UNIVERSITEIT ANTWERPEN

MASTER'S THESIS

Activity-aware DEVS simulation

Author:
Yentl VAN TENDELOO

Promotor:
Prof. Dr. Hans VANGHELUWE
Co-Promotor:
Dr. Kurt VANMECHELEN

A thesis submitted in fulfillment of the requirements for the degree of Master of Science: Computer Science in the Modelling, Simulation and Design Lab Department of Mathematics and Computer Science

June 28, 2014

Abstract

The common view on modelling and simulation of dynamic systems is to focus on the specification of the state of the system and its transition function. Although some interesting challenges remain to efficiently and elegantly support this view, we consider in this thesis that this problem is solved. Instead, this thesis focuses on a new point of view on dynamic system specifications: the computational activity exhibited by their discrete event simulation. Such a viewpoint opens up new apportunities for analyzing, modelling and simulating systems.

This thesis develops an efficient, distributed simulation framework based on the Discrete EVent system Specification (DEVS) formalism. Adding knowledge about locality of computation to simulation models allows for efficient and automated load balancing in distributed simulation.

Furthermore, several features are added to the simulation kernel to make it more usable, applicable and more efficient in a variety of domains. These features currently call for a domain-specific simulators. They also make it possible to analyze the effect of activity-awareness when more advanced simulation algorithms and a more elaborate set of features are in place.

Samenvatting

De doorgaanse kijk op modelleren en simuleren van dynamische systemen, focust zich op de specificatie van de toestand van een systeem en zijn transitie functie. Hoewel er nog interessante uitdagingen zijn om dit efficient en elegant te ondersteunen, veronderstelt deze thesis dat dit probleem opgelost is. Deze thesis zal zich focussen op een nieuwe visie tegenover het specifieren van dynamische systemen: de activiteit die merkbaar is vanuit de discrete event simulatie. Zulk een oogpunt introduceert een nieuwe manier voor het analyseren, modelleren en simuleren van systemen.

In deze thesis wordt een efficiente en gedistribueerde simulatie architectuur gebouwd, gebasseerd op het Discrete EVent system Specification (DEVS) formalisme. Door het toevoegen van kennis over de lokaliteit van berekeningen aan het gesimuleerde model, wordt het mogelijk om efficientere en geautomatiseerde balancering uit te voeren in gedistribueerde simulaties.

Daarbovenop worden verschillende opties toegevoegd aan de simulator, zodat deze bruikbaarder en efficienter wordt in een wijd spectrum aan domeinen, waar momenteel een domein-specifieke simulator voor nodig is. Hierdoor wordt het ook mogelijk om het effect van activiteit op geavanceerde simulatie algorithmen, met een varieteit aan additionele functionaliteit, te bestuderen.

Acknowledgments

First of all, I would like to thank my promotor, Hans Vangheluwe. He has been my mentor during the last three years for my Bachelor's thesis, my research internships at the MSDL research group, and my Master's thesis. He has introduced me to the academic world, sparked my interest in modelling and simulation, and has given me the opportunity to contribute to interesting research projects. I also want to thank him for giving me the opportunity to attend the DS-RT conference in Delft. I would like to thank my co-promoter, Kurt Vanmechelen, for his contributions to the distributed benchmarking. I would also like to thank my friends and family for their moral support throughout my whole study career.

Contents

1 Introduction **10**
- 1.1 Context 10
- 1.2 Problem statement 10
- 1.3 Alternatives 10
- 1.4 Objectives and contributions 11
- 1.5 Technical remarks 11
- 1.6 Outline 11
- 1.7 The DEVS formalism 12
- 1.8 Distributed simulation 18

2 Design **24**
- 2.1 UML Class Diagram 24
- 2.2 UML Sequence Diagram 27
- 2.3 Modifications to Mattern's algorithm 29
- 2.4 Model distribution 31
- 2.5 Classification 33
- 2.6 Modelling PythonPDEVS 33

3 Functionality **38**
- 3.1 Classic and Parallel DEVS 38
- 3.2 Realtime simulation 39
- 3.3 Tracing 42
- 3.4 Dynamic Structure DEVS 43
- 3.5 Checkpointing 46
- 3.6 Nested Simulation 47
- 3.7 Termination condition 48
- 3.8 Simulation continuation 50
- 3.9 Model reinitialization 51
- 3.10 Random numbers 51
- 3.11 Halted simulation detection 52
- 3.12 Transfer functions 53
- 3.13 Visualization 54
- 3.14 Documentation 55
- 3.15 External contributions 55
- 3.16 Feature matrix 56

4 Performance **58**
- 4.1 Methodology 58
- 4.2 Local simulation 58

4.3 Distributed simulation 63
4.4 Domain information 69
4.5 Benchmarks 75
4.6 Comparison to adevs 78

5 Activity **83**
5.1 Definitions 83
5.2 Applications 85
5.3 Computation 92
5.4 Benchmarks 94

6 Conclusion **102**
6.1 Recap 102
6.2 Contributions 103
6.3 Future work 103

A Examples **105**
A.1 Model 106
A.2 Experiment 117
A.3 Scheduler 118
A.4 Static Allocator 120
A.5 Dynamic Allocator 121
A.6 Activity Relocator 122
A.7 Last state Relocator 123
A.8 Tracer 125

B Citylayout generator **127**

List of Figures

2.1 Class diagram of the PythonPDEVS simulation kernel . . . 25
2.2 Sequence diagram for the local simulation loop . . . 28
2.3 Sequence diagram for an inter-node message . . . 28
2.4 Sequence diagram for a relocation from node 1 to node 2, with node 0 as controller . . . 29
2.5 Naive modification of Mattern's algorithm in all circumstances . . . 29
2.6 Problem in the naive modification of Mattern's algorithm . . . 30
2.7 Mattern's algorithm (modified) in all circumstances . . . 30
2.8 Mattern's algorithm with transient red messages . . . 30
2.9 Mapping between atomic models and LPs . . . 32

3.1 Comparison of different simulation speeds . . . 39
3.2 Different realtime back-ends . . . 41
3.3 A GUI for the traffic light . . . 41
3.4 XML trace as visualized in XMLTracePlotter . . . 44
3.5 VCD trace as visualized in GTKWave . . . 44
3.6 Cell trace as visualized with Gnuplot . . . 44
3.7 Direct connection introduces problems for Dynamic Structure DEVS . . . 45
3.8 Distributed termination function . . . 49
3.9 Example of continuing a simulation at simulation time 5 . . . 50
3.10 Example of transfer functions . . . 53
3.11 Visualization of a model . . . 54
3.12 DEVSimPy . . . 56
3.13 Example model in $AToM^3$ using the DEVS formalism . . . 57

4.1 Direct connection flattens the hierarchy . . . 60
4.2 Direct connection and its impact on distributed simulation . . . 60
4.3 Example of Quantized DEVS . . . 63
4.4 Allocators allow more elegant code . . . 64
4.5 Naive messaging vs. message grouping . . . 67
4.6 Problematic situation in termination algorithm if no second phase is used . . . 68
4.7 The extendability of the PythonPDEVS simulation kernel . . . 70
4.8 Comparison of message copy methods . . . 72
4.9 Comparison of state saving methods . . . 72
4.10 Performance comparison with and without memoization . . . 75
4.11 Fire spread benchmark with quantum 1.0 . . . 76
4.12 Fire spread benchmark without quantization . . . 76
4.13 A DEVStone model with depth 3 and width 2 . . . 76
4.14 DEVStone results with collisions . . . 77
4.15 DEVStone results without collisions . . . 77

4.16 PHOLD model, both the distributed and local version . . . 77
4.17 Performance for the PHOLD model on a cluster . . . 78
4.18 Comparison to adevs with *ta* of 1 . . . 82
4.19 Comparison to adevs with random *ta* . . . 82

5.1 Activity scanning in the two-phase approach . . . 84
5.2 Different activity definitions . . . 85
5.3 Application of qualitative activity can reduce complexity from $O(n)$ to $O(r)$ . . . 86
5.4 Activity relocation on a synthetic ring model . . . 89
5.5 Quantitative external example . . . 91
5.6 Activity prediction to filter out unnecessary relocations . . . 93
5.7 Different activity computation methods . . . 94
5.8 Synthetic benchmark for qualitative activity . . . 95
5.9 Fire spread benchmark for qualitative activity . . . 96
5.10 Figure 5.9 zoomed in . . . 96
5.11 Polymorphic scheduler performance . . . 97
5.12 Figure 5.11 zoomed in . . . 97
5.13 Scheduler performance during step-wise simulation . . . 97
5.14 Speedup with and without activity tracking for varying number of nodes . . . 97
5.15 Activity tracking on a 5-node synthetic model . . . 98
5.16 Activity tracking on a 50-node synthetic model . . . 98
5.17 Activity distribution with activity tracking (3 nodes) . . . 98
5.18 Activity distribution without activity tracking (3 nodes) . . . 98
5.19 A road stretch (source: `http://msdl.cs.mcgill.ca/people/hv/teaching/MoSIS/assignments/DEVS/`) . . . 99
5.20 Example city layout model for 2 cars over 2 nodes . . . 100
5.21 City layout benchmark for quantitative internal activity (3 nodes) . . . 101
5.22 City layout benchmark for quantitative internal activity (5 nodes) . . . 101

List of Algorithms

1.1 Classic DEVS atomic model abstract simulator . . . 15
1.2 Classic DEVS coupled model abstract simulator (aka coordinator) . . . 16
1.3 Classic DEVS root coordinator . . . 16
1.4 Mattern's algorithm when sending a message from P_i to P_j . . . 22
1.5 Mattern's algorithm when receiving a message at P_i . . . 22
1.6 Mattern's algorithm when receiving a control message $< m_{clock}, m_{send}, count >$ in P_i . . . 23
1.7 Mattern's algorithm when receiving a control message $< m_{clock}, m_{send}, count >$ in P_{init} . . . 23
2.1 Modified Mattern's algorithm when sending a message from P_i to P_j . . . 31
2.2 Modified Mattern's algorithm when receiving a message at P_i . . . 31
2.3 Modified Mattern's algorithm when receiving message $< m_{clock}, m_{send}, wait_vector, accumulate_vector >$ in P_i . . . 31
2.4 Modified Mattern's algorithm when receiving message $< m_{clock}, m_{send}, wait_vector, accumulate_vector >$ in P_{init} . . . 32
2.5 Modelled PythonPDEVS internal transition function . . . 35
2.6 Modelled PythonPDEVS external transition function . . . 36
2.7 Modelled PythonPDEVS output function . . . 36
2.8 Modelled PythonPDEVS time advance . . . 36
3.1 Dynamic Structure DEVS simulation algorithm . . . 46
3.2 Random number generator code . . . 52
4.1 Transition processing algorithm with memoization . . . 74

Listings

3.1 An example interrupt file . . . 42
3.2 An example invocation of the interrupt method . . . 42
3.3 An example invocation of the interrupt method in Tk . . . 42
3.4 Verbose trace of a single timestep in the simulation . . . 43
3.5 A simple nested simulation invocation . . . 48
3.6 A verbose trace after a modification happened with continuation . . . 50
4.1 Example adevs model from its documentation slightly shortened to fit on a single page . . . 80
4.2 Example PythonPDEVS model that is semantically equal to Listing 4.1 . . . 81
5.1 Comparison of different activity measuring methods . . . 87
5.2 Pseudo-code in Python indicating the place of both activity calls . . . 88
A.1 The citylayout benchmark model including some domain-specific optimizations . . . 106
A.2 A simple experiment file . . . 117
A.3 A custom scheduler . . . 118
A.4 A custom static allocator . . . 120
A.5 A custom dynamic allocator . . . 121
A.6 A custom relocator . . . 122
A.7 A custom ”last state” relocator . . . 123
A.8 A custom tracer . . . 125
B.1 The citylayout benchmark model generator . . . 127

1
Introduction

This chapter presents the context of this thesis and the problems it addresses. A brief introduction is given to the DEVS formalism and to distributed synchronization protocols.

1.1 Context

The dynamics of complex systems are modelled explicitly using appropriate formalisms to be simulated on digital computers. Simulation thus opens new perspectives on science, inducing a methodology shift in the scientific community. It is now widely accepted that scientific analysis is founded on three pillars: theory, experimentation and simulation. [18]

1.2 Problem statement

With the growing demand for computing resources by modern simulation applications, the need for efficient simulators increases. For this reason, Parallel and Distributed Simulation (PaDS) becomes necessary, not only for the performance improvement it may yield, but also as the state-space of current large-scale models becomes too large to represent in a single computational node.

This does however not remove the need for efficient sequential simulation algorithms for two reasons:

1. Not all kinds of models will become more efficient when parallelized. Examples of this are very tightly coupled models, or models that do not have enough computation in their transition functions to warrant parallelization.
2. Distributed simulation still requires efficient algorithms, for example, scheduling and event routing. These constitute the main algorithms used in sequential simulation too.

There is thus a need for efficient sequential and parallelised simulation algorithms that can be used to simulate models in the general-purpose DEVS formalism.

1.3 Alternatives

Current approaches to improving simulation performance have mainly focused either on modifications to formalisms, or on modifications to the synchronization protocols.

The use of different dedicated formalisms has the major disadvantage that they are often incompatible with existing generally used, formalisms. Each of these formalisms often only has a single (prototype) simulator supporting it, forcing modellers to port their models to the different formalism and simulator. A few simulators do aim at improving the sequential simulation algorithms, while remaining compatible with the general formalism.

A lot of research has gone into the development of efficient distributed synchronization protocols, such as time warp and its various enhancements. Most of the high-performance, general-purpose DEVS simulators only support parallel simulation using shared memory, which is insufficient to perform large distributed simulations.

1.4 Objectives and contributions

The main objective of this thesis is to enhance the performance of DEVS simulation through the addition and exploitation of user-provided domain-information, both in sequential and in parallel simulation. One such kind of domain-information, which will be the focus of this thesis, is the use of activity, both system-monitored and user-provided. Additionally, our prototype simulator will support a variety of features to make it widely applicable in a variety of domains.

Our objectives are thus three-fold:

1. Provide a wide variety of useful features in the simulator (e.g., real-time simulation, dynamic structures, multiple tracers, . . .);
2. Enhance sequential simulation algorithms using domain-information;
3. Enhance distributed simulation performance, mainly by load-balancing, using domain-information;

Such domain-information and features should not conflict with a general-purpose DEVS simulation, meaning that no new formalism will be introduced and all features and enhancements should be backward compatible with existing DEVS formalisms. There should also be no interference with ongoing standardization of the DEVS formalism. Activity-unaware simulators should be able to ignore all domain-information.

These features as well as performance enhancements are implemented and evaluated in this thesis by means of extensions to the PythonDEVS simulator. These extensions are implementation-independent and are equally well applicable in other simulators.

1.5 Technical remarks

PythonPDEVS is written in the Python[1] programming language. Development is focussed on Python 2 for compatibility reasons, though the source code is fully compliant with Python 3 as well.

The middleware is based on the MPI-2 standard[2], using the wrapper module *MPI4Py*. Note that PythonPDEVS requires the `THREADS_MULTIPLE` MPI option to be enabled. For optimal performance, it is recommended to use an MPI implementation that uses interrupt-based receives instead of one that uses busy looping. For these reasons, benchmarking and testing was done using the MPICH3[3] implementation.

The source code is fully documented to be usable by Sphinx[4] for automated documentation generation. The documentation also contains an example for most of the features.

The latest version of the PythonPDEVS simulator, the corresponding documentation and a copy of this thesis can be found online at `http://msdl.cs.mcgill.ca/people/yentl/`.

1.6 Outline

The remainder of this thesis will be as follows: This section will continue with a brief introduction to different variants of the DEVS formalism and to some distributed synchronization protocols. It is mostly based on the literature study performed in previous work[72, 73].

Section 2 discusses the design of the PythonPDEVS simulator that was built for this thesis. This design comprises UML diagrams, both class and sequence diagrams, together with a brief explanation of the most important classes. At the end, the modelling of (the most relevant subset of) the PythonPDEVS simulator in DEVS itself is presented.

Section 3 presents a non-exhaustive list of the most important features that were incorporated in the simulator. Trivial enhancements or changes, such as a progress bar, are not mentioned in this list. For a more exhaustive list, the *release notes* of the simulator are better suited. Each feature will have a brief explanation of how it is used, the rationale behind it and the implementation (with possible conflicts with other functionality or optimizations).

Starting from Section 4, the performance is taken into consideration and evaluated through the use of synthetic as well as realistic benchmarks. All performance enhancements that the PythonPDEVS simulator supports, with the exclusion of activity, will be presented and evaluated. Note that this section is a necessary precursor that is already related to activity: activity itself is only a means of automatically acquiring information about the future computational resource needs of a running simulation. This information is then used as replacement for the manual input that was required for the performance enhancements in Section 4.

[1] ***Python***: `https://www.python.org/`
[2] ***MPI-2***: `http://www.mpi-forum.org/docs/mpi-2.0/mpi-20-html/mpi2-report.html`
[3] ***MPICH3***: `http://www.mpich.org/`
[4] ***Sphinx***: `http://sphinx-doc.org/`

Section 5 continues with the addition of *activity*, making our simulator an *activity-aware* simulator as the title of this thesis suggests. Activity is only mentioned this late in the thesis, even though it is the focal point of this thesis. Clearly, it is important to consider activity in the context of an efficient and feature-rich simulator. The presence of all previously mentioned features will also have an impact on the opportunities for performance enhancements through activity. Both the common definitions of and our extensions to this definition will be discussed in detail. We will allow for both system-monitored activity and user-defined activity, which can be transparantly used in several performance enhancements introduced in Section 4.

Finally, Section 6 gives the conclusions of this thesis, relates back to our objectives and shows possible directions for future work.

A non-trivial model in PythonPDEVS, which is used as our final benchmark, is shown in Appendix A.

A city layout generator needed for one of our benchmarks is presented in Appendix B.

1.7 The DEVS formalism

In this section, a basic introduction to several DEVS formalism variants is given. Dozens such variants exist [5] , though only the three main formalisms that are of interest to the remainder of this thesis will be presented: Classic DEVS, Parallel DEVS and Dynamic Structure DEVS.

The details of these formalisms, most importantly the proofs of closure under coupling for all of the formalisms except Classic DEVS will be omitted. The interested reader is referred to the references for each formalism, where the respective proofs can be found.

1.7.1 Classic DEVS

Classic DEVS (CDEVS)[81, 77] is the formalism that spawned the many extensions that exist today. Hence, it is discussed here as an introduction to the formalism. Subsequent formalisms will be presented in how they differ from the Classic DEVS formalism.

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

Atomic DEVS

An *atomic DEVS* model is a structure:

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

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

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

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

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

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

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

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

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

The *output function* λ maps the sequential state set 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.

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

With ϕ defined as the *'null event'*.

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

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

[5]Others include DynDEVS[71], CellDEVS[80], Parallel CellDEVS[70], Fuzzy DEVS[81]

with e the elapsed time since the last transition.

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

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

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

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.

$\{M_i\}$ is the set of components containing the atomic [6] DEVS structure of all subcomponents referenced by elements in D.

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

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

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

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

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

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

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

Finally, the *tie-breaking function select* is used to resolve collisions between multiple components. Two or more components collide if they are scheduled to execute their internal transition function at the same time. The function must return exactly one component, which will execute its internal transition function.

$$select : 2^D \rightarrow D$$

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.

The coupled model

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

should be transformed to the atomic model

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

So we have to define all the variables of the atomic model as a function of the given coupled model. The input and output variables X and Y are easy, since they stay the same.

$$X = X_{self}$$
$$Y = Y_{self}$$

[6] This can be relaxed to both atomic and coupled models thanks to closure under coupling.

