationale

String comparisons are rather expensive, since they should compare the strings character-by-character. If the string are long and contain a long identical prefix, this comparison can take some time. Comparing integers is a very fast operation, so a speedup should be the logical consequence.
As an additional gain from this optimisation, we also receive a *unique ID* for every model, which can be used to uniquely address this model.

Complexity

The complexity of the initialisation will increase, since an extra sort is done with complexity $O(n \cdot log(n))$ to initialize the identifiers. However, the comparsisons that happen afterwards will decrease from $O(length\_string)$ to $O(1)$.

Empirical results

It should be noted that the original PythonDEVS simulator doesn't ever use the string comparison, though this causes an indeterminism bug as Python will compare instances of classes by address instead of by the name (as it should be). However, in the new modified simulator, this comparison is done frequently.
To make the comparison somewhat more fair, the original simulator is patched to include a deterministic sort function.
Even though the speed-up of this feature is either very small or even slows down, there is an increase. The fact that this new feature requires a rather expensive set-up makes this feature look a lot slower. In long-duration simulation runs (with not too many models), it will be clearly visible that there is a speed up.

3.3.4 Direct connection

The use of direct connection allows models to immediately send messages to each other, even when there are multiple coupled models between them. The end-user will not experience any difference except for an increased performance, this is due to the fact that the verbose output will show the exact same behaviour as without direct connection. The most important problem with this feature is that we use Classic DEVS, which requires a *select* function. So the coupled models don't just have a purely structural role, but will also influence the model to transition first. To solve this problem, the approach from [12] is used to implement a *flattened_select* function. Since the *select* functions of the coupled models will only expect models that are their direct children, the *flatten_select* function should mimic this behaviour. This way, the *select* function will be called with the exact same parameters as without direct connection. Besides all this, also a small change was required to the verbose output, since the output would otherwise be dependent on whether or not direct connection was used.
This is a rather intrusive change and the user might not want to use this functionality. As indicated in [12], direct connection in itself will not have a large impact in some situations, since the scheduler will have more elements to process, which could have otherwise been neglected. Even though this performance issue has been fixed by using an heap that only contains active elements, the direct connecting of elements might take some time. If the user models a flat model himself, there is no need to perform all these algorithms, as they will not change anything. It also makes it easier to compare the output of a direct connected version versus the normal version.

Rationale

As mentioned in [12, 21, 4], there is a huge overhead when messages pass through intermediate coupled models. Direct connection allows us to drop all this overhead and pass these messages immediately. Of course, in models that barely use coupled models, the effect will be nearly unnoticable. For deeply nested models that exchange a lot of messages, this approach will have a huge impact. Even more so for huge messages, since messages are (or should be) copied between every coupled model to fully comply to the DEVS formalism.
Like previously mentioned, the main downside is the management of the *select* functions. This means that this approach might not be best when a lot of collisions occur. As we did in the heap approach, the number of collisions should be rather low, so this shouldn't be a real problem.

Complexity

There is no real change in complexity, since the hierarchy should still be passed when executing the *select* functions (so only in the case of a collision). Though the constant cost decreases since no more messages are exchanged between the coupled models. In the cases that no collisions occur, the complexity will no longer be dependent on the hierarchy depth of the couplings between atomic models.

Empirical results

While [12] states that the use of direct connection isn't faster than the normal use of coupled models, this problem has been alleviated by using the heap approach as a scheduler. However, this empirical study doesn't take these changes into account and still use the original eventlist that gets sorted. This causes the direct connection algorithm to be a lot slower in most cases. This is a nice example of a feature that only works good in combination with another feature.
As a side note, there seems to be a small increase in performance when direct connection is used even in flat models. This is due to the fact that a frequent check is of the form 'not direct_connect and X', so Python can stop the evaluation of this expression as soon as it determines that *direct_connect* is true. When direct connection is not used, both of these expressions have to be evaluated, slightly increasing the performance.
The results for direct coupling without any other optimisation are not that shocking and are often slower than the original. This is due to the extra set-up that is required. Note that this causes a high rise in complexity for the deeply hierarchical models. This shows the same behaviour as mentioned in [12], notably that direct connection does not speed up simulation (in the case of PythonDEVS) by a huge amount. However, that slowdown was caused by the bad complexity of the sorting list. With a heap, this direct connection will increase complexity.
To show this result, there is a special *modDC* and *mod* entry. *mod* is the modified simulator without direct connection, while *modDC* is the exact same simulator, but with the direct connection steps. From these cases, it becomes clear that direct connection does vastly increase performance in some cases if it is implemented together with all other optimisations. On the other hand, the *high interconnection* model shows a slowdown for when direct connection is used. Therefore we have decided to keep direct connection as an (by default enabled) optimisation step. Some huge models that are not (or barely) hierarchical would not benefit from the direct connection and the extra set-up would cause major delays.
This is a very good example of the synergy that is possible between different optimisations.

3.3.5 Lazy scheduling

When the heap implementation is used and there is a collision, it is necessary to reinsert the models that weren't selected, since the heap needs to be complete. The elements were popped to begin with, because only the first element can be accessed efficiently in a heap.
With *Lazy scheduling* we don't immediately repush these elements, as it might be possible that they are not influenced by the selected model. If this is the case, the element is just added to a list that is checked the next time this coupled model needs to select a model for its internal transition. Should the model be rescheduled, we don't have to do any extra work, since the element will have already been pushed by the external transition function.

Rationale

When an event is selected for collision, it is first popped, only to be (possibly) pushed again later. To prevent this, we wait for the external transition to happen and check whether or not it is really necessary to push the element again. If the model would be selected for collision multiple times, it would take several *push* and *pop* iterations before the model is eventually selected.
The bad side of this approach is that it is only helpful in case of many collisions and when the number of colliding elements is not much bigger than the list of elements.

Complexity

A combination of *push* and *pop* for these colliding elements would take $O(c * log(n))$ in the worst case, with c being the number of colliding elements and n being the number of elements in the schedulers heap. With our new approach, we reduce this to $O(c)$ in the best case, since we just have to iterate over the colliding elements again and again. Of course, in most cases there will be an influence by the previous model, so this will be slightly reduced.
Also, we have to make some extra checks before we can be certain that the element is not changed or invalid, which has a serious performance penalty.
So while the complexity might lower in some special situations, the general case actually gets slower due to the higher constant cost.

Empirical results

From the empirical results, we can confirm our doubts from the complexity analysis. Most of the time, *Lazy scheduling* actually slows down the simulation in most cases due to all the extra checks that are required. However, a (relatively slight) performance gain can be seen in the case of many collisions that don't reschedule.
Since the results are relatively bad and only interesting in specialised cases, we decided to make this optimisation optional, with it being *disabled* by default.
Note that the lazy optimisation was applied *without* direct connection and above the completely modified version (*mod*), so it should be compared with *mod* and not *modDC*.

3.4 Profiler-induced optimisations

The following changes are mainly found when profiling the code with both cProfile [26] and line_profiler [19]. These changes are not necessarily changes that imply huge performance gains, but were easy to implement and have an often inneglectable impact on the performance of the code. Most of these changes were implemented with the intention of lowering the number of function calls, as this is a rather expensive operation in Python, other programming languages like C, are less vulnerable and might even use inlining to further optimise the code.

3.4.1 Dictionary access

When accessing a dictionary, Python will throw an IndexError if the element could not be found in the dictionary. So it might be better to just access the dictionary in the hopes of obtaining the element, catching the error if it appears. Note that this change has not been done everywhere, since using exceptions is slightly less readable (certainly when multiple exceptions are nested) and will imply a certain overhead, which sometimes doesn't outweight the benefits.

Rationale

Dictionary accesses in the original version were often done based on the *keys()* function of the dictionary. This can become rather slow, as it would search in the list of keys whether or not the element is present in the dictionary. If the element was found, it would search for the element again and retrieve its value. Most models use a rather small dictionary for messages, but if the number of ports would increase drastically, this optimisation would reduce the time it takes to retrieve the value of the port. Models with a small number of ports will actually slow down, since it takes some time to process the exception handling that might be necessary.

Complexity

The Python data structure complexity reference [27] mentions that retrieving an element from a dictionary has average case complexity $O(1)$ and amortized worst cost $O(n)$. But before this access happens, the list of keys is iterated, which has length n and thus takes $O(n)$ to iterate. So accesses to the dictionary would take $O(n)$ in the average case. When this first check is avoided, dictionary accesses can happen in $O(1)$ average case. The amortized worst case of the dictionary access would still be $O(n)$, as is the original case.

Empirical results

As expected, the empirical results are rather disappointing in most models. So it could be deduced that this optimisation isn't really worth the effort in most cases. To show that this optimisation isn't useless, a rather unrealistic test is performed where a lot of messages are passed on a lot of ports. In this situation, there is a huge increase in performance due to this optimisation.

3.4.2 Infinity offloading

The *INFINITY* in the modified PythonDEVS simulator is done using the *float('inf')* notation instead of defining a custom *Infinity* class and singleton. For compatibility reasons, the INFINITY notation is still allowed, as this just became a global variable that is assigned the value *float('inf')*. This change does cause some slight changes to the verbose output, since it will now print *inf* instead of *+INFINITY*.

Rationale

Profiling showed that a lot of time was spent calling the small comparison functions in the *Infinity* class. This is a rather strange approach compared to XSY [18], as this one uses the positive infinity defined in IEEE 754 (floating point numbers). These have the desired semantics and are perfectly supported by the Python interpretter without any extra function calls.
Besides this performance gain, the maintainability also increases, since there is a lot less code.

Complexity

There obviously is no decrease in complexity, as both versions are $O(1)$. However, the constant part decreases immensly.

Empirical results

From this table, it seems that there is no vast increase in performance with this optimisation. This is wrong, since most of the time that is spent in the big models is caused by the sorting, so the decrease in this infinity optimisation is somewhat neglible. There is a huge difference between the modified version with and without this optimisation, since the comparison and calls to this class remain constant. This means that a relatively small decrease would become relatively large if the other optimisations are done. This is another nice example of these synergies.

3.4.3 Caching of model names

The full name of a model is completely static at simulation-time, assuming that no Dynamic Structure DEVS is supported. Therefore, the value is being cached after the first time that it is requested, avoiding the expensive calls up to the root model.

Rationale

This is a rather logical decision, as normally this function will initiate a lot of calls to other models. Should it be a distributed simulator, this would be even more wasteful as it would require network transmission. This would decrease the simulation time for deeply hierarchical models.
Furthermore, due to the other optimisations, we are guaranteed that the cache will be filled *before* actual simulation begins, avoiding delays during the simulation itself.

Complexity

The complexity drops from $O(h)$ to $O(1)$ during simulation-time, but remains $O(h)$ the first time it is called. This first call does actually happen at initialisation-time, due to the string comparison feature. So at simulation-time, this call will always be $O(1)$.