The state S now encompasses the parallel composition of the states of all the submodels, including their elapsed times:

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

with total states Q_i are 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 needs to be kept for every model separately, since the elapsed time of the new atomic model will update more frequently than each submodel's elapsed time.

The time advance function ta is constructed using these values and calculates the remaining time for each submodel. This construction causes multiple calls to the time advance function of component i. This does not form a problem in the case that this function is deterministic (as it should be!). So it is illegal to return a random variable that gets regenerated each time the time advance function is called. Note that pseudo-random number generators are deterministic.

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

The component that will have a transition is chosen with the *select* function. It 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 does not mean that nothing happened, but rather that the "output" is consumed internally.

The *internal transition function* is defined for each part of the flattened state separately:

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

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 due to the output it produces. 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:

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

Abstract simulator

An abstract simulator defines the operational semantics of the formalism. A direct implementation of this abstract simulator is often extremely inefficient as the focus is on clarity and conciseness. Implementations of real simulators are free to deviate from the abstract simulator, as long as the semantics are (proved to be) equivalent.

The abstract simulator for Classic DEVS Atomic models is shown in Algorithm 1.1. The algorithms for Coupled models are shown in Algorithm 1.2. Finally, the root coordinator (the main simulation loop) is defined in Algorithm 1.3. As the atomic models are the leaf of the hierarchy, they will perform the actual processing of the events, whereas the coupled models simply forward the events to the atomic models. These algorithms use the following variables:

- t_l: the simulation time at which the previous transition of this model took place.
- t_n: the simulation time at which the next internal transition of this model will take place.
- t: the simulation time enclosed in the event.
- e: the simulation time that has currently passed between the previous transition and the current simulation time.
- s: the current state of the atomic model.
- x: an incoming event.
- y: an outgoing event.
- $from$: indicates the source of the event.
- $parent$: indicates the parent model of the current model.
- $self$: indicates the model itself.

The different kinds of messages are:

- i: an initialization message, indicating that the model should be constructed for the first time.
- $*$: a transition message, indicating that the model has to perform an internal transition.
- $done$: a done message, indicating that the submodel has finished processing the previous message.

Note that the operational semantics for coupled models is equivalent to the translational semantics given by flattening a coupled model to an atomic one.

Algorithm 1.1 Classic DEVS atomic model abstract simulator

if receive $(i, from, t)$ message **then**
  $t_l \leftarrow t - e$
  $t_n \leftarrow t_l + ta(s)$
else if receive $(*, from, t)$ message **then**
  if $t = t_n$ **then**
    $y \leftarrow \lambda(s)$
    if $y \neq \phi$ **then**
      send $(y, self, t)$ to $parent$
    end if
    $s \leftarrow \delta_{int}(s)$
    $t_l \leftarrow t$
    $t_n \leftarrow t_l + ta(s)$
    send $(done, self, t_n)$ to parent
  end if
else if receive $(x, from, t)$ message **then**
  if $t_l \leq t \leq t_n$ **then**
    $e \leftarrow t - t_l$
    $s \leftarrow \delta_{ext}((s, e), x)$
    $t_l \leftarrow t$
    $t_n \leftarrow t_l + ta(s)$
    send $(done, self, t_n)$ to $parent$
  else
    error: bad synchronization
  end if
end if

Algorithm 1.2 Classic DEVS coupled model abstract simulator (aka coordinator)

if receive $(i, from, t)$ message **then**
  for all d in D **do**
    send (i,t) to d
    $active_children \leftarrow active_children \cup \{d\}$
  end for
  $t_l \leftarrow max\{t_{ld} | d \in D\}$
  $t_n \leftarrow min\{t_{nd} | d \in D\}$
else if receive $(*, from, t)$ message **then**
  if $t = t_n$ **then**
    $i* = select(\{M_i.t_n = t | i \in D\})$
    send $(*, self, t)$ to $i*$
    $active_children \leftarrow active_children \cup \{i^*\}$
  end if
else if receive $(y, from, t)$ message **then**
  for all $i \in I_{from} \setminus \{self\}$ **do**
    send $(Z_{from,i}(y), from, to)$ to i
    $active_children \leftarrow active_children \cup \{i\}$
  end for
  if $self \in I_{from}$ **then**
    send $(Z_{from,self}(y), self, t)$ to $parent$
  end if
else if receive $(x, from, t)$ message **then**
  if $t_l \leq t \leq t_n$ **then**
    for all $i \in I_{from}$ **do**
      send $(Z_{self,i}(x), self, t)$ to i
      $active_children \leftarrow active_children \cup \{i\}$
    end for
  end if
else if receive $(done, from, t)$ message **then**
  $active_children \leftarrow active_children \setminus \{from\}$
  if $active_children = \phi$ **then**
    $t_l \leftarrow t$
    $t_n \leftarrow min\{M_i.t_n | i \in D\}$
    send $(done, self, t_n)$ to $parent$
  end if
end if

Algorithm 1.3 Classic DEVS root coordinator

$t \leftarrow t_N of topmost coordinator$
while not $terminationCondition()$ **do**
  send $(*, main, t)$ to topmost coupled model top
  wait for $(done, top, t_N)$
  $t \leftarrow t_N$
end while

1.7.2 Parallel DEVS

Parallel DEVS (PDEVS) was presented in [16] as a revision of Classic DEVS, to provide a formalism that has better support for parallelism. At the time of writing, Parallel DEVS has replaced Classic DEVS as the default DEVS formalism in many simulators.

The corresponding abstract simulator can be found in [17]. The main change in this formalism is the introduction of *bags*, the *confluent transition function* and the removal of the *select* function.

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, as the order is non-deterministic.

$$\delta_{ext} : Q \times X^b \rightarrow S$$

- 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.

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

- λ is the *output function*, but now it outputs a bag instead of a single event. If a single event is desired, a bag with only one element should be sent. The null event is equal to an empty bag.

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

  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. This constraint is removed in Parallel DEVS, making multiple messages on a single port possible. To allow this, bags are needed.

Coupled DEVS

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

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

The disappearance of the *select* function means that colliding models should transition in parallel instead of sequential. This is why there is a possibility for parallelization, since the output function and transition functions can now happen in parallel by merging different bags. The atomic models themselves 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.

1.7.3 Dynamic Structure DEVS

Dynamic structures are found in many different kinds of models, though unsupported by Classic DEVS or Parallel DEVS. One of the most successful solutions to this problem is the Dynamic Structure DEVS (DSDEVS) formalism, proposed by [5, 6]. It basically uses a *network executive* in each coupled model that can receive events 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.

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 behavioral and not structural. Though an atomic model is now able to send events to the network executive to initiate the structural change, so it might be needed to update the output list.

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 inherited), namely that the *network executive* should not receive an event that collides with other events. This constraint is needed to guarantee that there is a deterministic network structure, since the order in which *external transition functions* are processed is non-deterministic, making the network structure non-deterministic. On the other hand, it is possible to do them at exactly the same time, as long as these events do not collide (as the network structure is deterministic in this case). This can be done by using intermediate states to separate 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:

$$\chi \notin D$$

1.8 Distributed simulation

Distribution of a DEVS simulation can be done using different synchronisation protocols. Two main categories exist: *conservative* and *optimistic*. As all of our own work is focused on optimistic synchronization, we will mainly focus on this category. Conservative synchronization is presented in order to give a well-founded explanation for our choice for optimistic algorithms.

The interested reader is referred to [22, 20, 73] for a more in-depth discussion and more tweaks to the protocols, or additional techniques in [78, 14, 13]. These techniques and a mention of their use on emerging platforms, such as web-based simulation, are also presented in [30].

1.8.1 Conservative

Conservative synchronization protocols will process events in timestamp order. If it cannot be guaranteed that an event is the next one, e.g. when there is a possibility that a remote event with a smaller timestamp may arrive, the simulation will block until the simulator is certain about the next event. Causality errors are avoided at the cost of blocking until it can be guaranteed that no causality errors can occur by processing the next event. Some more advanced aspects, like *bounded lag* and *conditional information* will not be discussed here.

Local causality constraint

A discrete event simulation is said to obey the *local causality constraint* if and only if it processes events in increasing time stamp order. With conservative synchronization, it is guaranteed that this local causality constraint is always respected. This means that all events are always processed in timestamp order.

Note that the local causality constraint alone is not sufficient to guarantee a simulation run that is exactly equal to the sequential simulation of the same model. The additional constraint is that simultaneous events (events with the same timestamp) are processed in the same order [7].

Even though adherence to these two constraints is sufficient to satisfy correct simulation, the prevention of causality errors is clearly not a necessity for a correct simulation. Multiple LPs in the same model may be independent of one another, thus a (slightly) different processing order does not necessarily cause causality errors.

Deadlock avoidance: null messages

Definition 1.8.1. If a logical process at simulation time T can only schedule new events with timestamp of at least $T+L$, then L is referred to as the *lookahead* for the logical process. This means that, if an LP is at simulation time T, all connected LPs can be sure that no event will be received from that LP before simulation time $T+L$.

Using information about the *lookahead* of an LP, a logical solution to the deadlock problem is the Chandy/Misra/Bryant algorithm, which introduces *null messages*. Null messages are messages without any actual simulation content and only function as a notification from the sender to the receiver that a specific simulation time has been reached. Such messages will notify other logical processes about the local simulation time and subsequently allow the algorithm to progress. After the processing of an event, this raises the need to send a null message to all connected LPs. Since these messages are used to make a guarantee, it is possible to send a message with $T+L$, with L being the lookahead, instead of T and thus the other nodes will be able to advance even further into the future.

When the null message itself is processed (i.e. when it is the first element of all queues), the LP will only have to update its clock and resend null messages with a higher timestamp. This provides a solution for deadlocks in most, but not all, cases. One of the unsolved problems is a deadlock when the lookahead is equal to 0, as it will now keep sending exactly the same null messages without any advancement of the simulation time.

Another performance problem might occur if the lookahead is very small compared to the actual time increase required to allow simulation. In such a case, the LPs will have to keep sending null messages until the deadlock is solved. Actually, this case becomes very much like normal *discrete time* simulation as we sometimes have to progress with a fixed timestep if the difference becomes too big for the lookahead. To make matters worse, these messages must traverse the network, possibly causing high latency until the next simulation event is actually processed.

Often it is redundant to send null messages, for example when there are still unprocessed messages in the influencee's input queue. In such cases it could be advantageous to use a *demand-driven* approach for sending such null messages. And while this can significantly reduce the number of messages on the network and the respective overhead, it causes higher delays when a null message is requested due to the round trip delay.

The performance of the null message algorithm is thus highly dependent on the size of the lookahead. Consequently, this algorithm is only recommended in cases where the lookahead is big enough. Conservative synchronization is not that general as lookahead is a characteristic of the model. Several models are even impossible to simulate using this algorithm, specifically models with a lookahead loop that is equal to zero. It is also possible for lookahead to fluctuate throughout the simulation. However, it is not allowed to decrease instantaneously due to previous promises to the influencees.

Deadlock detection and recovery

Another solution to the deadlock problem is to allow deadlocks to happen, but provide a mechanism to detect them and recover from this situation. The main advantage is that it doesn't require null messages to be sent and thus lowers the message overhead (both sending, receiving and processing). If no problem occurs, there is also no need to perform such synchronization, which allows for a low-overhead simulation. A disadvantage is that such detection can only happen as soon as the deadlock has already occurred and the simulation is already halted. Even before a complete deadlock has formed, several LPs could be blocked, which is undetectable with this algorithm. In the worst case, it is possible that the simulation comes to a slow halt, where most of the LPs are blocked most of the time.

Detection To be able to detect whether or not deadlock has occurred, we construct a tree with the controller as root node. Children in the tree will be models that have received messages from their parents and thus are able to process new events. As soon as the controller declares that messages are safe to process, the tree will be constructed while messages - and subsequently computation - spread throughout the network. As soon as an LP becomes deadlocked, it will remove itself from the tree and signal its parent about this removal. If the parent becomes a leaf in the tree and has no computations of its own, it will also be deadlocked and thus remove itself from the tree. The complete simulation is than deadlocked as soon as the controller, which was the root node, becomes a leaf node.

[7] Some other problems still occur, though they are mostly at a lower abstraction level, e.g. floating point operations with different precisions on different machines

Recovery As soon as a deadlock is detected, it is still necessary to recover from them, which is done by having the controller send out messages which are safe to progress. The actual problem that remains is determining which messages are safe to process. A straightforward solution is to select the message with the lowest timestamp in the complete simulation and mark it as safe. More problems remain, as it might be the case that some messages are still in transit between different nodes and are therefore not taken into account when searching for the message with the lowest timestamp. This problem of so called *transient messages* is further explored in [22]. In the following algorithms, the assumption is made that no transient messages exist.

To determine this lowest timestamped event, communication between all nodes is necessary, as we are trying to establish a global state. In order to prevent a lot of communication overhead and the associated bottleneck at the controller, an algorithm to compute this safe message can use a spanning tree, where the first message is passed over the edges. At the end, the actual minimum will have arrived at the root node, which broadcasts this value, possibly using the same spanning tree again.

While this algorithm is relatively easy, it has the major drawback that it only allows one message to become safe and thus there is the possibly of recreating a deadlock after that message is processed. In this case, lookahead information could be used in order to declare multiple messages as safe and thus allow more messages to be processed, increasing concurrency.

Conclusion

The common goal of conservative synchronization protocols is to ensure that the *local causality constraint* is respected and thus guarantee that messages are processed in timestamp order. If this is the case, in combination with repeatable input message ordering, it is guaranteed that the same results will be produced as in a sequential execution. Problems might arise due to deadlocks, which has several solutions depending on the model that has to be simulated. To allow for more concurrency, *lookahead* can be exploited which is again dependent on the model that will be simulated.

Performance is very much related to the kind of model: high performance can be achieved if the model has a high lookahead, but a very small lookahead causes nearly sequential execution.

These limitations cause us to believe that conservative synchronization is not recommended for general models, as there are several distinct algorithms, each of them optimized for a specific type of models. The effect of lookahead is too big for the user to ignore, forcing the user to take the algorithms into consideration while modelling[20].

In our implementation, we want the algorithm to be as transparent as possible for the end user. Our goal is to have completely transparent switching between distributed and sequential simulation. Because a lookahead value cannot be determined automatically, conservative synchronization is unsuited for our goals.

1.8.2 Optimistic

In contrast to *conservative* synchronization, where only safe events are executed, *optimistic* synchronization will process as many events as possible in the assumption that they are safe. While this completely avoids the overhead associated with safeness checking, it is possible for violations to happen. Such violations must then be discovered and rectified, which constitutes the overhead associated to optimistic synchronization. *Time warp*[31] is one of the best known optimistic synchronization protocols and is presented in the remainder of this section. Other methods, like *lightweight time warp*[40, 41] are not elaborated on, as they are not implemented in PythonPDEVS.

Synchronization

An important aspect about *time warp* is that it does not use any additional communication between different LPs at all. It furthermore doesn't require the other LPs to send messages in timestamp order, nor does it require the network to deliver these messages in time stamp order. Support for models with zero lookahead is also relatively simple, while this is known to be problematic in several conservative synchronization protocols.

While no additional communication is really mandatory, some global communication in the form of a *Global Virtual Time* is highly recommended in order to have some kind of knowledge about all other LPs. The message overhead of this synchronization is relatively small and should not incur a huge slowdown or cause the network to become a bottleneck. Additionally, this synchronization only happens sporadically and its frequency can be a configuration parameter.

Stragglers

Definition 1.8.2. A *straggler message* is a message with timestamp T_m that is received at simulation time T_n, with $T_m < T_n$. It is thus a *late* message.

Because no guarantees are made about the safety of messages before they are processed, there is always the possibility for messages to arrive at a time before a previously processed time. This could mean that the simulation at a certain LP is already at simulation time t_2, when suddenly a message with timestamp t_1 is received (with $t_1 < t_2$). This was impossible with conservative synchronization, as the LP would not progress to time t_2 if it knew that there is the possibility for a message with an earlier

timestamp to arrive.

Should no straggler messages ever happen, optimistic synchronization is (in theory) a perfect technique because each LP can just simulate at its maximum pace without taking other LPs into account. Sadly, reality is different and *straggler messages* appear relatively frequently, depending on the type of model and the allocation. Whereas conservative algorithms were highly influenced by the amount of lookahead, optimistic algorithms are highly influenced by the number of stragglers that occur.

Rollback

Due to the possibility of straggler messages, the simulation algorithm might have to roll back to a previous time, right before the *local causality constraint* was broken. If a straggler message with timestamp t is received, the *complete LP* will have to roll back to a time right before simulation time t. This rollback is effectively an *undo* operation on the LP, which must then:

1. *Reset* the state variables of all its models to the state at time t.
2. *Unsend* all messages that were sent to other LPs after time t.
3. *Reprocess* all incoming messages that were already processed after time t.

Clearly, a lot of work that has already happened has to be repeated if these computations are rolled back. All computations that were done in the meantime have to be discarded as they used old (and probably invalid) information. This isn't completely wasteful, as conservative synchronization would have been blocking to make sure that the event is safe to process, whereas optimistic synchronization would have been progressing, hoping that no straggler would arrive. The disadvantage is that this rollback operation itself will also have a certain cost. As a rolled back LP should also unsend all of its messages that were send in the period that is being rolled back, these messages have the potential to cause a rollback at different nodes.

A rollback that was caused by a straggler message is called a *primary rollback*. A rollback caused by another rollback is called a *secondary rollback*.

State saving

While rolling back doesn't seem to be that much of a problem, the old states need to be known, requiring continuous saving of every state. All events that were received must also be stored after they were processed in order to allow them to be reused in the case that a rollback is required. Even all messages that get sent by the LP need to be saved somewhere, as we have to know where these messages must be invalidated in the case of rollback. Clearly, all this state saving does require a lot of additional overhead due to the many memory operations that are necessary.

Anti-messages

In order to *unsend* messages, the old messages that have to be unsend need to be known. These messages can be altered to signal that they are actually *anti-messages* and thus that they cancel out the corresponding message. Several situations are possible when an anti-message is received:

- **The corresponding message is not yet received**
  In this case, the anti-message effectively hopped over the actual message. The anti-message should be queued and as soon as the corresponding message arrives, both of them should be deleted.
- **The corresponding message is not yet processed**
  If the corresponding message is still in the input queue, the anti-message will cause the removal of this message from the queue, in which both of them get deleted.
- **The corresponding message is already processed**
  This is the worst case, as now a rollback to the time before processing this message is required. After the rollback is completed, this reduces to the situation where the message is not yet processed.

To save some memory, the anti-message only has to contain the timestamp, destination and message ID instead of the complete message.

Fossil collection

Time warp requires a lot of memory to accomodate for all messages and state saving, which should be removed as soon as they are no longer necessary. Fossil collection does exactly that: search for messages and states that will no longer be used and remove them.

This search becomes trivial as soon as a lower bound for the simulation time at all simulation kernels is found. Such a lower bound is called the *Global Virtual Time (GVT)*. All states and messages before the GVT can be removed, as it is impossible to receive messages from before the GVT. Clearly, if no messages are received from before the GVT, there will be no rollbacks to a time before the GVT.