Empirical results

The decrease of simulation time is only visible when the *getModelFullName()* function is called a lot. However, the original PythonDEVS simulator never called this function, except when the verbose output was being generated. This is actually a bug in the original simulator, which was also fixed in this version, thus increasing the use of this function (combined with the string comparison optimisation, it will be barely used).
It should be noted that, to show the difference with this optimisation, verbose output was enabled in the simulation run. This will cause some extra side-effects to slow down the simulation, like the Unix pipe to transfer this data.
The *deeply hierarchical* test[1] finished in $6.85s$ for the original version, while it took $6.792s$ for this optimised version. The performed test was with 10 elements, for a duration of 10000 simulation time units. Clearly, the results aren't very great in most cases due to the increased set-up time, though a minor improvement can be seen if the simulation runs for long enough.
Again, this will be a more effective optimisation should it be done in a distributed simulator.

3.5 Bugfixes

The original PythonDEVS simulator had a few bugs, indeterminisms or 'strange' implementation decisions. These have been solved in the modified version, but not in the original version unless explicitly mentioned.
The following bugs have been fixed, compared to the original simulator:

- A comparison of INFINITY in the models (nearly) always failed, since the infinity singleton is also *deepcopied*, meaning that what should be a singleton gets copied. This is solved by using the float representation of infinity.
- Indeterminism in the imminent children: as previously mentioned, the original simulator would sort the elements by address of the object instead of name.
- *Time advance* didn't check for < 0: as mentioned in the comparison, this was needed to pass the testing framework.
- Deepcopy also performed for External Input/Output coupling: as mentioned in the comparison, this was needed to pass the testing framework.
- The elapsed time variable was set to 0 before the internal transition function was called. Though the DEVS formalism doesn't state that this variable can be used, it is rather strange to provide this variable when it cannot always be used. Note that this doesn't violate the DEVS formalism, as this value should be exactly equal to the *time advance* function of this model.
- There was an undefined variable in one of the error messages

3.6 Results

As previously mentioned, most of these changes cooperate in such a way that the individual changes don't really have any noticable impact (or even slow down a little). The combined result is a lot better and doesn't only speed up the simulation, but also decreases the complexity. Most of these examples are rather artificial, in order to better show the decrease in complexity.
All tables will have top entries representing the used configuration. This has the form X/Y, with X being a model specific parameter that influences the number of models (model-specific) and Y being the end time of the simulation. The results were obtained using an *Intel i5-2500 3.3GHz* processor, with a 5 run average. The times were obtained within the Python code itself (wall clock time), so there is no extra delay for loading Python itself.

[1] We didn't include this data in the comparison table at the end, since it requires verbose output.

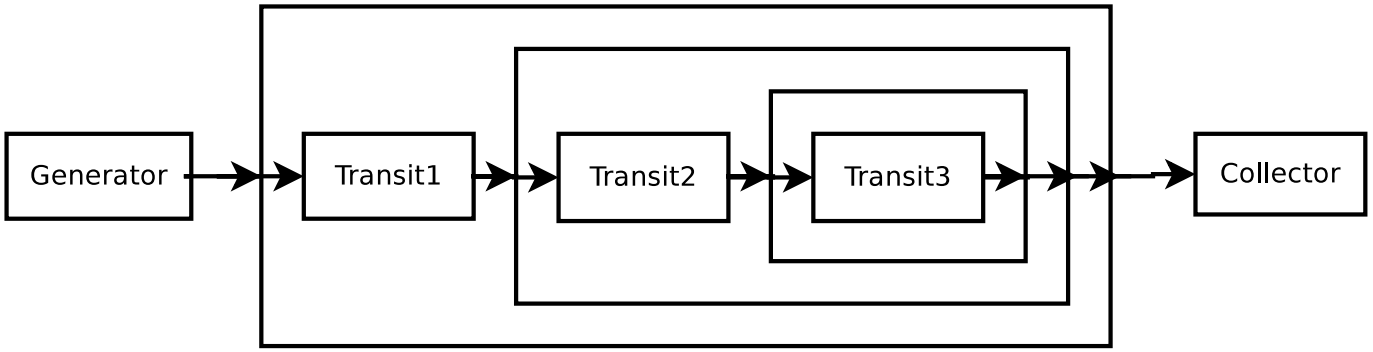

Figure 3.1: A deep hierarchy model with $X = 3$

The analysis has already happend in the relevant parts and since the completely modified version doesn't add any extra changes, another analysis here would just replicate the ones in the *Empirical results* sections.
Note that the final mod version will also call the Cython version of the compiler if it is available (which it was in our case), so this contributes to the higher simulation speed. Note that there is a seperate entry 'Cython' in these tables, this is a version of the simulator with the model under testing compiled too, which makes some kind of a 'specialised simulator'. For most users, it isn't recommended to do this, as it greatly limits the flexibility that Python provides and requires constant recompiles. However, it is included for reference.

3.6.1 Deep hierarchy

The deep hiearchy test builds a model in a recursive way. Each Coupled model will have a Transit element, which just holds the message for 1 time unit, and another Coupled model but with a lower value, $X - 1$. This causes extremely deep hierarchical models. The main purpose of this test is to show the advantages of direct connection, as this would avoid all these intermediate coupled models completely at the cost of a (slightly) higher set-up. The most notable result is that Direct Connection alone has *nearly no impact* the simulation. It only becomes interesting when combined with the finished simulator.
Note that for some of these entries, we had to raise the maximum recursion depth of the Python interpreter. Should direct connection have been used, this would of course not have proved a problem.