Algorithm 1.4 Mattern's algorithm when sending a message from P_i to P_j

send the actual message to P_j
if color = white **then**
  $V[j] \leftarrow V[j]+1$
else
  $t_{min} \leftarrow min(t_{min}, timestamp)$
end if

Algorithm 1.5 Mattern's algorithm when receiving a message at P_i

if msg_color = white **then**
  $V[i] \leftarrow V[i]-1$
end if
process the message and use the enclosed timestamp

Global Virtual Time

Definition 1.8.3. *Global Virtual Time (GVT)* at wall-clock time T during the execution of a Time Warp simulation is defined as the minimum time stamp among all unprocessed and partially processed messages and anti-messages in the system at wall-clock time T.

Computing the GVT is not as simple as it might sound. Many factors need to be taken into account, with transient messages being the most important problem to take into account. Additionally, a problem might arise because LPs don't send their minimal times at exactly the same time, making it possible for a message to slip between the cracks and be undetected.

An algorithm that allows for asynchronous message passing and that does not block is called *Mattern's algorithm* and is discussed next. Other algorithms include Samadi's algorithm[59] and Time Quantum GVT[15].

Mattern's algorithm

Mattern's algorithm was defined in [42] and proposes a general way of finding the GVT (or at least an approximation). It supports many different network structures, though only an example of a ring algorithm is given. Furthermore, it also offers several possible optimizations, depending on the amount of information that is available to the LPs (e.g. knowing which LP should still receive a transient message).

The algorithm is presented in four parts:

- Algorithm 1.4 is invoked for every message that is send to another node;
- Algorithm 1.5 is invoked for every message that is received from another node;
- Algorithm 1.6 is the part of the algorithm that runs at every node as soon as a control message is received;
- Algorithm 1.7 is called at the start of the simulation by the controller and will start the algorithm.

The algorithm is based on two colors, with every LP having either one of them. All LPs start in color *white*, where they record the number of messages they send and receive. As soon as a control message is received, their color switches to *red*. All *white* messages (thus messages *sent* when the LP had color *white*) will be taken into account in the minimal time calculation and in the counter. As soon as the LP became *red*, it will lower its minimal time as soon as an earlier message is received.

As soon as all *white* messages are received, no *transient* message is present that can lower the GVT. This is an important consideration, since this is an asynchronous algorithm: simulation simply continues throughout the complete algorithm and thus transient messages will always be present, but we only avoid these messages from having an influence. This contrasts with blocking GVT algorithms, where there are no *transient* messages at all when the GVT is determined.

If after the first round there are no *white* messages in transit (meaning that there were no *transient* messages to start with) the algorithm is already finished and actually reduces to a simple ring algorithm. In the case where some *white* messages are still in transit, a second round is necessary to wait for these *transient* messages. A *red* message has no meaning in the algorithm because it doesn't matter whether or not they are transient. As soon as an LP is *red*, it should still take into account the actual sending of the message, as it is possible that an earlier message is sent in case a rollback happened.

Conclusion

Time warp offers a general solution for the parallellization of models, at the cost of possible violations. These violations have to be resolved, which causes additional overhead. Time warp's main problem is the lack of upper bound to the amount of required memory, which requires additional algorithms and techniques to take care of this problem.

Algorithm 1.6 Mattern's algorithm when receiving a control message $< m_{clock}, m_{send}, count >$ in P_i

if color = white **then**
  $t_{min} \leftarrow \infty$
  $color \leftarrow red$
end if
wait until $V[i] + count[i] \leq 0$
send $< min(m_{clock}, T), min(m_{send}, t_{min}), V + count >$ to $P_{(imodn)+1}$
$V \leftarrow 0$

Algorithm 1.7 Mattern's algorithm when receiving a control message $< m_{clock}, m_{send}, count >$ in P_{init}

wait until $V[init] + count[init] \leq 0$
if $count = 0$ **then**
  $GVT_approx \leftarrow min(m_{clock}, m_{send})$
else
  send $< T, min(m_{send}, t_{min}), V + count >$ to $P_{initmodn+1}$
  $V \leftarrow 0$
end if

While it seems that time warp is unaffected by the properties of the model that is being simulated, it is highly dependent on the distribution of the model as this is responsible for the rate at which they progress and possibly diverge. For increased performance, models at different LPs should pass as few messages as possible due to each message being a potential straggler.

1.8.3 Conclusion

Since we want to provide transparent distribution and parallelisation to our user, the conservative approach is unsuited for our implementation. An optimistic algorithm can handle every kind of model and the user is thus not required to provide any information. Should the user have more knowledge, time warp can profit from this by creating an ideal distribution of the model.

One could argue that conservative protocols are still applicable if the deadlock detection and recovery method is used and the lookahead is assumed to be 0 unless overridden by the user. This is partially true, though it still is no competitor to optimistic algorithms for several reasons:

1. Deadlock detection and recovery is still highly dependent on the lookahead. Even though a lookahead of 0 is not a problem concerning deadlocks, it will have a severe impact on the performance. If the default lookahead of 0 were to be used, simulation could frequently stall as only a single timestamp will be marked safe.
2. The requirements for efficient optimistic synchronization are much closer to the expectations of the user. It is intuitively clear that models that are mostly independent and have an equal distribution between them, will parallelize well. On the other hand, it is not that clear in which situations deadlocks can occur.
3. An optimistic synchronization protocol will be able to handle relocations well, whereas this will have severe consequences in conservative protocols. The lookahead (should it be defined) would vary drastically with different allocations, putting even more of a burden on the user.

2
Design

This chapter discusses the design of the PythonPDEVS kernel. A class diagram of the simulation kernel is presented, followed by a discussion of sequence diagrams of the core algorithms. Our GVT algorithm is based on Mattern's algorithm, but slight modifications were done. These modifications and their rationale are mentioned in this section too. To conclude, a subset of PythonPDEVS is modelled in DEVS and results are evaluated.

2.1 UML Class Diagram

A compact class diagram of the PythonPDEVS simulation kernel is shown in Figure 2.1. Only the most important methods, attributes and associations are shown to keep it compact. Additionally, some very similar classes were collapsed into a single class. This is the case for the different schedulers (`SchedulerML`, `SchedulerSL`, `SchedulerAH`, `SchedulerDH`, `SchedulerHS` and `SchedulerNA` are simply represented as `Scheduler*`) and for the different tracers (`TracerVerbose`, `TracerVCD`, `TracerXML` and `TracerCell` are represented as `Tracer*`).

The remainder of this section will briefly explain all classes. A more in-depth explanation can be found in the *internal documentation* section of the documentation.

`Allocator`

Nearly identical to the `Relocator`, though it is only executed in the beginning of the simulation and takes some more parameters, such as the load on links between models.

`AtomicDEVS`

The superclass for every atomic DEVS model, it implements basic control over such models such as performing rollbacks.

`BaseDEVS`

This is the base class of all user-defined DEVS components. It provides basic support for fetching the model name, creating ports, deleting ports, ... Other responsibilities include the processing of Dynamic Structure DEVS constructs.

`BaseSimulator`

The `BaseSimulator` is the actual simulation kernel and will handle all simulation control. It is a subclass of the `Solver` entity, which does the actual DEVS simulation.

These kernels govern the simulation and implement the time warp protocol, GVT algorithm, ...

`Controller`

Some parts of the `BaseSimulator` are only required for a specific node and can thus be extracted to a different *subclass*. The `Controller` is responsible for termination detection, initializing the GVT algorithm, initializing relocations, performing tracing, ...

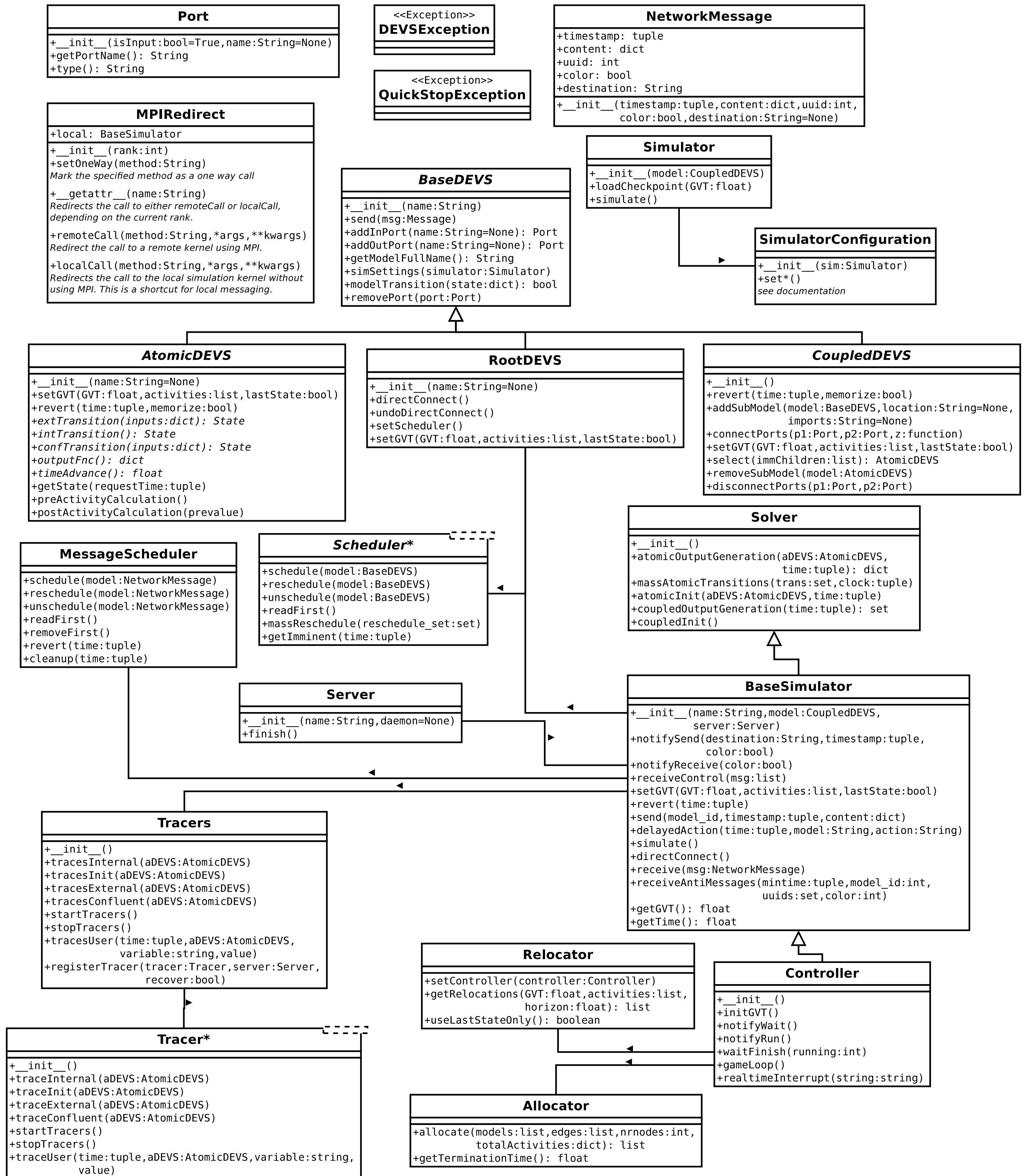

Figure 2.1: Class diagram of the PythonPDEVS simulation kernel

CoupledDEVS

The superclass for every coupled DEVS model, it implements basic control over such models such as performing rollbacks.

DEVSException

An exception thrown when something is very wrong in the simulation. Depending on the error message enclosed, this can either signal a violation of the DEVS formalism in the models, or a bug in the simulation kernel. Simulation will immediately halt as soon as such an exception is thrown.

MessageScheduler

A `MessageScheduler` is used to schedule all incoming network messages, or unschedule them if a corresponding anti-message is received. The simulation loop will poll it for all received messages up to the current simulation time.

MPIRedirect

The `MPIRedirect` class will provide a layer above MPI to make it resemble Remote Method Invocation (RMI). This makes network calls relatively transparent for the rest of the simulator. Both synchronous (blocking) and asynchronous (non-blocking) calls are supported.

Due to this somewhat unnatural use of MPI, we pay a small performance penalty. However, the overhead will be relatively small and eases the development of the rest of the simulation kernel. Furthermore, actual usage of MPI is completely abstracted, making e.g. RMI possible without any modifications to the simulator itself.

The exchanged messages are in the format of a pickled string, due to a lack of standardization for the DEVS formalism. On the other hand, this allows us to focus more on performance instead of interoperability. Interoperability was the main focus in [61, 58, 35, 3]. Moving to an interoperable middleware would only require changes to the `MPIRedirect` and `Server` classes.

NetworkMessage

A data class for the messages transfered over the network.

Port

A class to represent a simple port (either input or output) of the model.

QuickStopException

The `QuickStopException` is a special kind of exception that can be thrown within the main simulation loop to abruptly interrupt the transitioning phase. When for example a rollback is requested, the transition function would normally finish first, which is unnecessary as a rollback will undo the results right away. After such a `QuickStopException`, the kernel will be in an inconsistent state, which is always corrected again due to the rollback that will be performed afterwards.

Relocator

A user-configurable class that takes activity measures and the current distribution in order to create a new distribution. This new distribution is then used to perform migrations.

RootDEVS

During the actual simulation, all `CoupledDEVS` models will be rewritten and only a single one remains: the `RootDEVS` model. All actual simulation methods are thus enclosed in the `RootDEVS` class instead of `CoupledDEVS`.

Scheduler*

These different schedulers all have a similar interface: `schedule`, `unschedule`, `massReschedule`, ... This is a modular component, so it can simply be replaced by a component that the user would (or at least could) write. Some of them are enclosed in the PythonPDEVS distribution.

Server

This class is responsible for all network communication, as it will be called to get proxies to the different nodes. Furthermore, it will also poll the network for incoming messages. As soon as a message is received, it will be put on its own thread to be processed.

A distinction is made between *control* and *data* messages:

- **Control** messages are used to direct the simulation itself. These messages are generated by the GVT algorithm, setup of the simulation or tracers. They are used infrequently and are handled by creating a new thread for each of them.
- **Data** messages are the actual DEVS events that need to be transferred. They are processed by a thread pool, as there is no chance for deadlocks.

`Simulator`

The only class that the user has to use is the `Simulator` class, as it provides an interface to the complete simulation. Configuration options can be set on this class for simplicity, though they are actually redirected to the `SimulatorConfiguration` class.

`SimulatorConfiguration`

All configuration is done on this class instead of the `Simulator`. It performs all checking of the configuration that is being done and throws `Exceptions` should the requested configuration be illegal. It is mainly split up as the `Simulator` class would otherwise become much too large and its interface would not be as clean.

`Solver`

This class contains all code related to actual DEVS solving and (local) message routing.

`Tracer*`

The actual tracers that will perform all necessary tracing. Its methods are invoked by the `Tracers` class and have access to the controller object in order to sequentialize all tracing.

`Tracers`

All user-configurabe `Tracer` classes are registered with the `Tracers` class, which handles all broadcasting of messages, registration, deregistration and checkpoint recovery.

2.2 UML Sequence Diagram

The 3 most important algorithms are presented here in the form of a UML Sequence Diagram. These diagrams are again a (minor) simplification of the actual situation that is represented, to be able to keep the diagrams concise.

2.2.1 Local simulation

Local simulation is the basic case in which only a single core is used. The sequence diagram is shown in Figure 2.2.

In local simulation, the `Simulator` object receives the `simulate()` call after it was initialized. It will redirect this call to the simulation kernel (`BaseSimulator`), which will enter the simulation loop. Subsequently, calls will be made to the `Solver` and `Scheduler` components.

This algorithm is completely different from the abstract simulator for performance reasons: The abstract simulator uses generic messages, containing a type variable which have to be checked at every model. In our implementation, the **, X, Y, init, done* messages are replaced by function calls such as `atomicInit`, `coupledOutputGeneration`, . . . Additionally, the code becomes more readable for those unfamiliar to the DEVS abstract simulator.

2.2.2 Distributed simulation

Distributed simulation is, due to our choice for *Time Warp*, very similar to local simulation. The major difference lies in the checking for new external (possibly straggler) messages, which have to be injected in the simulation. The actual remote messaging part will be the main focus in Figure 2.3.

As soon as a call to the `send` method of the `BaseSimulator` is made, it will prepare the message for network transfer. When the message is prepared, it is used as a parameter for the `send` method of the `Server`. A call will be made to the own `Server`, which is responsible for the actual (asynchronous) transmission. The remote `Server` receives the message and notifies the `BaseSimulator`, which queues the message for processing.

In the simulation loop, the queue is polled and all queued messages are scheduled in the `MessageScheduler`. The `MessageScheduler` will then return all of its scheduled messages at the required time, which the `BaseSimulator` injects in the simulation.

2.2.3 Relocation

Relocation is a joint effort of 3 simulation kernels: the controller initiates the relocation, the source sends the model and the destination receives the model. The sequence diagram is shown in Figure 2.4.

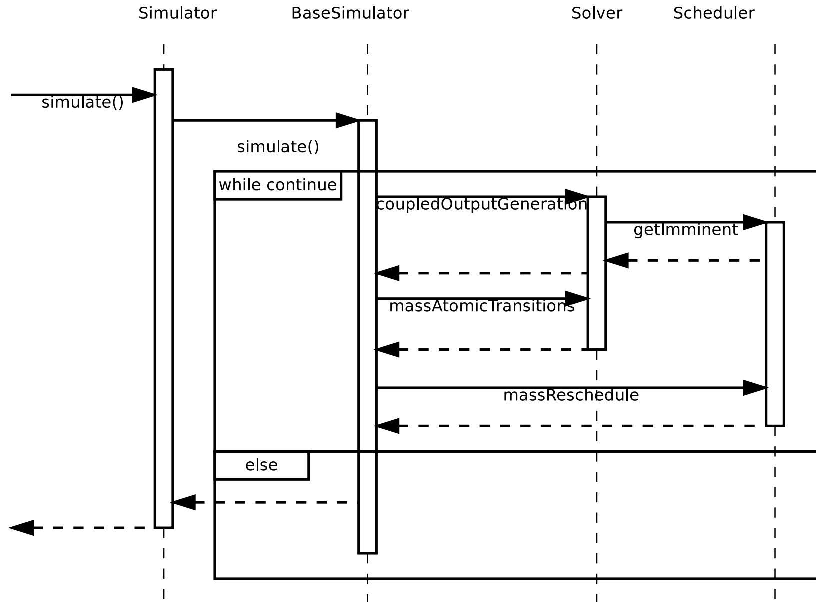

Figure 2.2: Sequence diagram for the local simulation loop

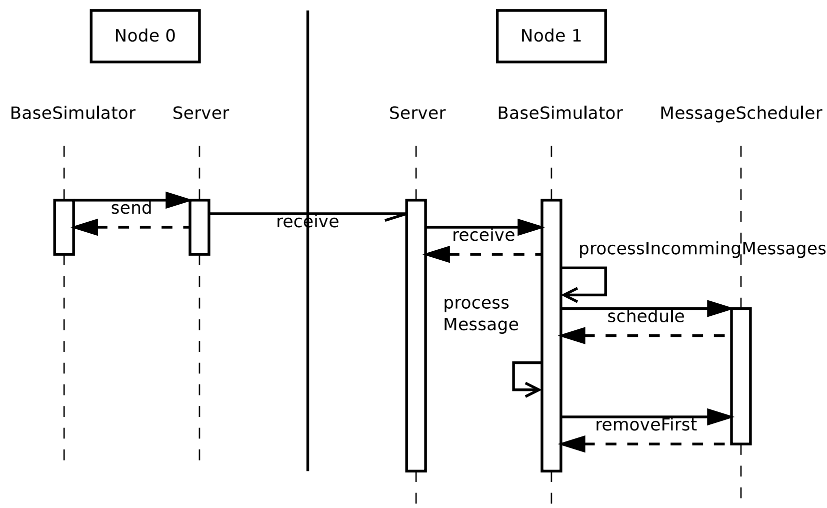

Figure 2.3: Sequence diagram for an inter-node message

The controller will signal both the source and the destination kernels that they should block further simulation as soon as possible, roll back to the GVT (which has just been updated) and notify the controller as soon as this is done. A rollback to the GVT is required to prevent the sending of the (possibly immense) state vector and a list of all sent messages. When rolling back to the GVT, the state vector will contain exactly 1 state because all states before the GVT were removed with fossil collection and all states after the GVT are removed by the rollback. The same holds for the list containing all sent messages. All such calls are made asynchronous, as otherwise relocation would be a completely sequential operation.