	100/100	**100/1000**	**200/100**	**200/1000**
vanilla	1.876	19.0	6.768	68.756
mod	0.188	1.844	0.366	3.594
modDC	0.014	0.08	0.022	0.096
modLazy	0.19	1.858	0.37	3.628
Cython	0.012	0.082	0.018	0.104
cache	0.126	1.228	0.258	2.48
copy	0.13	1.246	0.258	2.48
DC	0.87	8.68	3.234	32.632
dict	0.132	1.292	0.268	2.6
heap	0.254	2.356	0.552	4.872
infinity	0.11	1.032	0.214	2.066
influence	0.13	1.24	0.256	2.482
string	0.144	1.258	0.32	2.556

Table 3.1: Performance with very deeply nested models (times in seconds)

3.6.2 High interconnection

The high interconnection test builds a model with X elements, all of which are coupled together, each on its own port. The actual data flow is from element n to element $n+1$. This nicely shows the reduction in complexity when only the active ports are being checked instead of every element's port. Needless to say, this test will highly favor the changed influence algorithm.

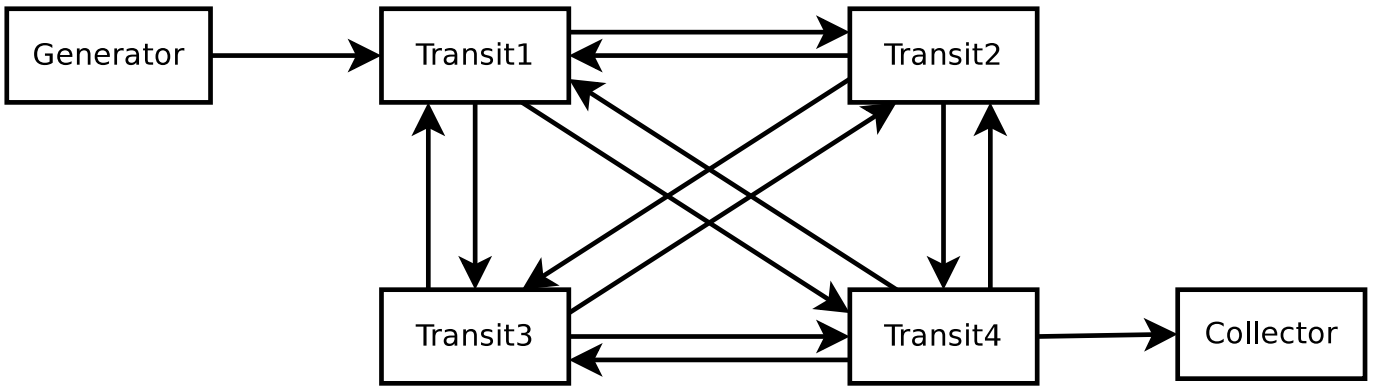

Figure 3.2: A high interconnect model with $X = 4$

	4/100	**10/100**	**100/100**	**150/100**
vanilla	0.02	0.062	5.004	11.446
mod	0.006	0.008	0.136	0.364
modDC	0.006	0.008	0.15	0.424
modLazy	0.006	0.008	0.136	0.364
Cython	0.006	0.008	0.144	0.4
cache	0.016	0.056	4.898	11.436
copy	0.02	0.062	4.972	11.402
DC	0.02	0.062	5.076	11.686
dict	0.018	0.062	5.074	11.658
heap	0.016	0.054	4.86	11.314
infinity	0.016	0.056	4.914	11.28
influence	0.018	0.034	1.228	2.618
string	0.018	0.058	4.906	11.482

Table 3.2: Performance in the case of many interconnections between models (times in seconds)

3.6.3 Traffic experiment

This model uses the same model that was used in the comparison of simulators, but makes it more flexible by making the number of road segments inbetween the generator and collector variable as a parameter. This is probably a more realistic model with very few (if any) collisions, so this model will give good results for the heap optimisations. As complex messages are passed, the user-defined copy function will also speed things up.
Since the models do have some computation inside them, the speedup will be slightly less than in other tests.

	3/100	**3/1000**	**3/10000**	**50/500**	**200/100**
vanilla	0.102	1.008	10.862	36.868	48.222
mod	0.028	0.286	3.454	2.064	0.796
modDC	0.026	0.270	3.326	1.958	0.746
modLazy	0.03	0.288	3.502	2.088	0.792
Cython	0.026	0.254	3.156	1.844	0.698
cache	0.092	0.94	9.9	31.383	40.572
copy	0.084	0.852	9.128	35.692	48.082
DC	0.1	1.012	10.688	36.448	47.742
dict	0.102	1.03	10.996	38.042	50.22
heap	0.088	0.882	9.444	22.48	27.37
infinity	0.078	0.796	8.532	31.06	41.59
influence	0.096	0.98	10.398	19.826	21.98
string	0.1	1.014	10.76	36.698	48.22

Table 3.3: Performance in the traffic experiment (times in seconds)

	10/1000	30/1000	50/1000	75/1000
vanilla	0.18	0.808	1.88	3.812
mod	0.034	0.086	0.14	0.214
modDC	0.036	0.088	0.148	0.234
modLazy	0.036	0.084	0.132	0.198
Cython	0.034	0.084	0.14	0.222
cache	0.132	0.558	1.268	2.5
copy	0.182	0.812	1.91	3.85
DC	0.176	0.814	1.9	3.852
dict	0.202	1.008	2.448	5.048
heap	0.08	0.26	0.488	0.862
infinity	0.14	0.636	1.494	2.958
influence	0.298	1.668	4.124	8.714
string	0.178	0.806	1.88	3.868

Table 3.4: Performance with high number of collisions (time in seconds)

3.6.4 High number of collisions

An interesting point in Classic DEVS is when multiple collisions occur. This is a major source of inefficiency in many algorithms, since we should have already found all other models that have to transition, but have to recheck all of them due to possible influences. This source of inefficiency is solved in Parallel DEVS.
In this test, we have a generator, which has 10 output ports. On each of these ports, a tenth of all processors is connected. As soon as a processor has received a message, it will have an internal transition, which doesn't generate any output. This way, one tenth of all processors will have a collision.
This setup is ideal for the *lazy optimisation* as mentioned in section 3.3.5, which will not reinsert colliding elements in the heap immediately.

3.6.5 High messages

This model is a slight modification of the *high intercoupling* model. Instead of only sending and receiving messages on port 0, every output port receives the data and every input port is checked. The behaviour is exactly the same, since only the data from port 0 is actually used. In this model, the optimisation for the dictionaries is clearly visible and this is actually the only reason why this case is included...

	4/100	10/100	100/100
vanilla	0.036	0.256	160.10
mod	0.014	0.048	3.092
modDC	0.014	0.048	3.096
modLazy	0.014	0.048	3.064
Cython	0.016	0.048	2.978
cache	0.034	0.248	160.484
copy	0.036	0.256	160.532
DC	0.036	0.258	162.012
dict	0.03	0.13	11.114
heap	0.036	0.254	159.73
infinity	0.034	0.25	159.408
influence	0.026	0.106	7.526
string	0.036	0.252	157.802

Table 3.5: Performance when a lot of messages are sent around (times in seconds)

3.6.6 City traffic

To test a somewhat more realistic model, we decided to modify the city map generator that was used as a case study in [25] to also produce DEVS code (for a subset of possible models), so that it is compatible with PythonDEVS. This model is somewhat of an extension to the *traffic experiment*, as it combines multiple *road stretches* at an *intersection* that has *traffic lights*. The cars

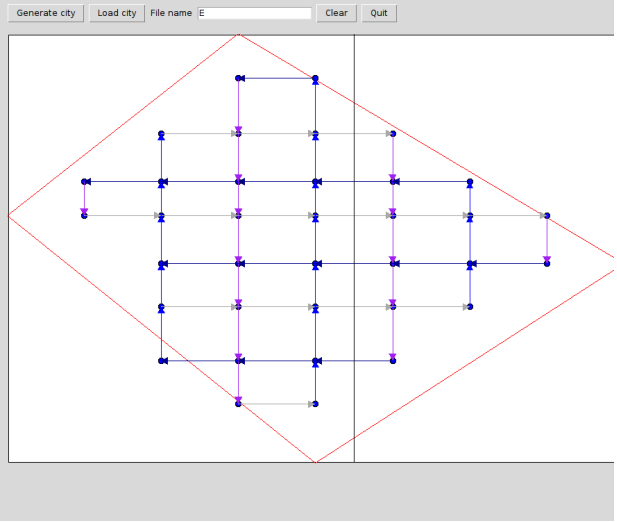

Figure 3.3: City 1

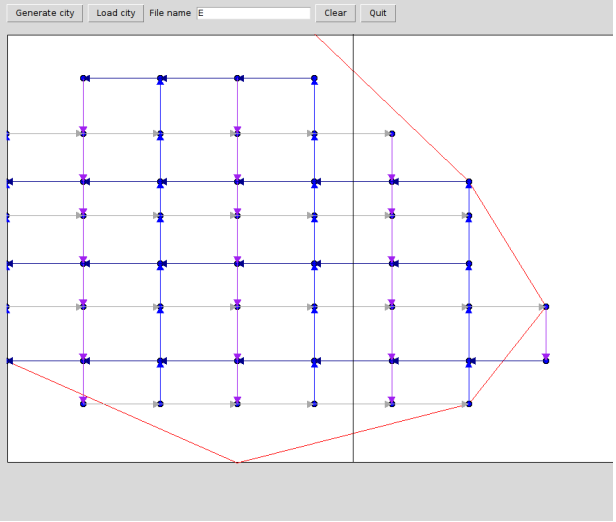

Figure 3.4: City 2

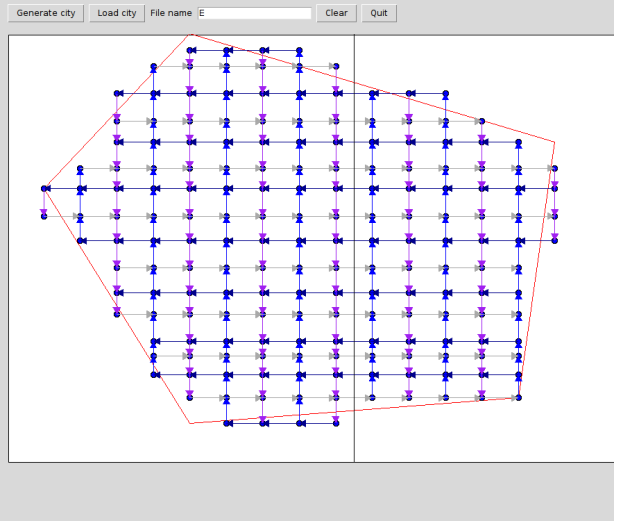

Figure 3.5: City 3

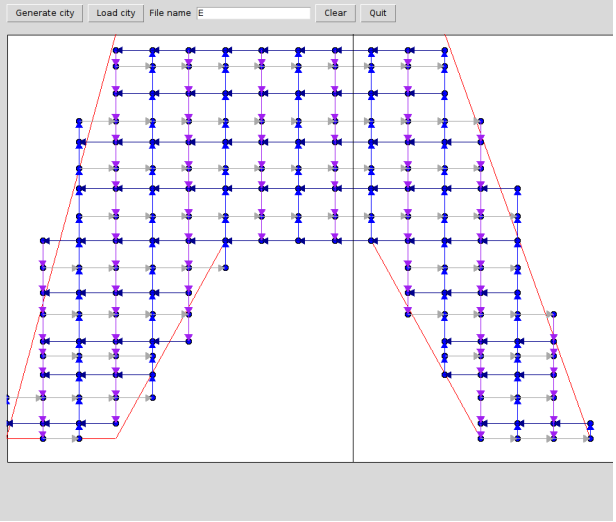

Figure 3.6: City 4

get generated in *residential* buildings and have to arrive at *commercial* buildings. All cars receive a path as soon as they leave the building (so they know how to drive). Only a subset is supported in the sense that some parameters in the city map generator don't influence the DEVS models. These tests are conducted only on the important versions, as the performance improvement in other versions will not be that clear. Note that, while the simulation should use a variety of random numbers for e.g. inter arrival time, preferred speed, traffic light delays, ... we simply set this to a fixed value, to make sure that the *exact* same result is found with the different simulators. This should not normally be a problem, as the simulators work completely deterministic, but there was a slight difference in the order of initialisation of variables, making it possible that an other situation is being simulated. We only performed these tests for both the vanilla and the completely finished 'default' modified version, because the other versions are relatively unrelevant as they will produce approximatelly the same results.

	city 1	**city 2**	**city 3**	**city 4**
vanilla	16.9	35.8	459.6	334.6
mod	1.68	2.77	16.32	11.83

Table 3.6: Performance in different cities (times in seconds)

3.6.7 Comparing with ADEVS

Since ADEVS is the fastest simulator, it is our goal to achieve approximately the same performance. Of course, this is somewhat difficult due to the fact that PyDEVS uses Python, while ADEVS uses C++. Therefore, we will only try to achieve the same complexity, or big-Oh notation. From our previous plots, it wasn't clear how ADEVS performed, since its simulation time stayed very low. Therefore, we include a plot of only ADEVS. This plot makes it clear that it has a linear complexity.