As soon as both the source and destination are locked, they notify the controller that relocation can start. At this time, only the controller knows the source and destination of the relocation. The source is notified by the controller that it should transfer some of its models to another node.

First of all, all nodes except the source and destination are notified of this change, which is necessary to keep all routing information correct. The source kernel then sends all messages that are still buffered for the soon-to-be-relocated model to the destination. Finally, all states are transferred to the destination and they are cleared at the source for space saving.

As soon as the source notifies the controller that it has done its job, the controller will decide whether or not to unlock the nodes. Should multiple relocations be queued, the controller will only unlock those that are no longer required and will start up the next relocation, depending on which kernels are locked at the moment.

Figure 2.4: Sequence diagram for a relocation from node 1 to node 2, with node 0 as controller

Figure 2.5: Naive modification of Mattern's algorithm in all circumstances

2.3 Modifications to Mattern's algorithm

In the original version of Mattern's algorithm, it is not possible to restart the algorithm in order to get a second GVT. As we want to determine the GVT at regular intervals, restarting the algorithm is an important requirement. The modifications are presented in 2 steps: first a naive implementation is presented, followed by taking care of some border cases that arise.

2.3.1 Naive implementation

The main cause of the problem is that simulation always stops in the *red* color, without ever going back to *white*. This makes it impossible to restart the algorithm, as the color must be *white* at the start of the algorithm. A very naive solution would be to start of with a *'make white'* phase, or turn the messages *white* again in the second pass, but this has the obvious problem that not all messages will be logged in the second pass.

One solution to this problem is to skip the possibility for early termination and *always* perform the second phase, in which the color is also turned *white* again. This makes sure that messages are always accounted for, without major modifications to the algorithm. This is the algorithm that was implemented in previous versions of PythonPDEVS. However, modelling and subsequent simulation of PythonPDEVS (Section 2.6) revealed a slight flaw, which is addressed next.

2.3.2 Correcting special cases

Figure 2.5 shows all possible kinds of messages. The basic algorithm accounts for all types of messages, except for messages of type 5 [1]. With messages of type 5, the original algorithm would simply not log the message as it was after the execution of the algorithm. In the naive modification of the algorithm, the message will be accounted for as an ordinary *white* message.

Now the problem becomes obvious: it is possible for multiple *white* messages to be in transit, of which only some are sent before the node turned red. An example of a problematic situation is shown in Figure 2.6. The message being sent from LP 1 to LP 2 is incorrectly flagged as a *white* message, which makes LP 2 think that it has received all white messages. LP 2 progresses and takes into account the time of the message from LP 1 to LP 2, instead of the actual message for which it was waiting (the message from LP 3 to LP 2). As soon as the GVT algorithm is finished, the actual transient message is received. In the worst case, this message was sent from a time before the GVT and thus simulation is faulty.

In order to resolve this problem, we shouldn't allow *new white* messages to be detected as *old white* messages. A simple solution

[1] type 6 is handled by reinitializing the algorithm as soon as the color becomes *white* again

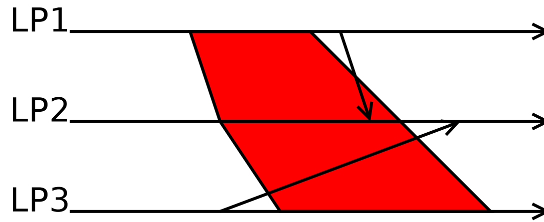

Figure 2.6: Problem in the naive modification of Mattern's algorithm

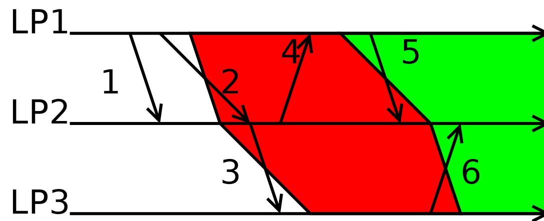

Figure 2.7: Mattern's algorithm (modified) in all circumstances

to this problem is the introduction of 2 additional colors: *white 2* and *red 2*. In addition to a new set of colors, the *V* vector needs to be duplicated too: one for the *white* messages and one for the *white 2* messages. Both of them cannot use the same vector, as the same problem would arise again.

When a message is received, its color must be checked for either *white* or *white 2* and the corresponding vector needs to be changed. The same is true when sending a message, as the correct vector has to be selected. The order of the colors then becomes: *white, red, white 2, red 2, white, red, white 2, red 2, ...* .

Only 2 additional colors are sufficient, as we only need a distinction between old and new colors. The amount of additional colors is thus independent of e.g. the number of simulation nodes. However, the color can no longer be transferred as a boolean (2 possibilities), but needs to be transferred as an integer. This adds a negligible overhead to the GVT algorithm. While technically also possible with a total of 3 colors, as there is no need for *red 2*, the algorithm becomes more symmetric with 4 colors.

This solution is shown in Figure 2.7, where *white 2* is represented as green. Clearly, messages of type 5 will no longer be misinterpreted as *white* messages, as they now have the *white 2* color and are accounted for as such.

Another problem is that only *white* messages are being waited on. While *red* messages are being accounted for in the *Tmin* variable, this variable is reset when entering the second GVT computation. In the original algorithm, it is unnecessary to wait for these *red* messages, as they are allowed to be transient by design. This problem is shown in Figure 2.8: messages of type 1 are impossible as the LP will only turn *white* as soon as it has received all *white* messages. Messages of type 2 on the other hand, are still possible because there is no such blocking behavior for the *red* messages.

To solve this problem, we also have to wait for all *red* messages before coloring a node *red* again. This required some more invasive changes to the algorithm, as now *red* messages are also stored in a vector. Effectively two vectors will be passed in each phase: a vector that is being waited on, and a vector that is being accumulated for a first run. As soon as the controller is reached, the next phase of the algorithm begins. At this point, the waiting vector should be empty and be replaced by the accumulating vector. Finally, a GVT is found and the accumulated vector has to be saved somewhere and should be used to initialize the next run of the algorithm.

It can now be seen that all possible combinations are safe:

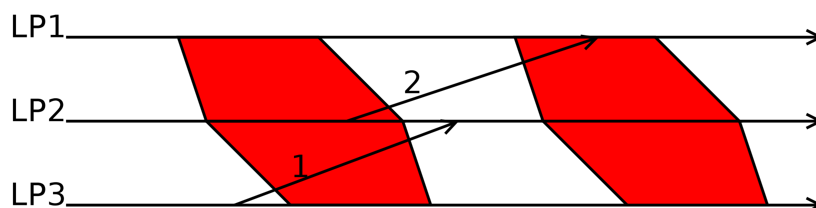

Figure 2.8: Mattern's algorithm with transient red messages

- Messages of type 5 are now reported as *white2* (*white1*) messages instead of *white1* (*white2*) messages and will not cause any confusion in the vector.
- Messages being sent from a *red* region will first have to be received before the color can become *red* again.

2.3.3 Complete version

The complete version of the modified algorithm, with all border cases taken care of, is shown in algorithms 2.1, 2.2, 2.3 and 2.4.

Algorithm 2.1 Modified Mattern's algorithm when sending a message from P_i to P_j

send the actual message to P_j
$V[color][j] \leftarrow V[color][j]+1$
if color = red1 **or** color = red2 **then**
  $t_{min} \leftarrow min(t_{min}, timestamp)$
end if

Algorithm 2.2 Modified Mattern's algorithm when receiving a message at P_i

$V[color][i] \leftarrow V[color][i]-1$
process the message and use the enclosed timestamp

Algorithm 2.3 Modified Mattern's algorithm when receiving message $< m_{clock}, m_{send}, wait_vector, accumulate_vector >$ in P_i

if color = white1 **then**
  $prevcolor \leftarrow red2$
  $nextcolor \leftarrow red1$
else if color = red1 **then**
  $prevcolor \leftarrow white1$
  $nextcolor \leftarrow white2$
else if color = white2 **then**
  $prevcolor \leftarrow red1$
  $nextcolor \leftarrow red2$
else if color = red2 **then**
  $prevcolor \leftarrow white2$
  $nextcolor \leftarrow white1$
end if
wait until $V[prevcolor][i] + wait_vector[i] \leq 0$
$wait_vector = wait_vector + V[prevcolor]$
$accumulate_vector = accumulate_vector + V[color]$
$V[wait_vector] \leftarrow 0$
$V[accumulate_vector] \leftarrow 0$
send $< min(m_{clock}, T), min(m_{send}, t_{min}), wait_vector, accumulate_vector >$ to $P_{(i\%n)+1}$
$t_{min} \leftarrow \infty$
$color \leftarrow nextcolor$

2.4 Model distribution

With Time Warp, normally every atomic model would constitute an LP and take part in the time warp implemenation. Therefore, every atomic model would have its own kernel and take care of e.g. saving its output messages.

In PythonPDEVS we take a different approach, where we group all atomic models into a single LP. This approach is already in use by e.g. *DEVS-Ada/TW*, but was criticized in [33, 32] as it causes rollbacks on all atomic models at that node instead of only a single atomic model. Instead of considering every atomic model as an LP, we create only a single LP at every node. This LP will then contain all of the atomic models that are present at that single node and will be responsible for the maintenance associated with time warp.

Our approach is shown in Figure 2.9. Normally, this model would consist of 4 different LPs: *A*, *B*, *C* and *D*. As there are very active connections between *A* and *B*, they are placed at the same core to prevent long network latencies. However, both *A* and *B* still have to save the event when sending and receiving it and possibly roll back if the message was a straggler. With our approach, only 2 LPs exist: *AB* and *CD*, as models *A* and *B* are merged into a single LP. The very active connection between *A*

Algorithm 2.4 Modified Mattern's algorithm when receiving message $< m_{clock}, m_{send}, wait_vector, accumulate_vector >$ in P_{init}

if color = white1 **then**
  $prevcolor \leftarrow red2$
  $nextcolor \leftarrow red1$
else if color = red1 **then**
  $prevcolor \leftarrow white1$
  $nextcolor \leftarrow white2$
else if color = white2 **then**
  $prevcolor \leftarrow red1$
  $nextcolor \leftarrow red2$
else if color = red2 **then**
  $prevcolor \leftarrow white2$
  $nextcolor \leftarrow white1$
end if
$wait_vector \leftarrow accumulate_vector$
$accumulate_vector \leftarrow 0$
if color = white1 **or** color = white2 **then**
  $GVT_approx \leftarrow min(m_{clock}, m_{send})$
  save the *wait_vector* for the next run
else if color = red1 **or** color = red2 **then**
  wait until $V[prevcolor][i] + wait_vector[i] \leq 0$
  $wait_vector = wait_vector + V[prevcolor]$
  $accumulate_vector = accumulate_vector + V[color]$
  $V[wait_vector] \leftarrow 0$
  $V[accumulate_vector] \leftarrow 0$
  send $< min(m_{clock}, T), min(m_{send}, t_{min}), wait_vector, accumulate_vector >$ to $P_{(i\%n)+1}$
  $t_{min} \leftarrow \infty$
  $color \leftarrow nextcolor$
end if

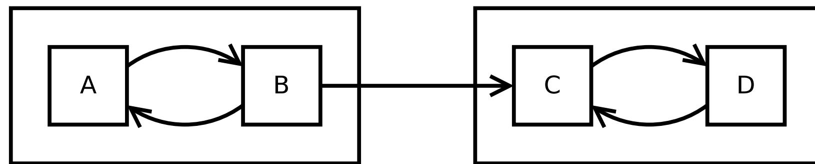

Figure 2.9: Mapping between atomic models and LPs

and B is now no longer under the influence of time warp and will impose **no** overhead at all. However, when model C receives a straggler, not only C will be rolled back, but D also, as they are under the same LP.

The following advantages of this approach were identified:

1. The number of LPs drops drastically and correspondingly the overhead associated with time warp will also drop;
2. Inter-core messages can skip all message serialization required to transfer using time warp;
3. Inter-core messages do not have to be saved, not when receiving them and not when sending them;
4. Inter-core messages will *never* become straggler messages;
5. As the number of LPs is drastically decreased, it becomes possible to increase the (implicitly used) lookahead by grouping models at the same core.

There is however a slight disadvantage: as all atomic models are grouped into the same LP, the complete node will have to be rolled back instead of a single atomic model. The problem becomes even worse when lots of atomic models are grouped in the same LP, as this causes more and more rollbacks. While this can cause slow performance in some cases, the performance is drastically increased in most situations. For these specific situations, one of our optimizations comes into use: *memoization* (Section 4.4.3). While this is basically comparable to lazy reevaluation, it is extended to the scope of every atomic model

seperately, making hits much more likely. When our distribution technique is combined with memoization, the disadvantages are strongly reduced in importance, while the advantages still hold.

Another solution to this problem would be to use hierarchical simulation locally, as was done in [33, 32]. However, PythonPDEVS performs direct connection (Section 4.2.2) which does not map that well to a hierarchical simulation methodology.

2.5 Classification

PythonPDEVS is classified according to the classification proposed by [2].

Some kind of tree transformation is performed in the form of reduction: direct connection is used, which reduces the height of the hierarchy to 1. Tree expansion is not used, as PythonPDEVS does e.g. not rely on subtrees for its distribution.

In terms of tree splitting, PythonPDEVS deviates from both the single and the multiple tree structure: every node has the complete tree and simulates all of it, though messages that are sent to models that are not local to the node, will be sent to the node that actually simulates this model. Additionally, models that are not local to the node will not be scheduled either. The main advantage of this lies with relocation: every node knows the structure completely, though it simply ignores its existence throughout the simulation. By sticking to a single tree (though in different incarnations at different nodes), relocations can happen at the level of atomic models instead of the complete tree. The actual simulated tree is thus the *overlap* of all simulation trees. Note that having the simulated tree at every node is not necessarily a memory problem, as only the actually simulated part will be accessed throughout the simulation. The inactive part of the tree can simply be swapped to disk by the operating system. While it is available for relocations, it doesn't affect the memory consumption. A lot of functionality is provided with this method though: the complete model is constructed sequentially at a single node, meaning that the use of e.g. random numbers is absolutely no problem for determinism. Also, selecting a different allocation of the model poses no problems as it is still constructed at the same node.

Node clustering is done using multiple processes. Every node has its own simulation loop and all processes communicate with each other.

Process distribution uses the multiple processors approach. True distribution requires distributed memory instead of shared memory, which is therefore also the chosen memory model for PythonPDEVS.

In conclusion, PythonPDEVS uses the $N-N-N$ strategy: many processors, many trees and many processes. All of this using an optimistic (time warp) algorithm.

2.6 Modelling PythonPDEVS

The PythonPDEVS simulator is written in Python, though the core is additionally modelled in DEVS too. By modelling the core algorithms (such as time warp synchronization, GVT approximation, relocation algorithm and the core of the DEVS formalism) in DEVS too, it is now possible to simulate (a reduced version of) PythonPDEVS with PythonPDEVS itself. It is a reduced version, in the sense that not all features are included for clarity. Some of the unmodelled features are Dynamic Structure DEVS, Classic DEVS, nesting and tracing.

By modelling PythonPDEVS, a DEVS model can be simulated within a *simulated* PythonPDEVS simulator. The simulated PythonPDEVS instance is a DEVS model too and is responsible for the simulation of the internal model.

2.6.1 Rationale

Several reasons for modelling (and subsequently simulating) PythonPDEVS exist, of which the most important in this context is deterministic debugging. In distributed simulation, behavior is non-deterministic as it is dependent on the execution order of different threads, the latency imposed on connections, ... This makes it extremely difficult to debug: reproducing the scenario often requires tedious rerunning of the simulation in search of the same problem, depending on how often the situation occurs. Furthermore, if some more information is desired (such as traces, logs, prints, ...), this will alter the simulation run, possibly avoiding the incorrect scenario entirely. Even worse: an attempt to fix the problem might seem to fix the problem, while it only made the problem become less frequent. For such reasons, deterministic debugging is required, which can be provided by modelling PythonPDEVS in PythonPDEVS.

Another important advantage is that the behavior of distributed simulation can be checked in unknown situations, allowing us to answer questions such as "What would happen if there was a huge delay between nodes?" or "Would simulation remain correct if there is a high amount of jitter?". While these situations can be emulated using network emulators like *netem* [2] , this requires intrusive changes to the system and is not completely deterministic either, as the problem with threading still remains. Some questions are unnecessary though, since our middleware (MPI) guarantees that no messages are lost, no out-of-order messages will be received, ...

[2]***netem***: `http://www.linuxfoundation.org/collaborate/workgroups/networking/netem`

Last but not least, simply modelling the simulator already reveals some bugs as the code had to be reviewed and a suitable DEVS representation had to be found.

The idea is based on [67], where a previous version of PythonDEVS is modelled with the intention of simulating difficult to reproduce scenario's for distributed simulation and see how crash recovery would handle some specific situation. In this thesis, the checkpointing and recovery algorithms were not modelled, as this was not part of the rationale and would needlessly complicate the model.

2.6.2 Assumptions

Due to some design choices in PythonPDEVS, we can make some assumptions for the model. Most of these are related to the middleware, for which we have chosen to use MPI. MPI uses TCP for its communication and thus allows us to make the following assumptions:

1. No packets will be lost during transmission.
2. Packets will arrive in the correct order. While this is not strictly required for a time warp implementation, we still require this for the initialisation of the simulation.

2.6.3 Model

Lots of code from the actual simulator was reused, as both use the Python programming language. Instead of importing, the code for core algorithms was copied as many modifications were required. Such changes relate to removing unused features, removing interaction with MPI, removing semaphores, ...

Pseudo-code for the model is shown in four parts:

1. Algorithm 2.5 shows the internal transition function;
2. Algorithm 2.6 shows the external transition function;
3. Algorithm 2.7 shows the output function;
4. Algorithm 2.8 shows the time advance function.
5. The confluent transition function is left at the default: calling the internal transition followed by the external transition function.

While most of the model is the same as the actual simulator, some important changes were necessary:

1. There is no MPI middleware to communicate with. Instead, a queue is added to the state which will be filled. In the output function, all messages in this queue will be output as a discrete event.
2. There is no threading in DEVS. While this could be emulated by using different atomic models, all threaded events are processed sequentially in one atomic model, with the added benefit that it becomes simpler to protect critical sections.
3. Tracing of the lowest simulated model is not possible, as this would require the transition functions to have side-effects that are not caught by the time warp implementation of the upper simulator.
4. As transition functions in DEVS should not block, locking was done differently.

Many parts of the algorithm refer to other functions, which are identical to these used in the actual simulator and are therefore not shown here.

In addition to a modelled version of PythonPDEVS, a modelled network is required which introduces some jitter, different arrival order, packet loss, ... all of them configurable.

2.6.4 Calibration

Calibration of the model is required in order to find parameters of the model so that it behaves similar to reality. The two most important values to find are:

- The transfer time of the network: this requires both an average and a distribution function, as there is some kind of non-determinism in real network latency. To guarantee determinism in the simulation, it is required to make this transfer time deterministic, e.g. using a pseudo-random number generator.

Algorithm 2.5 Modelled PythonPDEVS internal transition function

```
Clear queue
Clear transitioning
if should run GVT and at controller then
  Run GVT algorithm
end if
if GVT algorithm waiting then
  Try GVT algorithm
end if
if relocation required then
  Perform relocation
  return state
end if
if locked then
  return state
end if
newtime ← processMessages(simulation time)
if simulation finished then
  return state
end if
transitioning ← getImminent()
Clear output
for all adevs in transitioning do
  Add adevs.outputFnc() to output
  transitioning[adevs] ← 1
end for
for all port in outputs do
  if port destination is remote then
    Prepare the message and put it in the messagequeue
  else
    Put messages on port in the model input
    transitioning[destination] ← transitioning[destination] or 2
  end if
end for
for all adevs in transitioning do
  adevs.timeLast ← simulation time
  if transitioning[adevs] = 1 then
    adevs.state ← adevs.intTransition()
  else if transitioning[adevs] = 2 then
    adevs.state ← adevs.extTransition()
  else if transitioning[adevs] = 3 then
    adevs.state ← adevs.confTransition()
  end if
  ta ← adevs.timeAdvance()
  if ta = 0 then
    adevs.timeNext ← (simulation time, age + 1)
  else
    adevs.timeNext ← (simulation time + ta, 0)
  end if
  Append the state to the oldStates of adevs
  Clear input of adevs
end for
Reschedule all models in transitioning
Determine the next simulation time
return state
```

Algorithm 2.6 Modelled PythonPDEVS external transition function

```
for all port in inputports do
  for all msg on port do
    if msg is a simulation event then
      if destination is no longer local then
        Forward to new destination
      else
        if timestamp of message <= simulation time then
          Perform rollback to timestamp of message
        end if
        Schedule the message in a queue
      end if
    else if msg is a remote method call then
      Execute the method call
    end if
  end for
  Reset simulation time
  return state
end for
```

Algorithm 2.7 Modelled PythonPDEVS output function

```
return messagequeue
```

Algorithm 2.8 Modelled PythonPDEVS time advance

```
if message needs to be sent then
  return 0.01        {Placeholder: should represent the processing time of the middleware}
else if simulation not finished then
  return 0.1         {Placeholder: should represent the processing time of an event}
else
  return INFINITY
end if
```

- The computation time of the model: the time the transition function takes should be as close to reality as possible, to make realistic interleavings possible. A simple way of determining this time is by taking the actual time it takes in the model (as the function is really executed). This is not a recommended solution, as simulation would become non-deterministic again. A beter solution is the use of a solution similar to the network transfer time.

Such calibration results are clearly depent on the system (i.e. computation time of transition functions), the network (i.e. transfer time, latency, jitter) and the model being simulated (i.e. computation required in the transition functions). These values can be changed to represent other systems, thus making it possible to determine the behavior on such systems.

As simulation has to be correct in every possible set-up, these parameters do not matter when checking for correctness. If a "what-if" test is to be executed, these parameters should of course correspond to the situation that is tested.

2.6.5 Results

By simply modelling the simulator itself, some strange parts of the code were found and were corrected. One of the found bugs was caused by the naive modification of the GVT algorithm, which was used in previous versions. All code that was required to fix the problem in the modelled version was also copied to the real simulator. As this problem could have also been identified without the use of a modelled simulator, solving it would have been possible too. However, it is unknown whether or not such situation actually occurs (and is thus taken care of), or if it simply didn't occur in the test runs. The advantage of deterministic debugging becomes clear: the naive algorithm is used in the modelled simulator and a situation is searched where a problem occurs. Afterwards the problem is fixed in the model and it is simulated again. This guarantees that the problematic situation will deterministically occur, immediately showing the impact of the modifications.

Slight bugs were found during relocation too: messages and anti-messages that were still in the network during the relocation are still destined towards the old node. As soon as these are received, the old node should forward them to the new node. The GVT algorithm has to be notified of the forwarding too, as otherwise messages might not be taken into account correctly.

Some errors were due to the dynamic typing nature of Python: the GVT approximation algorithm uses the simulation time without the age field, whereas all other parts of the simulation use the age field. Therefore, in some cases the time was of the form `float`, and otherwise in the form `(float, int)`. Sadly, Python does not complain on comparisons such as `float < (float, int)`. One bug existed where the tuple was passed instead of the `float`, which caused some messages not to be included in the GVT algorithm because $5.0 < (3.0, 1)$ evaluates to `True`. This problem was quickly solved, as it was perfectly reproducable and adding trace functions and logging didn't interfere with the occurance of the problem.

3

Functionality

Together with improved performance, PythonPDEVS has various new features in contrast to previous versions. All features that are presented in this chapter are completely new, or are modifications of existing functionality. These modifications offer more applicability, makes the functionality easier to use or simply provides a more elegant implementation.

Since PythonPDEVS has been completely rewritten, all features were reimplemented though. Some functionality was removed, as it was too invasive in the kernel, while offering little usefulness. Not all features are shown here though, as some of them are only usable as performance tweaks (they don't add any functionality). Such features are discussed in chapter 4.

Each feature is presented (*how is it used?*), followed by its rationale (*why is it useful?*) and finally some small notes on the actual implementation in PythonPDEVS (*how is it implemented?*).

3.1 Classic and Parallel DEVS

Our simulator can be used in two different modes: either the Classic DEVS formalism is used, or the Parallel DEVS formalism. Both of these formalism are described in Section 1. Even though the simulation formalism can simply be toggled for the simulator, the user still has to use the correct formalism for its models and switching the formalism of models themselves isn't that trivial most of the time. For example, the *select* function is not used in Parallel DEVS simulation, so the user should not depend on it in Parallel DEVS. Additionally, Parallel DEVS models require a *confluent* transition function and use bags instead of simple messages on the port.

Rationale

Both formalisms have their own use cases, making support for both of them interesting. Classic DEVS has the advantage that it is somewhat conceptually simpler by not using bags (thus no requirement to be independent of the order) and not using a confluent transition function. As it is conceptually simpler, it is often used in educational settings. Parallel DEVS on the other hand, has the advantage that it is more efficient and has more possibilities for parallelization and distribution. Due to the increased focus on performance, the simplicity is slightly reduced, so it is mostly used in performance-critical situations.

Other simulators only choose for one of these formalisms (with the notable exception of JAMES II[27]), which makes it impossible to use that simulator (and its features) with a different formalism.

Implementation

Our simulator is built completely with Parallel DEVS in mind, since this is the formalism that can offer the highest performance. For Classic DEVS, some wrapper code was added to make it interact with the Parallel DEVS simulation kernel. This wrapper is responsible for putting all messages in bags, since Classic DEVS only uses a single message on every port. Some components of the simulation algorithm need to be altered, most notably the *select* algorithm.

Using Classic DEVS is not that efficient as Parallel DEVS due to all wrappers that are in place, though this doesn't violate our rationale.

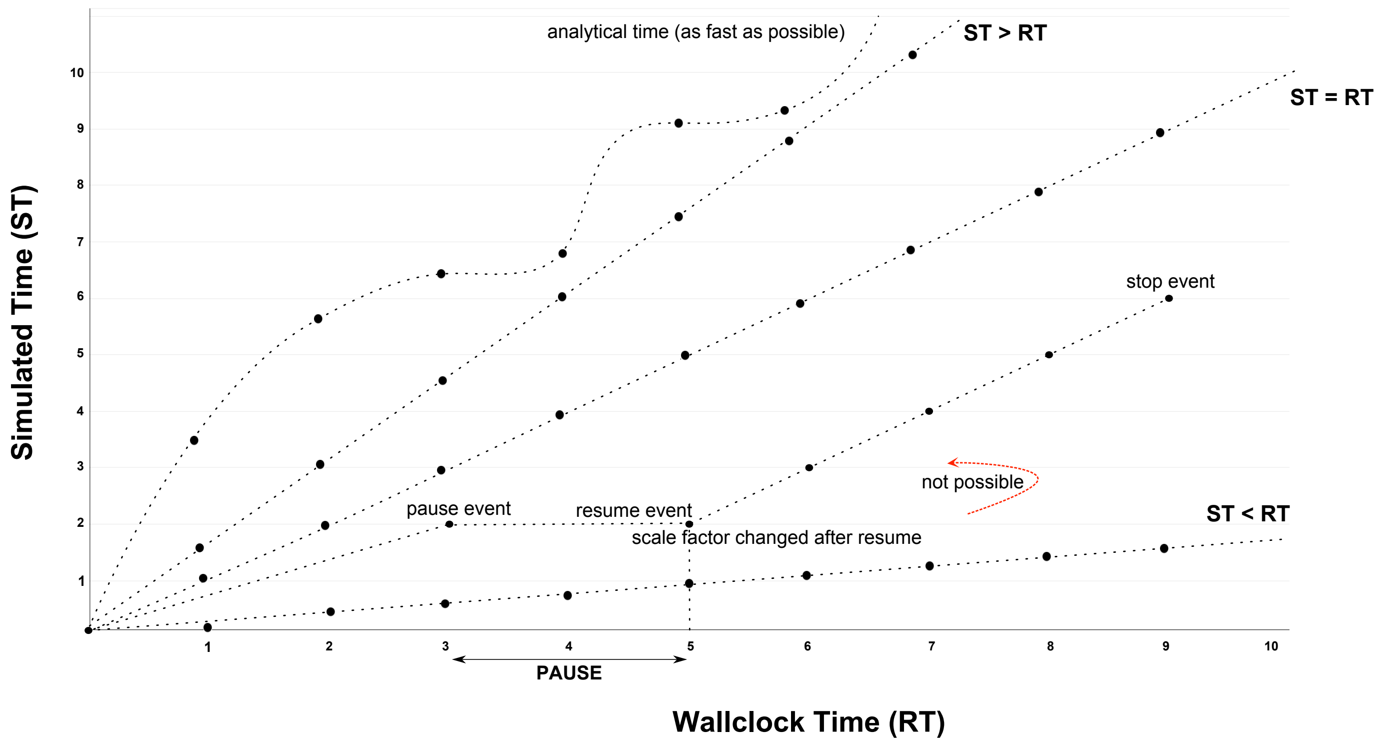

Figure 3.1: Comparison of different simulation speeds

3.2 Realtime simulation

Whereas many DEVS simulators only support as-fast-as-possible simulation, PythonPDEVS supports realtime simulation too. In realtime simulation, the time advance of a model will be used as the delay in *wall clock* time instead of simply an increase in simulation time. This means that there will be a link between both *wall clock* and *simulation* time. Due to the support for *scaled* realtime simulation, this link is not necessarily 1:1. Figure 3.1 clearly shows this difference: all realtime variants progress linearly in simulation time as wall clock time progresses. On the other hand, as-fast-as-possible simulation advances at the maximal speed, though it is wobbly as some intervals in simulated time contain more computation than others.

Furthermore, different realtime back-ends are supported to best fit the actual scenario in which it is used. These back-ends are further discussed in Section 3.2.1.

Previous versions also supported realtime simulation, though scaled realtime simulation and the game loop back-end are added. The realtime back-end was completely rewritten and the realtime kernel is more general and can thus be split off to be used in different projects.

Rationale

Realtime simulation is useful in a variety of domains, such as interactive simulation and realtime deployment. The model can be tested using as-fast-as-possible simulation, but actual deployment is as-fast-as possible.

Realtime simulation becomes even more applicable due to support for multiple back-ends, as it can be integrated in user interfaces, games or basic (command-line) testing.

Implementation

The implementation is tightly integrated within the main simulation loop for as-fast-as-possible simulation. This allows (nearly) all other features to be used in realtime simulation too without any changes. Polling is required to check for external interrupts, as this cannot be ran on its own thread due to the different back-ends.

It is necessary to take into account that the actual processing of a simulation step takes some time. The delay returned by the time advance is thus not identical to the time between both simulation steps. Take for example a transition that happens at time t_1 with time advance ta, so that the next transition happens at time $t_1 + ta$. Performing the transition will take a time t_{trans} and consequently the wait command will only be executed at time $t_1 + t_{trans}$. If a naive wait of time ta is performed, the actual

transition will only happen at time $t_1 + t_{trans} + ta$. It is thus necessary to subtract t_{trans} from ta before performing the wait, causing a transition at time $t_1 + t_{trans} + (ta - t_{trans}) = t_1 + ta$.

More information on the realtime kernel and its implementation can also be found in the specific realtime kernel documentation. Documentation is bundled with the realtime kernel module that can be used in other projects.

3.2.1 Back-ends

As was already mentioned, our realtime kernel supports a variety of back-ends. Adding additional back-ends is rather simple, as these back-ends are written as slim and modular as possible. Most realtime DEVS simulators in use today are limited to a single back-end (almost always thread-based). By supporting multiple back-ends however, the same model can be ran in a variety of situations.

An example of this is TkInter [1] , which has its own event loop. Using TkInter together with threads is known to cause problems, as TkInter should have complete control over event scheduling. By using a TkInter-specific back-end, this limitation is avoided and we completely rely on TkInter to perform all required scheduling.

A similar rationale is applicable for the game loop back-end, where the simulator is not responsible for timing but relies on an external component.

Raw threads

This is the most frequently used option for realtime simulation: simply have the Python interpreter do the required waiting on the current thread. While this approach is clearly the simplest, it can interfere with other modules, such as TkInter. Additionally, our tests have shown that the granularity of these built-in functions is rather limited in some situations.

Nonetheless, it is still the simplest back-end to use as it requires no user intervention or configuration. Its usage is encouraged in situations where no other back-end applies.

TkInter events

TkInter is often the choice when coupling a GUI to a Python program. However, TkInter has its own event scheduler to process GUI events and the like, interfering with a realtime kernel that uses threading. Our TkInter realtime back-end provides seamless integration with TkInter, by using the TkInter event scheduler for simulation events.

The only configuration that is required, is that the Tk root instance is passed to the simulator at configuration time. Otherwise, the realtime kernel would have no idea which instance to call in order to perform the scheduling. It is worth noting that the user should still manually perform the Tk `mainloop` call, though this is always required when running Tk.

Game loop

A game loop mechanism has practically the same reasoning behind it as TkInter events: the actual simulation might not be the only thing going on. The main difference is that the user himself is responsible for determining at what time the simulation is about to run, by calling a `step` function. Outside of these method calls, absolutely no simulation code is run whatsoever.

The user is required to call the simulation `step` method manually and the call will return as soon as all transitions up to that time are processed. If this method is called too late, a delay between the time of the interrupt and the processing will occur. The same is true for when a transition function should happen, but no `step` call is made: the processing will simply be delayed.

Overview

The different methods are shown in Figure 3.2. This immediately shows the difference in invocation: *raw threads* is a simulation-kernel maintained thread that will unblock at the specified times. *TkInter events* are identical, but the timing thread is maintained externally by the UI and consequently it also contains some UI events that are unrelated to the simulation. Finally, the *game loop* back-end has all timing done externally and events can only be processed as soon as the simulator is invoked.

3.2.2 Interrupts

The processing of interrupts within PythonPDEVS is again as modular as possible. Independent of the selected back-end, 2 different interrupt mechanisms are supported: file-based or manually calling an `interrupt` method.

File based

Using the *file based* interrupts, it is possible to specify a file that produces specified interrupts at the specified times. An example is shown in Listing 3.1. Each line has the form `time interrupt_port interrupt_value`, with the `time` being the absolute time at which the interrupt should happen, `interrupt_port` the name of the port as previously configured and `interrupt_value`

[1] TkInter (`https://wiki.python.org/moin/TkInter`) is a Python interface to Tcl/Tk (`http://tcl.sourceforge.net/`)

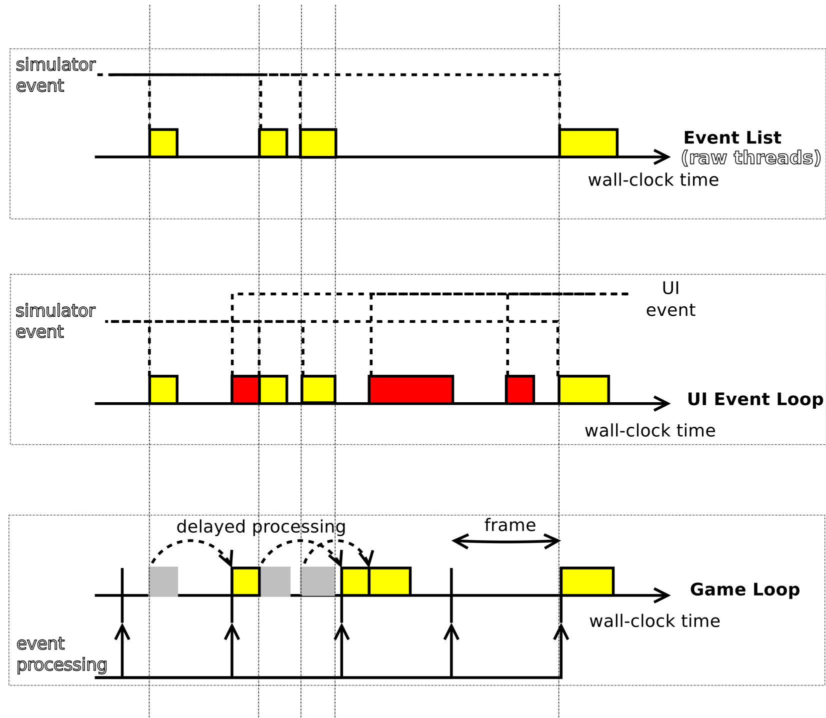

Figure 3.2: Different realtime back-ends

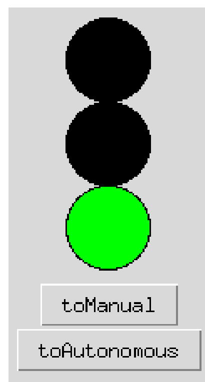

Figure 3.3: A GUI for the traffic light

the value to put on this port. For simplicity, the interrupt value always has to be a string, though the model can process the value to map it to an object of a different type.

Instead of simply parsing the file in one go, and scheduling all interrupts immediately, the file is traversed line-by-line to keep the approach scalable.

Manual interrupts

Manual interrupts are more advanced than the file based interrupts, though they are more versatile. Instead of providing a file with all interrupts and their times, this method requires the user to call the `realtime_interrupt` method on the simulator. This method takes a single parameter, which will be interpreted as the port on which the interrupt has to be generated, followed by the value to put on it. Both values are put into a single string object, seperated by a space, to maintain consistency with the file based method. The time is not required, as the time at which the call is made will be used as the time of the interrupt.

An example is shown in Listing 3.2, where the main thread is put into a loop and interrupts the simulator as soon as some input is provided from standard input. While this is a very simple example, more complex calls are possible too, such as calls caused by the receiving of network packets or due to UI events being processed (shown for Tk in Listing 3.3).

It is possible to manually write a file based interrupt generator using the manual interrupt mechanism.

```
10 INTERRUPT toManual
20 INTERRUPT toAutonomous
30 INTERRUPT toManual
```

Listing 3.1: An example interrupt file

```
while True:
    sim.realtime_interrupt(raw_input())
```

Listing 3.2: An example invocation of the interrupt method

3.2.3 Example

As an example of realtime simulation, we will show a basic traffic light `DEVS` model coupled with a TkInter GUI. The GUI will provide a visual representation of the traffic light and has 2 buttons: `toManual` and `toAutonomous`. It is shown in Figure 3.3.