Compared to our modified version of PyDEVS, this didn't seem that good at all, since PyDEVS seems to have a complexity somewhere between $O(n \cdot log(n))$ and $O(n^2)$. This came as a surprise, since PyDEVS uses (approximately) the same algorithms as ADEVS. Other reasons came to mind, though none of them should cause a different complexity, for example

- Different implementation language: Python is known to be slower than C
- Different interface: a lot of time in PyDEVS is spent in the *peek* and *poke* functions
- Different optimisations: it is possible that the C compiler performs some optimisations, like faster memory allocation

The only possibility that we could see, would be in the different formalisms. PyDEVS uses Classic DEVS, while ADEVS uses Parallel DEVS. Since the simulation didn't run on a multicore system, we would have guessed that this parallelisation was unused due to hardware limitations.
Finally, we found the solution, as it is due to a problem in the formalism itself. In case of a collision of internal events, Classic DEVS will select only one of them, while Parallel DEVS will execute all of them. The implications are that we will have to fetch a list of all *colliding elements*, select one and throw away the rest. In the original PyDEVS, this wasn't a real problem because a *check all* strategy was used. In our new version however, we pop *all* colliding elements and have to *repush* all but one of these elements. Parallel DEVS on the other hand, would never cause any *repush*es. It is interesting to mention that accesses to the heap are the *only* actions in the complete simulator that requires to take into account all models, so the problem had to be localised there.
In conclusion: the simulator itself is not the bottleneck for our complexity, but the *formalism*, even on *single-core, non-distributed systems*.

Varying the tests

In order to isolate this problem, we modified the models so that they would cause (nearly) no collisions. We reran the model (for a longer duration in simulated time, to get more accurate results) and found out that this was the cause of our increased complexity. With these modifications, the modified PyDEVS simulator has the same complexity as the ADEVS simulator (of course, up to a constant factor).
The model's delay before sending the received event was increased from 0.66 to 0.66666666, in order to minimise collisions. As a side note, the number of events that has to be processed will drop slightly, which is why the *ADEVS* plot for the collisions will be slightly higher. However, this increase is not enough to completely alter the complexity of the simulation, which causes our results for *PyDEVS* to be relevant.
Even though a model that never has any collisions is rather artificial, we could conclude that all other algorithms that are being used are similar to those used in ADEVS and that we could expect the same complexity should we have implemented Parallel DEVS in PyDEVS.