The buttons should call the `interrupt` method of the simulator and pass the desired value. When pressing the `toManual` button, the yellow light will start blinking. When pressing the `toAutonomous` button, the light will restart its autonomous behavior. All simulation will be performed by PythonPDEVS, whereas the TkInter code will simply read out the state of the model and perform visualization.

3.3 Tracing

Even though most of the tracing functionality is identical to our previous version, the tracing framework became more modular. Previous versions required manual hacking of the code, whereas the new tracer depends on *registered* tracers. Due to this increased simplicity, tracers can be added at simulation-time without altering any code of the simulation kernel itself. The user is thus able to add a custom, possibly domain-specific, tracer with minimal effort.

For completeness, all supported tracers are briefly mentioned.

Verbose

The verbose tracer is the most basic tracer, whose main purpose is to provide human-readable traces. All required information is shown as simple as possible: the type of transition that happened, along with all inputs/outputs (whichever is applicable), the time of the transition, the simulation time of the next transition, . . . An example is shown in Listing 3.4.

XML

Our XML tracer is a modular version of the tracer provided by [64]. This is still a general-purpose tracer with automated processing as its main use. However, XML traces quickly grow in size due to the verbosity of XML.

XMLTracePlotter[64], as shown in Figure 3.4, is a tool for visualizing such XML traces. This same format is also used in SimStudio[69], which has its one visualization tool. Custom processing/visualization of the trace is fairly simple, as a variety of XML parsers exist.

```
def toManual():
    sim.realtime_interrupt("INTERRUPT␣toManual")

def toAutonomous():
    sim.realtime_interrupt("INTERRUPT␣toAutonomous")

root = Tk()
b = Button(root, text="toManual", command=toManual)
b.pack()
c = Button(root, text="toAutonomous", command=toAutonomous)
c.pack()
root.mainloop()
```

Listing 3.3: An example invocation of the interrupt method in Tk

```
__  Current Time:       1.00 ______________________

    EXTERNAL TRANSITION in model <Processor>
        Input Port Configuration:
            port <inport>:
                Event = 1
        New State: 0.66
        Next scheduled internal transition at time 1.66

    INTERNAL TRANSITION in model <Generator>
        New State: 1.0
        Output Port Configuration:
            port <outport>:
                Event = 1
        Next scheduled internal transition at time 2.00
```

Listing 3.4: Verbose trace of a single timestep in the simulation

VCD

VCD traces are created according to the Value Change Dump standard [1]. It is a domain-specific tracer in the sense that it is only usable for digital circuits. As its domain is restricted, it can take some shortcuts or present the trace in a more sensible manner. One of the biggest advantages is tool support: a variety of tools exist that can visualize VCD traces, such as GTKWave[2], which is also shown in Figure 3.5.

This tracer was implemented as part of [74], but was rewritten to fit into the modular tracer framework.

Cell

The Cell tracer is based on the CD++[79] tracer. CD++ is a DEVS simulator primarily focused on (Parallel) Cell DEVS, though it has basic support for Classic and Parallel DEVS. This tracer is domain-specific and requires every model to have coordinates associated with it. However, PythonPDEVS is a general purpose DEVS simulator, which has no knowledge about the actual location of these atomic models. To pass this information to the tracer, each model should have an `x` and `y` attribute, indicating its location. Models that don't have such attributes defined, will be skipped by this tracer.

An example is shown in Figure 3.6, where a visualization of a fire spread is shown. The actual tracer only generates textual information in the form of a matrix, which can be visualized as a heat map by e.g. Gnuplot [3].

Custom

As was already hinted on, custom user-made tracers are supported too. This allows (very) domain-specific tracers, that are able to extract only the required data and present it in a relevant manner. The simulator basically calls the provided tracer and will pass the model that is transitioning and the time at which it happens.

To make the development of such tracers simpler, a template is provided in the PythonPDEVS source code. Note that the tracers should not perform any irreversible actions, such as printing or writing to file. This is because time warp can be used, which might require rollbacks to be performed. All time warp-specific code is provided and the user should only register a *reversible* call on the controller. This call will only be executed when it is safe to perform irreversible actions, so it can contain such actions. For elaborate examples, we refer to the different tracers that are provided with PythonPDEVS.

3.4 Dynamic Structure DEVS

As mentioned in Section 1.7.3, Dynamic Structure DEVS is a variant of DEVS that allows changes to its structure at simulation time. Changes can range from adding new ports, to deleting complete coupled models. By supporting Dynamic Structure DEVS, the user is allowed to perform all such changes during the simulation.

Since modifications cannot happen during the transitioning phase[5, 6], these changes will be done in their own phase. Our approach is based on that of adevs[54], where transitioned atomic models have a *model transition* method called. This is the only method of the model that is allowed to alter the structure, to prevent conflicts. Changes can be propagated higher up in the hierarchy (by returning `True`), since some changes might not be the responsibility of the atomic model. Atomic models are allowed to alter their input and output ports, whereas coupled models can alter their ports, their submodels and the connections between them.

[2]GTKWave: http://gtkwave.sourceforge.net/
[3]Gnuplot: http://www.gnuplot.info/

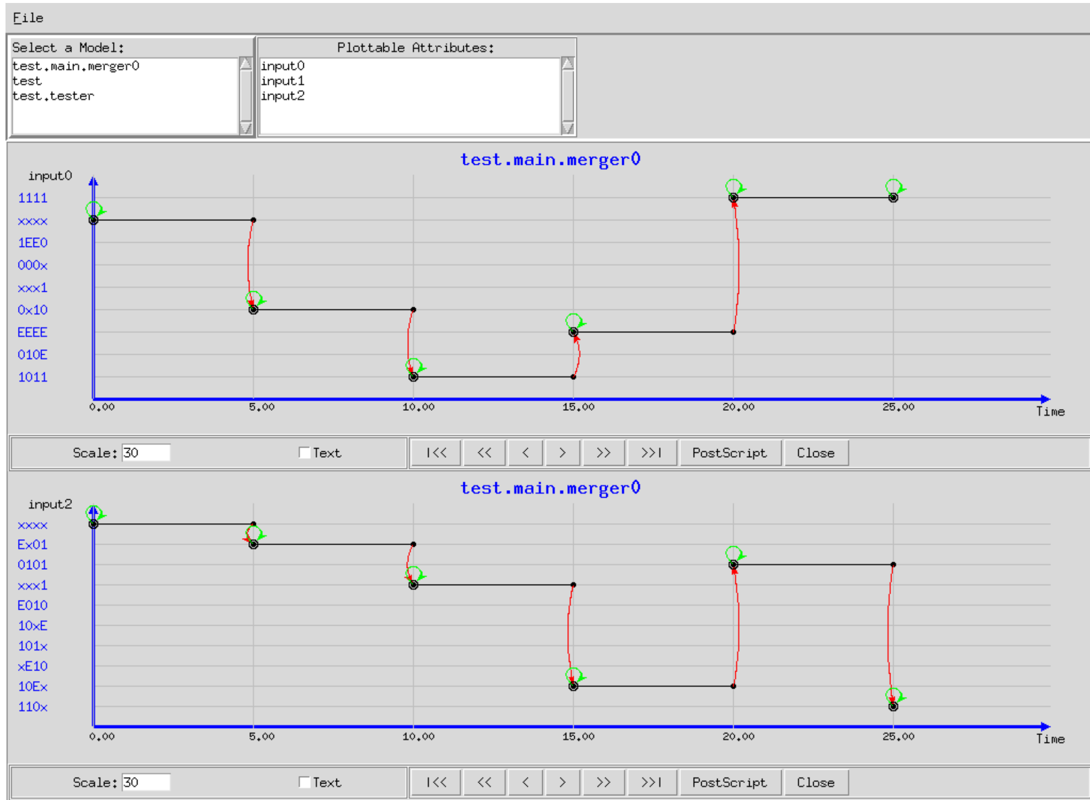

Figure 3.4: XML trace as visualized in XMLTracePlotter

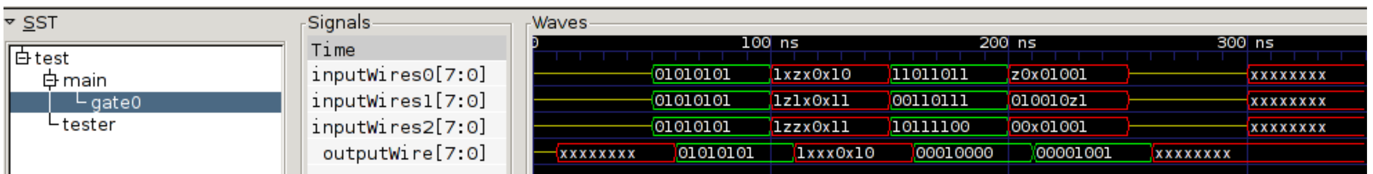

Figure 3.5: VCD trace as visualized in GTKWave

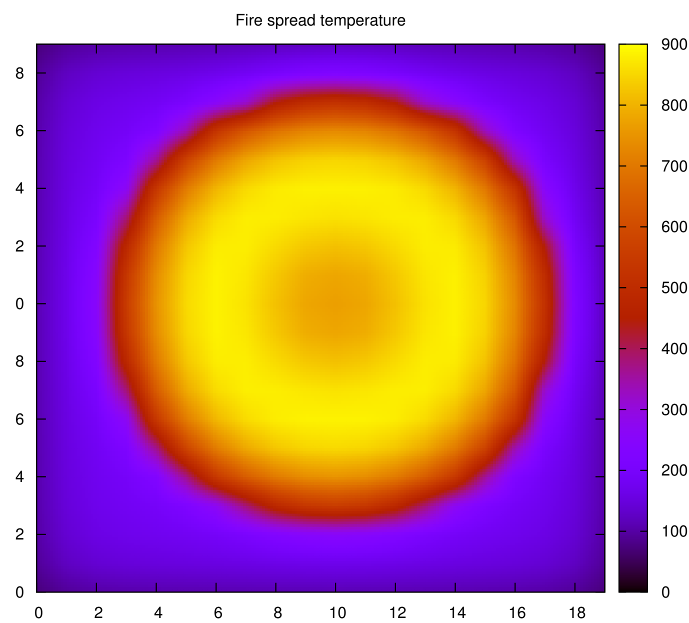

Figure 3.6: Cell trace as visualized with Gnuplot

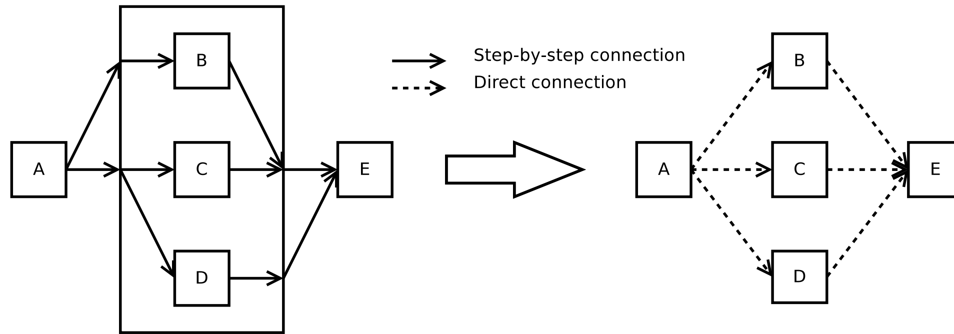

Figure 3.7: Direct connection introduces problems for Dynamic Structure DEVS

Dynamic Structure DEVS is supported in both Classic DEVS and Parallel DEVS simulation, but not in distributed simulation due to the bad match with time warp. With time warp, the simulation time at other nodes is unknown, so it would be possible that these nodes were restructured completely. If a message is then passed to that node, it is possible that the connection is no longer valid and that the message has to be forwarded. Furthermore, routing would be problematic as models might have been removed and can no longer be accessed remotely. This also means that not only the model states need to be saved, but also the complete structure of the model, making it infeasible for large simulations. We decided not to tackle all these problems, as a distributed Dynamic Structure DEVS simulation would be unlikely to cause a speedup in the average case due to all these problems that have to be solved at simulation time. It should be clear that conservative simulation (as in adevs) is a much better match for such dynamic formalisms.

Rationale

Dynamic Structure DEVS is not really an extension of DEVS[5, 6], since all such models can also be mapped to equivalent Classic DEVS models. The mapped Classic DEVS models would simply create all possible connections at startup and only use those that are flagged as active. However, this would put a lot of responsibility with the user, as he must first determine all possible combinations, create them manually and finally still needs to write all kinds of activation code in the model. In the worst case, the mapped Classic DEVS model would be infinite in size as time progresses. A simple example of this is a model that spawns a new model after a certain time step. If simulation runs for infinitely long, the number of created models also goes to infinity.

By supporting Dynamic Structure DEVS at the simulator level, such changes are possible in a structured way, preventing the user from having to think about all possible situations. That way, models are created *on-the-fly*, introducing no overhead at moments where the model didn't exist yet. This makes it possible for tracers to be notified of structural changes, offering some kind of notification to the user. To make this possible with the equivalent Classic DEVS model, the user could still write a custom tracer as explained in Section 3.3, which notices such 'changes', e.g. by inspecting states.

So whereas at first it doesn't seem necessary to actually implement Dynamic Structure DEVS at the kernel level, it offers some important advantages. Some state that Dynamic Structure DEVS can be used to increase performance, by removing models that are no longer used. This is not necessarily the case in PythonPDEVS though, thanks to the use of *activity* which is explained in chapter 5.

Another use case would be to support some kind of hybrid between *agent-based* simulation and DEVS simulation, since agent-based models heavily depend on such capabilities.

Implementation

Some drastic changes were necessary to allow structural changes. First of all, the main simulation loop had to be changed to include a *model transition* step as shown in Algorithm 3.1.

Secondly, PythonPDEVS uses *direct connection* to speed up simulation (Section 4.2.2). This means that all internal connections are restructured by removing all user-created step-by-step connections and adding a new direct connection that is able to cross through multiple coupled models at once. An example of this is shown in Figure 3.7. If the connection between model *A* and the port of the coupled model going to *C* and *D* were to be removed, only the connection between *A* and *B* should remain. However, the direct connection no longer has any information about which connections (or ports) were used to build up that connection. Dynamic Structure DEVS models are able to create or remove step-by-step connections though, so it is required to save all step-by-step connections separately. Related to this, PythonPDEVS also supports transfer functions (Section 3.12), which causes similar problems.

To address these problems, the direct connection algorithm was altered to create these direct connections as another kind of connections. When routing messages, the direct connected links are used, but when the user is accessing the links, the step-by-

step connections will be used. After every modification to the connections, all these routing connections are recreated. The same applies for the transfer functions. Rerunning the direct connection process at every modification is not that efficient, though we implicitly assume that structure changes are rather infrequent, as in [45].

Algorithm 3.1 Dynamic Structure DEVS simulation algorithm

```
clock ← scheduler.readFirst()
while terminationCheck() do
  for all scheduler.getImminent(clock) do
    Mark model with intTransition
    Generate and route output
    Mark destinations with extTransition
  end for
  for all marked models do
    Perform marked transition
    Send the performed transition to the tracer
  end for
  scheduler.massReschedule(transitioning)                    {Begin of new code}
  if use DSDEVS then
    for all marked models do
      if model.modelTransition(dict) then
        Add parent of model to marked models
      end if
    end for
    Rerun direct connection
  end if                                                       {End of new code}
  Clean model transition marks
  clock ← scheduler.readFirst()
end while
```

3.5 Checkpointing

Checkpointing refers to the capability of a simulator to recover from a crash, using *checkpoints* that were created during the simulation. Checkpointing makes it possible to add some additional code in the experiment, to perform a recovery if a checkpoint file is available. Should a checkpoint be available, simulation should continue from that checkpoint on and with the same configuration options. If no (valid) checkpoints are available, simulation starts from scratch again. Clearly, the results of a simulation that ran in one go, or one that required (possibly several) recoveries should be identical.

Previous versions already supported basic checkpointing, though some additional features were required, such as checkpointing of sequential simulations, performing sanity checks on the checkpoints and automated recovery.

When a recovery attempt is made, the simulator automatically searches for checkpoint files in the current directory and attempts to load the most recent checkpoint. Possibly related to the reason for the crash, the most recent checkpoint might be corrupt, e.g. because files were not flushed to disk. To handle this, the simulator should recover to the most recent, completely correct checkpoint.

Rationale

As soon as multiple nodes start cooperating on a single simulation, the chance of a crash increases rapidly. Not only the nodes can crash, but also the network connecting them. Since distributed simulations are mostly used for huge simulations, the simulation probably took a long time already, making a crash even worse. Checkpointing mostly mitigates this danger, because simulation can be continued from the last checkpoint.

Implementation

In our implementation, checkpoints can only be created at GVT boundaries [4]. This is simply because a GVT boundary is the only time at which a global synchronization is done. Checkpointing a time warp implementation at other times is very difficult, since all nodes are at different simulation times. Since all states except for the state at the GVT are still uncertain, only the state at the GVT will be stored. All previous states are removed by fossil collection and all future states are not saved for space

[4] For sequential simulation, this means that we will also have to run the GVT algorithm, which is trivial if only a single node is present.

considerations. This also makes it possible to ignore all saved inter-node messages.

Of course, if we were to save a checkpoint at every GVT calculation, we might end up with a lot of checkpoints and consequently a lot of overhead. Lowering the delay between two GVT calculations might be interesting to perform fossil collection faster, but this would then also increase the amount of checkpointing done. In the worst case, we would be creating a checkpoint every second, which is clearly overkill. To resolve this issue, a configuration option is provided that determines how many GVT boundaries should pass before a checkpoint is made.

Our checkpoints are simply based on Python's `Pickle` module, though we made several changes to the pickling behavior. We don't need to save all temporary states, temporary messages, locks, ... so these can be skipped. This skipping is not only done for space considerations, but also because the `Pickle` module cannot handle some objects such as locks.

During recovery, we request all nodes to test their version of the checkpoint before we actually determine the checkpoint to actually restore to. This has a slight overhead by unpickling the checkpoint at least twice, but it handles the situation where one of the checkpoints is corrupted which would otherwise crash the simulation again.

Tracers are notified that they should recover too, indicating that they should e.g. not overwrite the file, but append to it. However, due to technical limitations, it is possible that trace files become corrupted or incomplete. All file I/O is done using buffers, so if the simulation crashes while these buffers are still being flushed, these traces probably become incomplete. If the user really requires this kind of transactional functionality in the tracer, it is always possible to write it manually with a custom tracer.

3.6 Nested Simulation

Nested simulation is the possibility to start another DEVS simulation within the currently running simulation. A nested simulation can simply be called by instantiating a new `Simulator` object within the currently running simulation, for example within a transition function. A nested simulation is required to be sequential, though the nesting simulation is allowed to be distributed. Previous versions of PythonPDEVS did support distributed simulations to be nested, but this required a lot of work from both the user and the simulation kernel because all nodes needed to be locked. Having a nested sequential simulation is not that difficult and is completely transparent to the user.

An easy solution to nesting would be to start a new Python interpreter and run the nested simulation there. This does not provide an integrated approach and all communication between both simulations should happen using standard input and output. Starting a subprocess is also less straightforward than creating and simulation a new `Simulator` object.

As is to be expected, a nested simulation is unable to access its calling simulation, whereas the other way is possible: the caller of the nested simulation will have access to the simulated model.

Rationale

Apart from forcing a clear design without global variables, nested simulation also has several advantages for the user. The most important advantage is that it allows users to have a simulation that depends on another simulation. A contrived example of this would be where the time advance of a specific model depends on another simulation (possibly with a finer granularity), which is possibly influenced by the current state of the original model.

Of course, we simply offer this functionality to the user and it is still the user that has to judge whether or not this nesting is usable in the situation at hand.

Implementation

The implementation for nested simulation basically requires the absence of global variables. An example of this is the tracer: previous versions of PythonPDEVS simply had the tracer as a global variable, which gets called from everywhere in the code. With nesting, such situations must be avoided, as it would mix information from two different simulations.

The most important design change that was required for this, is the use of multiple `Server` objects. A `Server` is responsible for all communication with other simulation kernels, running at remote nodes. In sequential simulation, the node is always 0, which would cause problems if the nested simulation were to run at e.g. node 2. To solve the problem, the server will have its kernel and remote mappings overwritten during a nested simulation. After the simulation, the previous kernel and mappings are restored in order to continue the lower simulation as if nothing happened.

As some options are unavailable in a nested simulation (such as nesting), it is required to detect whether or not the current simulation is nested. Two global variables were used to handle this: whether the current simulation is nested and whether or not a simulation is running.

A special case to think about is that it is possible for multiple simulations to happen sequentially in the same Python interpreter, but not nested. This would be possible when the simulation is called in a loop for example. For this, we require additional maintenance of all global variables.

```python
from simulator import Simulator

class NestedProcessor(Processor):
    def __init__(self):
        Processor.__init__(self, "Nested")
        self.state.my_param = 15

    def timeAdvance(self):
        sim = Simulator(MyModel())
        sim.setTerminationTime(self.state.my_param)
        sim.simulate()
        result = max(sim.model.state.result, 1)
        return result
```

Listing 3.5: A simple nested simulation invocation

Example

A small example to show how easy this is to use, is presented in Listing 3.5. It shows a Processor model of which the `timeAdvance` method is dependent on the simulation results of another (unrelated) model. In the `timeAdvance` method, we simply instantiate another `Simulator` with the new model, possibly do some configuration, and start the simulation. This is exactly the same usage as if it were an experiment file, so in the most extreme case this code could even call other experiment files. At the end of the nested simulation, the results are obtained from the simulated model and this is returned as `timeAdvance` value.

More complete examples are shown in the documentation.

3.7 Termination condition

In contrast to a termination time (expressed in simulation time), PythonPDEVS also supports a termination condition. A condition can be provided instead of the termination time and will be checked at every simulation step. If the condition returns *True*, simulation will stop, otherwise it will continue.

This termination condition will have access to both the simulated time and the current (complete) model. It can contain everything that is desired and is yet another Python function in our case. While it is possible to have a termination time by writing a specific termination condition, a termination time is added specifically as it allows the simulation kernel to do some optimizations.

In distributed simulation, only the state of local models is known, whereas the state of remote models is unknown and might not even be computed yet. PythonPDEVS supports two different approaches, which can be combined:

- **forced-local**: at the start of simulation, the kernel is notified by the user of all models that will be used in the termination condition. These models will then be forced to run at the controller, making the termination condition a simple local termination condition without any network overhead at every simulation step. Relocations of these models will become impossible and will simply be ignored by the simulation kernel. However, it is possible that this causes a huge amount of rollbacks in some situations. In case the model is only accessed very infrequently, this might be unnecessary.
- **pull-based**: the model can run wherever is required and all accesses to the state will cause the controller to ask the complete state of the remote kernel. Clearly, this causes a huge overhead in most situations, so it is only advised when the state is rarely accessed. If the remote node is already further in the future, the old state will be returned in order to create a consistent snapshot. Should the remote node not have progressed that far in simulation time, the call will simply block until the actual time is reached.

Hybrid methods are possible too, where some model is checked frequently (using *forced-local*) whereas others are only checked (using *pull-based*) if certain (strict) conditions are met. A *push-based* method is not supported, as this would cause a flood of remote messages in most situations. If a push-based method is desired, it is best to use the forced-local approach.

Even though (basic, local) termination conditions were already supported in the first versions of PythonPDEVS, the possibility for distributed simulation adds some more difficulty to the methods that are used, which is why it is still mentioned.

Rationale

Allowing the user to define a custom termination condition, instead of simply simulating up to a specific time is useful due to the addition of *domain-specific* information. For example, in design space exploration some models might be known to be useless (in terms of being a candidate for further exploration) after only a fraction of the time it takes for a full simulation run. In such cases, it would be a waste of time to completely simulate the model up to the desired termination time, as the results will simply

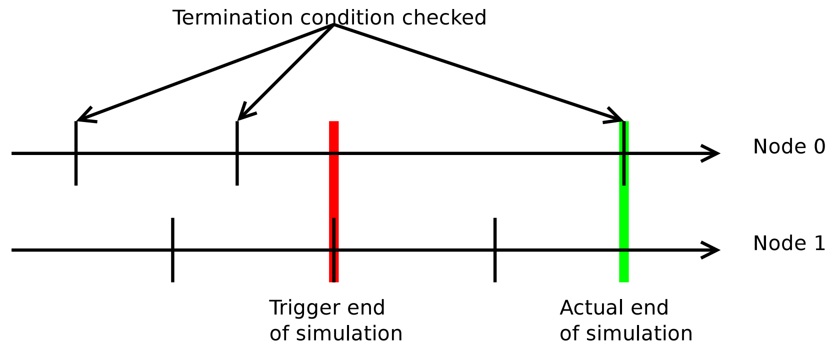

Figure 3.8: Distributed termination function

be thrown away. That way, the additional overhead for running a termination condition is mitigated by the (potentially huge) saving on early stopping a simulation.

Of course, it is possible to set a termination time and do manual polling of the model thanks to the continue functionality (Section 3.8), though this would require more coding by the user and will only check the condition after some time.

Implementation

For sequential (and thus local) simulation, the implementation for such functionality is trivial. Distributed simulation on the other hand, is rather difficult because all nodes are at a different simulation time due to time warp being used.

The check will always happen at the controller and all other nodes will simply simulate until infinity. As soon as the controller detects the end of a simulation with the condition, it will notify all other nodes of the controller's current time, which will then become the termination time of these nodes. If the nodes have already progressed past this time, their simulation will halt (though not yet reverted), otherwise they will continue up to that simulation time. No rollbacks will take place as soon as a termination time is received that has already passed, as it is possible for the controller to *invalidate* its previously sent termination time due to the controller being rolled back.

As soon as the controller is certain that it terminates at the broadcast time, simulation will halt. There is only one way to know this for sure: when all nodes have halted, for which we use the normal termination detection algorithms. We cannot wait for the GVT to progress past this time, as it is possible that some nodes hold the GVT back for these cases.

The implementation of both the force-local and pull-based methods is fairly trivial: force-local will simply alter the allocation at the start, whereas pull-based will iterate through the list of saved states until it finds the required time or block otherwise.

Notes about termination condition

The termination condition is different in terms of when it halts in both local, distributed and realtime simulation, even though it is supported in all of them. In local simulation, the termination condition is as to be expected: it is checked at every simulation step and can access all models.

In distributed simulation, the termination condition is only called at simulation steps that happen at the controller and it is only allowed to reference models that run at the controller, or remote models using the designated interface (introducing a significant overhead). This seemingly strange behavior of the simulation steps means that local and distributed simulations are not always exactly the same (in terms of termination time) in the case where a termination function is used. An example is shown in Figure 3.8: local simulation would end at the red line, whereas distributed simulation will only detect termination at the green line. If a remote model has to trigger the end of the simulation, simulation will only be halted as soon as the next simulation step at the controller happens. This causes a slight difference: with distributed simulation, an event from after the local termination time is still executed and traced. Such problems are easily solved by placing the models that can trigger termination at the controller (which is also a good idea performance-wise).

Realtime simulation is similar to local simulation, though this requires some special attention: there is a timeout between the different simulation steps due to the realtime simulation, so the condition is only checked after this delay. An example is a simulation step from time 5 to 15, with a termination condition that checks for time 10 to terminate. After time 5 has passed, a 10 second delay will be introduced, after which it is determined that $15 > 10$ and simulation should stop. The termination function is not executed continuously, as this would put unnecessary load on the system. Should exact termination at this time be required, it is possible to use the termination time functionality.

```
__ Current Time:      100.00 ___________________________________________


   EXTERNAL TRANSITION in model <Chain.CoupledProcessor_2.Processor0>
       Input Port Configuration:
           port <inport>:
               Eventsize = 1
       New State: 0.66
       Next scheduled internal transition at time 100.66


   INTERNAL TRANSITION in model <Chain.CoupledGenerator.Generator>
       New State: 1.0
       Output Port Configuration:
           port <outport>:
               Eventsize = 1
       Next scheduled internal transition at time 101.00


   USER CHANGE in model <Chain.CoupledGenerator.Generator>
       Altered attribute <model.state.value> to value <2>
```

Listing 3.6: A verbose trace after a modification happened with continuation

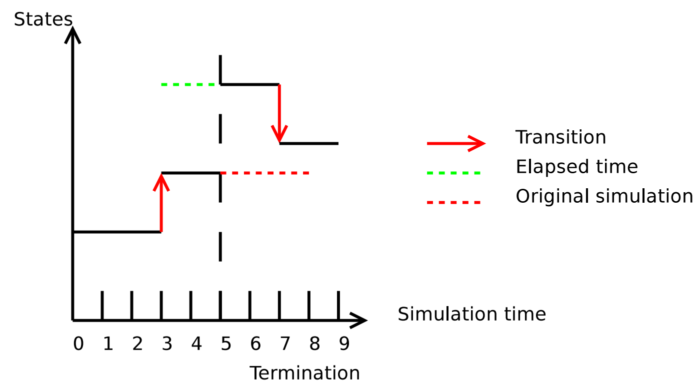

Figure 3.9: Example of continuing a simulation at simulation time 5

3.8 Simulation continuation

After a `simulate()` call has successfully returned, it is possible to call the same method again and thus continue the simulation. Tracers can be added or removed and some simulation options can be altered, before continuing with the new configuration. It is also possible to modify the termination time and thus execute the simulation in blocks.

Even more importantly, the state of models can be altered in between `simulate()` calls. After a modification to a model has happened, the time advance function of that model will be recomputed and the time of the next transition will be computed based on the elapsed time and the newly computed time advance. Of course, the resulting time should not fall before the termination time that was used to prevent causality violations.

Such model changes are notified to the tracers, which can then log the occurrence of a *user event*. Tracers might choose not to implement this and only the *verbose* tracer supports this by default, as shown in Listing 3.6.

An example is shown in Figure 3.9. The first time, the termination time is set to time 5, after which control returns to the user. The last transition happened at time 3 and the next transition should happen at time 8 (the time advance was thus 5). Due to the user event at time 5 (as this was the termination time), which changed the state to have the time advance return 4. Since these computations should be transparent to the user, the elapsed time is taken into account and the time advance will be computed as if it happened at time 3. This gives a time of 7 for the next transition. In this example, the time advance should return *at least* 2, as it should be at least the elapsed time to prevent causality problems.

Rationale

Continuing a simulation is useful if the model has an associated warm-up period, which would pollute e.g. a statistics gathering component. Another use case is when a period in simulation time is required where no tracing should be done (for performance or space reasons), but near the end this tracing becomes necessary (or vice-versa). This functionality was also used in a benchmark later on, to perform a simulation in different block.

Implementation

Continuing a simulation in a local simulation is fairly trivial, as we simply have to start up the loop again. All initialization can be skipped, though configurations should be rechecked as they can be altered.

In distributed simulation, this is somewhat more difficult due to the various distributed algorithms that also have to be restarted. Apart from this, we should not shut down the MPI server after simulation finishes, as the controller might decide at any time to restart it. A shutdown is only allowable if the server received a finish signal from the controller.

To recalculate the time at which we should transition, we need to compute the `elapsed` attribute before any modifications are made. This can be computed by subtracting the time of the last transition of that atomic DEVS model from the termination time. Even if a termination condition is used, the kernel will still convert this to a termination time internally (to signal remote nodes), so it will be usable in both situations.

3.9 Model reinitialization

With model reinitialization, we allow the simulation to be reinitialized after a simulation run. After reinitialization, it is possible to run the `simulate()` method again and replicate the previous simulation run. Since exactly the same simulation run is not useful, it is possible to modify the states of the model as with model continuation. Such changes could include altering some attributes of the state or of the model itself. It is mandatory to use these methods instead of simply altering the model, as the model could be distributed and the changes need to be propagated correctly.

In between different simulation calls, it is possible to add/remove tracers, toggle simulation options, ...just as with simulation continuation.

Rationale

The ability to reinitialize models is definitely simpler than creating a completely new `Simulator` instance with a new model. The reinitialization of a model is also faster, because all setup code can be omitted. In distributed simulation the broadcasting of the complete model can be avoided altogether.

Use cases are readily available, most of them being related to optimization, e.g. design space exploration. The model is simulated with slight modifications to some attributes and statistics are gathered every time. This way, it is possible to search for the optimal value for the parameter.

Implementation

For our implementation, we simply keep a pickled version of the model to be able to restore it after simulation. As soon as the reinitialization call is made, this pickled version will be extracted and will overwrite the working version of the model. The simulation time will also be reset and all tracers cleared.

It is necessary to keep a pickled version of everything and not simply of the states, as we should be able to handle the situation of Dynamic Structure DEVS too. In distributed simulation, this imposes no additional overhead because we had to pickle the model anyway to broadcast it over the network.

In local simulation, this pickling is considered an overhead. If a pickled copy is created, simulation will not alter the original model and states will not be updated automatically. For these reasons, local simulation requires the setting of a configuration flag to perform the pickling and induce the overhead.

3.10 Random numbers

In PythonPDEVS, a random generator can be instantiated and saved inside the model state. This object then has several methods to ask for a random number. Such a random number generator can be implemented by the user, though a basic generator is provided that should suffice in most situations. The default generator is merely an interface to the Python `random` module, but with the exception that it works deterministic in distributed simulation.

For sequential simulation, there is no advantage to using a random generator over the `random` library. In distributed simulation, the user has the option for using the provided generator, which provides deterministic results. The values are deterministic in all situations: whether or not simulation is distributed; whether or not the model is relocated; whether or not rollbacks on the model

happened; whether or not memoization is used; ...

To allow this feature to be used together with all our performance enhancements, all of these enhancements are supported in the generator.

Rationale

There is a clear need for random numbers in simulations. However, these should be pseudo-random as they have to be repeatable upon command. Most random number generators provide a *seed* to initialize the generator, if the same seed is used, the sequence of random numbers will be identical. Repeatability is important in a variety of circumstances, examples of these are benchmarking and debugging (as was also one of our arguments in Section 2.6) of models.

Using the `random` module in the model itself is guaranteed to be non-deterministic in distributed simulations, even if the seed is identical. Rollbacks will cause the complete simulation at that node to revert to the desired point in simulated time, but leaves the random number generator untouched. Consequently, random numbers will not be the same as in previous transitions, even though this is required for deterministic simulation.

Furthermore, if memoization (Section 4.4.3) is used, only the states of models are compared and nothing is said about the state of the random number generator. With relocations, the model starts using the random generator of another interpreter, which causes even bigger problems.

To solve these problems, we could patch all simulation code and also save the random generator state separately (and restore it where necessary). But this would still not give exactly the same results as if we were to use local simulation and still cause problems with relocations. Another reason is that in Parallel DEVS, the order in which transition functions happen is *inherently non-deterministic* (e.g. in PythonPDEVS this is dependent on the hash of the model, which varies between simulation runs).

Other solutions could include a *global random generator*, which will then serialize the request for random numbers. However, this solution is centralized and doesn't fit well in a time warp implementation. To make matters worse, it would only solve part of the problem, as it doesn't help in memoization.

Implementation

Our implementation is based on an object that is initialized with a specific seed, just like any other random number generator. However, the seed is local to the model, so calls for random numbers only alter the state of the local model itself. As a result, all operations on the random generator are local to the model, making it independent of the location in distributed simulation.

Since the generator is part of the state of the model, it will automatically be saved with state saving too and it will also be taken into account by memoization. The generator also has methods for custom state saving (as otherwise a reference would be stored) and for memoization (only the seeds should be compared).

The actual implementation is rather simple, as seen in Algorithm 3.2. The saved seed is stored as an attribute of the object. All calls to `random` are to the `random` module provided by Python.

Algorithm 3.2 Random number generator code

$random.seed \leftarrow savedseed$
$returnvalue \leftarrow random()$
$savedseed \leftarrow random()$
return *returnvalue*

3.11 Halted simulation detection

When the simulator detects that a simulation is taking excessively long at the same at the same simulation time, an error should be thrown and simulation has to stop. It is possible for simulation to halt due to a model that constantly has a time advance that is equal to 0.

This does not necessarily have to be in a loop, like in Causal Block Diagrams. Even a single model that continuously returns a time advance of 0 is enough, as simulated time doesn't progress at all.

Our detection is not perfect in the sense that it might return false positives. The simulation will be aborted if the same simulation time is kept for too long, by default 10.000 simulation steps. This abort will happen even if the time advance might eventually differ from 0, as it is only a naive approach. However, the simulation kernel cannot know this as the model is written completely by the user in a programming language.

A more invasive method would be to check whether the state changes after a transition was called. Finding exactly the same state for every model (possibly in a loop) would then indicate that the situation would eventually repeat itself. That method would have multiple problems:

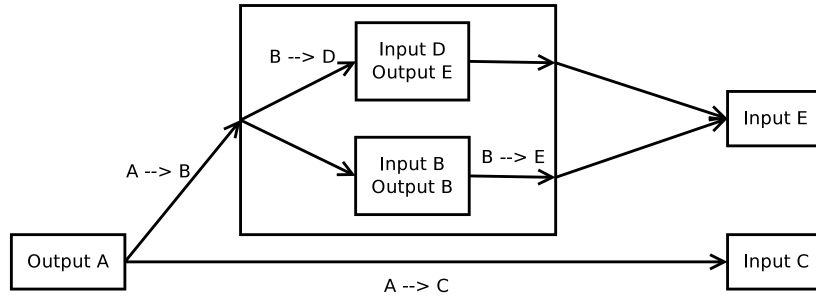

Figure 3.10: Example of transfer functions

1. We would need to do state saving in every simulation and for every atomic model, even for those that are local and that will never have a time advance equal to 0
2. A comparison between different states is required. Whereas this is already required for memoization, such a comparison method is still optional and would thus not help us in the general case.
3. Even if the state changes, it might be possible that the time advance will stay 0 forever. An example of this would be a model that keeps a counter in its state for how many times it has transitioned (which has nothing to do with the value of the time advance).

Rationale

According to the DEVS formalism, algebraic loops and *infinite simulations* are not allowed, since these are an abstraction of the real world. In reality, nothing can happen infinitely often within a finite timespan. The problem lies in detecting them, which would require a lot of effort and require some more information about the model and its state than is currently available. Classic DEVS already has some kind of basic protection against such loops, by preventing self loops.

Sadly, detecting loops with perfect accuracy is seemingly impossible without any kind of symbolic solving of the DEVS models. Therefore, only a basic algorithm is used, which is can find a few false positives, though it will interrupt every algebraic loop.

Implementation

Our implementation is fairly trivial: we keep a counter that is incremented if the next simulation time is equal to the current time and set to 0 otherwise. Since we use an *age* field for our distributed simulation, we already have this *age* field without any changes. We only need to check whether or not the *age* field is larger than a specified threshold at every simulation step.

3.12 Transfer functions

A transfer function is a function that can be coupled with a connection. All messages that pass over this connection will then be converted using the provided transfer function. These transfer functions will simply be given the (individual) messages as argument and the return value is used as the new message. Since they are coupled to the connection, it is possible to chain these functions just like it is possible to chain connections.