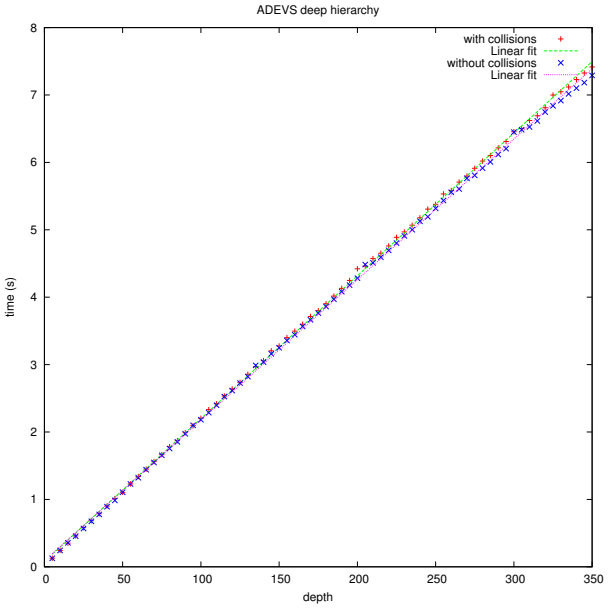

Figure 3.7: ADEVS deep hierarchy complexity

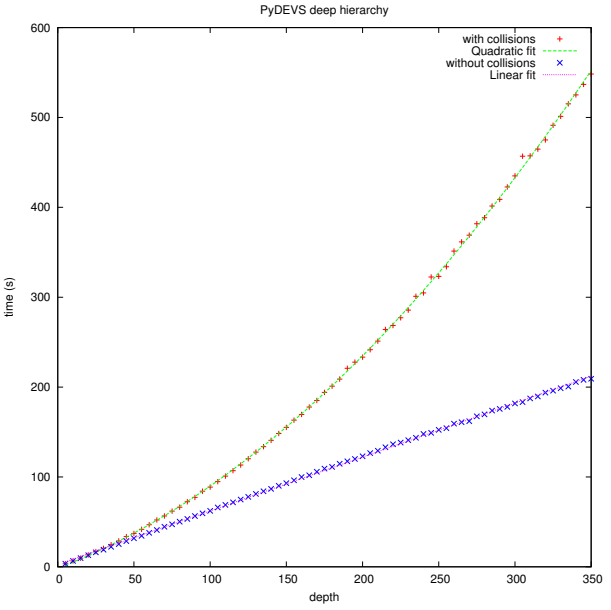

Figure 3.8: PyDEVS deep hierarchy complexity

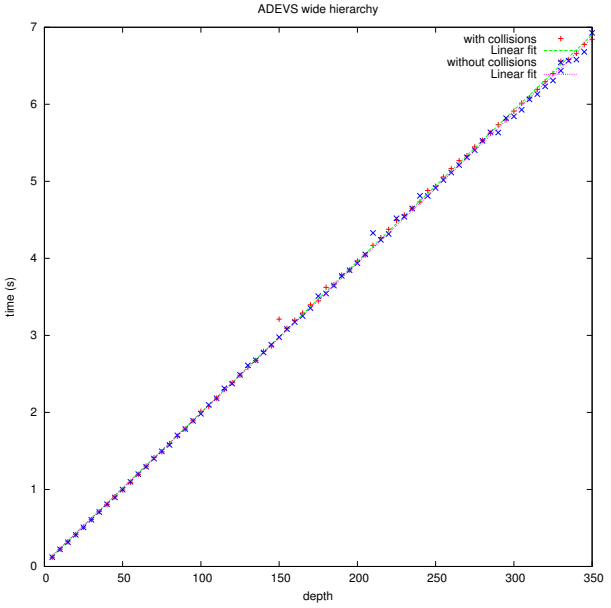

Figure 3.9: ADEVS wide models complexity

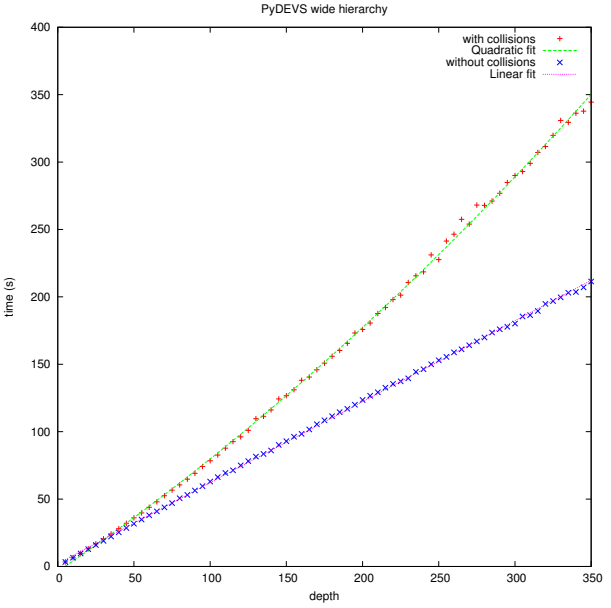

Figure 3.10: PyDEVS wide models complexity

3.7 Conclusion

We showed the different optimisations that were implemented to the original PythonDEVS simulator in an effort to reduce its time complexity. We can conclude that this effort has succeeded, as the PyDEVS simulation kernel has a time complexity comparable to *ADEVS*, which is (one of) the fastest simulators available.
All of these optimisations have a seperate version of the simulation kernel that shows the impact of this optimisation in certain situations. Afterwards, the complete version is compared to the original one in relatively 'real-life' models simulating a cities traffic system, which showed a simulator that is up to about 25 times faster for huge cities.
An interesting conclusion that we could draw from our complexity analysis is that the *Classic DEVS* formalism constrains the complexity due to the possible collisions that require an expensive intervention of the *select* function and requires some special attention.

Bibliography

[1] Devs-suite. http://devs-suitesim.sourceforge.net/.

[2] Pythondevs. http://msdl.cs.mcgill.ca/projects/projects/DEVS/.

[3] Vle: Virtual laboratory environment. http://www.vle-project.org/wiki/Main_Page.

[4] James J. Nutaro Alexander Muzy. Algorithms for efficient implementations of the devs & dsdevs abstract simulators. 2005.

[5] Javier Ameghino and Gabriel Wainer. Application of the cell-devs paradigm using n-cd++. In *In Proceedings of the 32 nd SCS Summer Computer Simulation Conference*, 2000.

[6] 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*, WSC '95, pages 781–785, Washington, DC, USA, 1995. IEEE Computer Society.

[7] Fernando J. Barros. Modeling formalisms for dynamic structure systems. *ACM Trans. Model. Comput. Simul.*, 7(4):501–515, October 1997.

[8] Fernando J. Barros. Abstract simulators for the dsde formalism. In *Proceedings of the 30th conference on Winter simulation*, WSC '98, pages 407–412, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[9] Jean-Sébastien Bolduc and Hans Vangheluwe. The modelling and simulation package pythondevs for classical hierarchical devs. Technical report, MSDL Technical Report, 2001.

[10] Matías Bonaventura, Gabriel A. Wainer, and Rodrigo Castro. Advanced ide for modeling and simulation of discrete event systems. In *Proceedings of the 2010 Spring Simulation Multiconference*, SpringSim '10, pages 125:1–125:8, San Diego, CA, USA, 2010. Society for Computer Simulation International.

[11] Frank M. Carrano. *Data Abstraction & Problem Solving with C++*. Addison Wesley, 2006.

[12] Bin Chen and Hans Vangheluwe. Symbolic flattening of devs models. In *2010 Summer Simulation Multiconference*, SummerSim '10, pages 209–218, San Diego, CA, USA, 2010. Society for Computer Simulation International.

[13] Yu Chen and Hessam S. Sarjoughian. A component-based simulator for mips32 processors. *SIMULATION*, 2010.

[14] A.C. Chow. Abstract simulator for the parallel devs formalism. *AI. Simulation, and Planning in High Autonomy Systems*, 1994.

[15] 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*, WSC '94, pages 716–722, San Diego, CA, USA, 1994. Society for Computer Simulation International.

[16] Ezequiel Glinsky and Gabriel Wainer. Devstone: a benchmarking technique for studying performance of devs modeling and simulation environments.

[17] Jan Himmelspach. Sequential processing of pdevs models.

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

[19] Robert Kern. Python line profiler. http://packages.python.org/line_profiler/, 2012.

[20] Yvan Labiche and Gabriel Wainer. Towards the verification and validation of devs models. In *in Proceedings of 1st Open International Conference on Modeling & Simulation, 2005*, pages 295–305, 2005.

[21] Wan Bok Lee and Tag Gon Kim. Simulation speedup for devs models by composition-based compilation. 2003.

[22] Xiaobo Li, Hans Vangheluwe, Yonglin Lei, Hongyan Song, and Weiping Wang. A testing framework for devs formalism implementations. In *Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, TMS-DEVS '11, pages 183–188, San Diego, CA, USA, 2011. Society for Computer Simulation International.

[23] Alexandre Muzy and Gabriel Wainer. Comparing simulation methods for fire spreading across a fuel bed. In *In Proceedings of AIS'2002*, pages 219–224, 2002.

[24] James J. Nutaro. Adevs. `http://www.ornl.gov/~1qn/adevs/`, 2012.

[25] Ernesto Posse. *Modelling and simulation of dynamic structure discrete-event systems*. PhD thesis, School of Computer Science, McGill University, October 2008.

[26] Python. Python cprofile. `http://docs.python.org/library/profile.html#module-cProfile`, 2012.

[27] Python. Python time complexity. `http://wiki.python.org/moin/TimeComplexity`, 2012.

[28] 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*, SCSC, pages 367–374, San Diego, CA, USA, 2007. Society for Computer Simulation International.

[29] RTSync. `http://www.rtsync.com/`.

[30] Hui Shang and Gabriel Wainer. A model of virus spreading using cell-devs. In Vaidy Sunderam, Geert van Albada, Peter Sloot, and Jack Dongarra, editors, *Computational Science ICCS 2005*, volume 3515 of *Lecture Notes in Computer Science*, pages 145–201. Springer Berlin / Heidelberg, 2005. 10.1007/11428848_50.

[31] Hui Shang and Gabriel Wainer. A simulation algorithm for dynamic structure devs modeling. In *Proceedings of the 38th conference on Winter simulation*, WSC '06, pages 815–822. Winter Simulation Conference, 2006.

[32] Hongyan Song. Traceplotter. `http://msdl.cs.mcgill.ca/people/bill`, 2006.

[33] Luc Touraille, Mamadou K. Traoré, and David R. C. Hill. Enhancing devs simulation through template metaprogramming: Devs-metasimulator. In *2010 Summer Simulation Multiconference*, SummerSim '10, pages 394–402, San Diego, CA, USA, 2010. Society for Computer Simulation International.

[34] Alejandro Troccoli and Gabriel Wainer. Implementing parallel cell-devs. In *Proceedings of the 36th annual symposium on Simulation*, ANSS '03, pages 273–, Washington, DC, USA, 2003. IEEE Computer Society.

[35] A. M. Uhrmacher. Dynamic structures in modeling and simulation: a reflective approach. *ACM Trans. Model. Comput. Simul.*, 11(2):206–232, April 2001.

[36] Hans Vangheluwe. The discrete event system specification (devs) formalism.

[37] Hans Vangheluwe. Traffic simulation assignment. `http://msdl.cs.mcgill.ca/people/hv/teaching/MSBDesign/COMP763B2008/assignments/assignment3/`, 2008.

[38] G. Wainer and N. Giambiasi. Discrete event modelng and simulation technologies. chapter Timed cell-DEVS: modeling and simulation of cell spaces, pages 187–214. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[39] Gabriel Wainer. Cd++. `http://cell-devs.sce.carleton.ca/mediawiki/index.php/Main_Page`, 1999.

[40] Gabriel Wainer and Norbert Giambiasi. Application of the cell-devs paradigm for cell spaces modelling and simulation. Technical report, 2001.

[41] Gabriel Wainer, Liliana Morihama, Viviana Passuello, and Pabelln I. Ciudad Universitaria. Automatic verification of devs models.

[42] Gabriel A. Wainer. Modeling and simulation of complex systems with cell-devs. In *Proceedings of the 36th conference on Winter simulation*, WSC '04, pages 49–60. Winter Simulation Conference, 2004.