An example is shown in Figure 3.10. All atomic DEVS models are annotated with the types they take as input and the types they create as output. Connections are annotated with the type conversion they perform. If there is a mismatch between them, a transfer function is added to the connection to perform the required translation. Connections without an annotation do not have a transfer function defined (or it is equal to the identity function).

Rationale

Support for transfer functions is a requirement according to the DEVS formalism. We do consider this a feature though, as nearly no DEVS simulator actually provides support for this. It is even a *forgotten* requirement in the testing framework presented in [39].

The DEVS formalism requires the presence of transfer functions, because they allow easy reuse of models. As an extremely simple example: assume that we have an atomic DEVS model that outputs a float, but the model that it is coupled to only accepts integers. In that case, the transfer function could be used to convert the float to an integer in the way that is required. It would also be possible to adapt either of the two models, to do the conversion there, though reuse would suffer from this. Adding another atomic model that does the conversion in between (with a time advance equal to 0) would also be possible, though it would increase the complexity of simulation and would alter the *age* fields previously introduced.

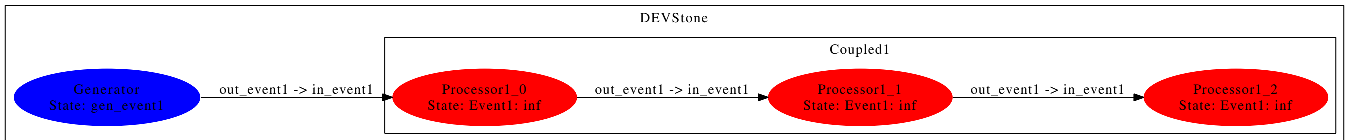

Figure 3.11: Visualization of a model

Implementation

Implementing transfer functions is fairly trivial in the abstract simulator, though our simulator is a lot more complex. The main problem is that we use *direct connection*, which will remove all step-by-step connections as was already mentioned in Section 3.4. These transfer functions basically require the same methodology as with normal connections: they need to be chained in-memory and be saved together with the one-step connections. As was the case with step-by-step connections, the originals still need to be saved for when the function is being changed later on in Dynamic Structure DEVS.

On the performance side, calling a function for every message that is being passed is clearly computationally intensive, even if it were the identity function. For that reason, no function will be called if no transfer function is defined, instead of calling an identity function.

To make sure that modularity is still enforced, copies of the events need to be made both before and after the transfer functions. Before, to make sure that the transfer function doesn't alter the original message, and after, to make sure that multiple models that receive the same event don't interfere with each other. All message copying is always done using the Python `pickle` library.

3.13 Visualization

One of the features that partially improves usability is built-in support for model visualization. The model is saved in a format that can be understood by GraphViz [5] , which will then be responsible for the actual rendering.

One such visualized model is shown in Figure 3.11. Atomic models at the same nodes are colored in the same color. Lines between atomic models indicate connections between them, optionally annotated with the ports that are being connected. These lines are connected directly, mainly to create simpler *dot* files for rendering with GraphViz.

A coupled model is also visualized and is mainly used to have GraphViz group atomic models.

Rationale

As with most DEVS simulators that are based on programming languages, PythonPDEVS supports the construction of models and connections using all programming constructs that are available. It is thus possible to create a model using e.g. recursion, loop constructs, reference passing, . . . The problem is that the only feedback about whether or not a connection is made, is in the simulation trace. This makes it impossible for the user to actually check whether or not the simulator has created the desired link without tedious trace inspection. A small oversight in programming, for example in the loop termination condition, could cause two models not to be connected, causing (seemingly) incorrect simulation traces.

By adding visualization of the model, it becomes possible to actually *see* the model that is being simulated, or at least the connection structure of models and their connections. Furthermore, the allocation of an atomic model in distributed simulation can have a huge impact on simulation performance, even though it is easy to make a small mistake here, just as in ordinary coding.

Visualization of the model thus offers an interesting tool to the user, though it is mainly important for debugging the model. While this is not really a usability feature in itself, as the simulator doesn't become easier to use, it provides an easier way of debugging model structure.

Implementation

An implementation is straightforward, as we simply have to write out the model in a textual format that is readable by GraphViz. We should note though, that there is some slight interference with direct connection: GraphViz does not support real 'port-based' graphs, so we should draw direct lines between nodes as is done by direct connection. However, the Coupled models should still be visualized for debugging reasons (and to force grouping in GraphViz). For this reason, part of the writing happens before direct connection and another part happens after.

[5]GraphViz: `http://www.graphviz.org/`

3.14 Documentation

Sphinx [6] generated documentation is included in the PythonPDEVS distribution, in the *doc* folder. Most of the features are really simple to use, so they don't require a lot of explanation. Nonetheless, every feature is explained and an example is presented.

The documentation also includes a list of all configuration methods that can be called and all possibilities for it. For example, all different state savers are listed, together with their effect, advantages, disadvantages, ... The documentation is mainly focused on the technical aspect of the simulation and how it can be used.

A list of common problems (and solutions) is also provided, to make debugging a lot simpler if the error seems to happen in PythonPDEVS itself. Additionally, all internal documentation is also included to make future contributions to PythonPDEVS a lot simpler.

Rationale

Having a lot of these features is nice, though no one will know that they exist, let alone use them, if they aren't documented thoroughly. Since PythonPDEVS is completely textual, there is a slightly higher learning curve then if it were GUI. To offer the user all possibilities, good documentation of the API is vital.

Due to the modular design of the PythonPDEVS kernel, lots of components can be written by the user manually and simply plugged-in. Examples of such components are the *Tracer*, *Scheduler*, *Relocator*, ... Writing one manually would be nearly impossible if their structure and interface wasn't explained in detail.

Since Python is an interpreted, dynamically typed language, no kind of compile-time checking is possible. A user might return nothing in the transition functions, whereas the kernel implicitly assumes that a state will be passed. The same holds for many other functions, such as the output function that assumes a dictionary with ports as keys and lists of messages as values. Performing all kinds of type checking at run-time would be possible too, though it would impose a huge overhead.

We choose to make the assumption that the user knows what he is doing, but we have also tested what would happen if this assumption is invalid. All exceptions that were thrown in that way were noted and were compiled in a list of *Common problems*. Because the exception trace states that the problem occurred in the simulator code itself, the user might think that this is a bug in the simulator. Most of the time however, this is due to one of these assumptions not being valid. The user can then look up the exception in this section and see whether or not it is listed there and solves the problem.

3.15 External contributions

In this section, we give a small overview of two projects that use Python(P)DEVS. Both of the projects aim at providing a modelling environment, combined with a user interface for the actual kernel. Clearly, if our kernel gets better performance, then these projects will automatically get this increased performance too.

3.15.1 DEVSimPy

DEVSimPy is an advanced modelling and simulation framework, which uses PythonPDEVS as its simulation kernel. This project is called *DEVSimPy* and is introduced in [11].

DEVSimPy includes an IDE for PythonPDEVS, which allows the user to create atomic models and couple them using the mouse. Of course, if the user wants to write the atomic DEVS models, he is still forced to manually write Python code.

Because DEVSimPy uses PythonPDEVS as its simulation kernel, it will automatically profit from all our performance enhancements and tweaks. This way, the DEVSimPy project gets all our functionality too, such as distributed simulation, Classic/Parallel DEVS simulation, ...

It is not a one-way relation though, as DEVSimPy causes the PythonPDEVS kernel to become more popular because it becomes a lot simpler to use. Furthermore, our documentation will not be that important anymore to the average user, as the GUI can simply be explored graphically. Additionally, all relevant information about these options can be provided within the GUI itself.

A screenshot of the current version of DEVSimPy is shown in Figure 3.12, where a *policeman* atomic model is connected to a *traffic light* atomic model.

Other DEVS simulators often contain a GUI or IDE too, such as CD++[10] and MS4Me[62]. A GUI interface to the simulation is provided by DEVS-Suite[34] and vle[57] additionally allows the creation of new couplings.

3.15.2 $AToM^3$

A formalism for DEVS that is written in $AToM^3$ is presented in [64]. Even though it is slightly outdated and does no longer work with the latest version of PythonPDEVS, the concept remains valid and worth mentioning here.

[6] Sphinx: `http://sphinx-doc.org`

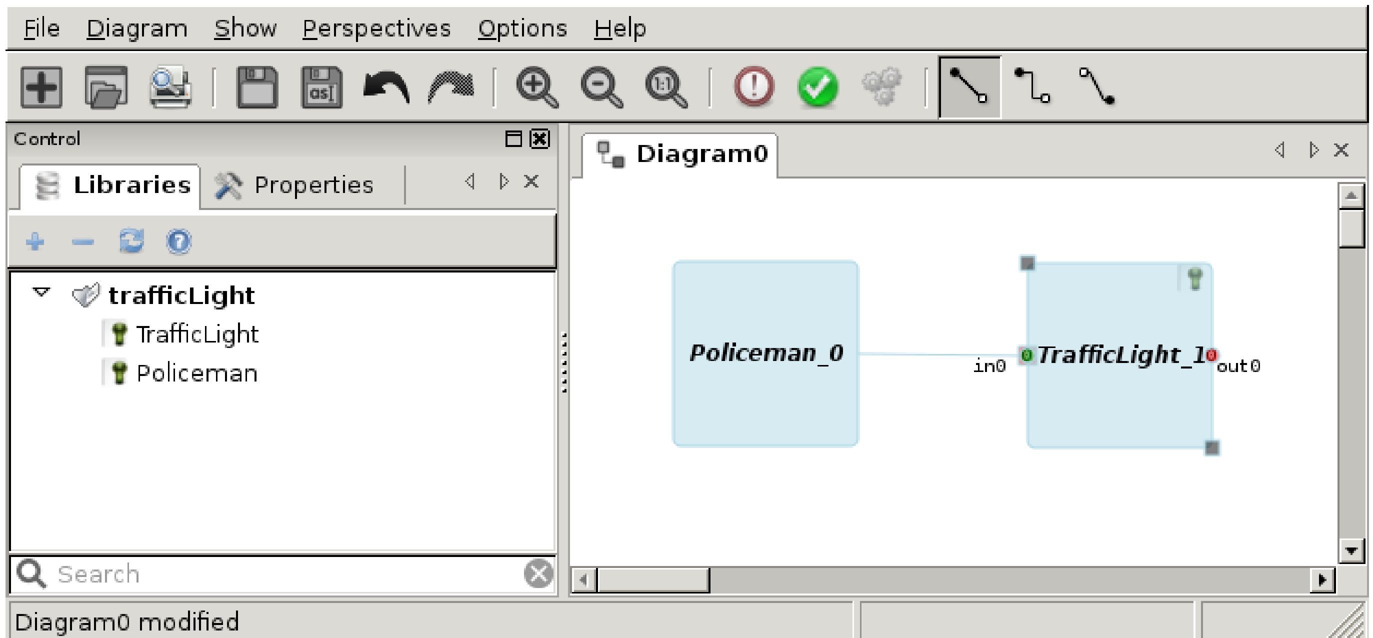

Figure 3.12: DEVSimPy

In contrast to DEVSimPy, which offers graphical tools for the construction of coupled DEVS models only, the *AToM*^{3} formalism offers this for atomic DEVS models too. The visual representation is similar to finite state machines, with states having an internal and external transition arc. The transition functions can be altered by the user by writing Modelica[19] code. An example is shown in Figure 3.13.

When the model is created, it will be written completely in Modelica, which is independent of Python and thus of PythonPDEVS. There were several reasons for using Modelica as an intermediate language instead of directly writing to Python. Related to this thesis, the most important reasons were (a full list is presented in [64]):

1. Modelica can be used for symbolic flattening of the actual DEVS model[12]. Symbolic flattening refers to actually applying the *closure-under-coupling* proof presented in Section 1.7.1 on a specific model. It will thus result in a single atomic DEVS model, which will have the same simulation trace as the original coupled model.

2. By using an independent language, it is possible to write out the code to other DEVS simulators too, such as adevs[54], vle[57], . . . The compilers to the specific simulator can exploit simulator specific options and achieve good performance if possible. Currently, writing exactly the same model in different simulators is hard, due to all simulators using slightly different interfaces, offering other options, using different implementation languages, . . . The simulator-specific compiler could be written by experts for that specific simulator to achieve the highest performance.

3.16 Feature matrix

Due to the variety of options that is possible, a feature matrix is shown in table 3.1 that distinguishes between the three kinds of simulation: *local*, *distributed* and *realtime* simulation. Even though realtime simulation is basically a sequential simulation, it is mentioned seperately because some features are not compatible with the realtime requirements.

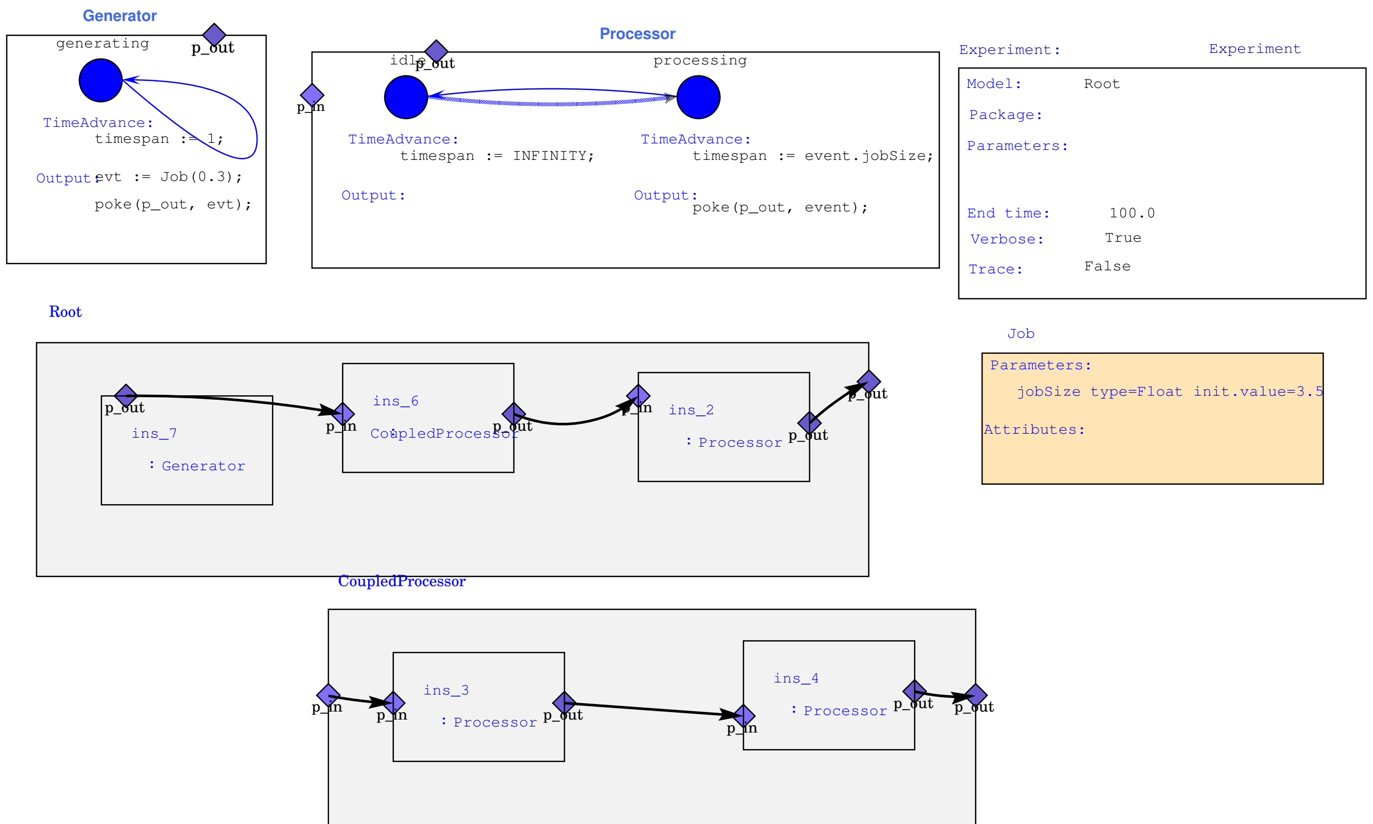

Figure 3.13: Example model in *AToM*^{3} using the DEVS formalism

Feature	local	distributed	realtime
classic/parallel	OK	OK	OK
tracing	OK	OK	OK
dynamic structure	OK	NO	OK
checkpointing	OK	OK	NO
nested simulation	OK	OK	OK
termination condition	OK	OK	OK
continuation	OK	OK	NO
reinitialization	OK	OK	NO
random numbers	OK	OK	OK
halted simulation	OK	OK	OK
transfer function	OK	OK	OK

Table 3.1: Feature matrix

4 Performance

In this chapter, all significant performance enhancements used by PythonPDEVS are presented. Efficiency should be interpreted relatively, as PythonPDEVS uses Python, which is a generally slow compared to compiled languages such as C++. Nonetheless, many of these optimizations are applicable in compiled languages too. As such, there is potential for these optimizations in other DEVS simulators too.

The techniques are grouped in 3 categories: local optimizations, distributed optimizations and domain-specific optimizations. At the end, both synthetic and *realistic* models are used as a benchmark to illustrate the efficiency that can be achieved.

Consistent with previous chapters, each optimization is split up in in an introduction, a rationale and some notes on the implementation. We will not go into empirical results of every modification, as some of them are tightly woven into the kernel, or have a significant impact on the design of the simulator. This makes it difficult to show a comparison between using the optimization and not using the optimization.

4.1 Methodology

All simulations were performed on a 30-node shared cluster of *Intel Core2 6700 dual-core @ 2.66GHz* machines with *8GB* main memory, running *Fedora Core 13, Linux kernel 2.6.34*. The number of nodes in use is always indicated. These machines have *CPython 2.7.5* and use *MPICH 3.0.4* as MPI implementation with the *MPI4py 1.3* wrapper. Local simulations only use a single core of a single machine.

All *PyPy* and *adevs* benchmarks were run on an *Intel i5-2500 quad-core @ 3.3GHz* with *4GB* main memory, running *Gentoo, Linux kernel 3.12.14*. All other software was at the same version.

Results are averages of 5 simulation runs. Due to the cluster being shared, excessive outliers were removed from the results. We acknowledge that this is not ideal, as these outliers might signal problems in the simulation algorithm that remain undetected in this way.

4.2 Local simulation

First we will discuss the local (non-distributed and non-parallel) simulation algorithm and its performance enhancements in comparison to the abstract simulator presented in Section 1.7.1. These local optimizations also have a significant impact on distributed simulation performance, as they basically use the same simulation algorithm. All optimizations to local simulation will therefore also be inherited by distributed simulation.

Local simulation is the only kind of simulation that can be objectively compared to other simulators, as distributed simulation is very dependent on other details such as middleware, shared memory assumptions, . . . Due to the close link between local and distributed simulation, the local optimizations often contains some remarks on how this works in distributed simulation.

Parallel DEVS is also discussed in this section, though this is using a sequentialized algorithm.

4.2.1 Simulation algorithm

Our simulation algorithm is very different from the *abstract simulator*, both for Classic DEVS[81], Parallel DEVS[17] and Dynamic Structure DEVS[7]. The most important difference is our replacement of *message passing* with method calls. Due to the synergy with *direct connection* (Section 4.2.2), all code for coupled models can also be severely simplified.

We will not present our simulation algorithm in great detail, as it is slightly obfuscated due to support for distributed simulation too. More information about the algorithm can be found in Section 2.2.1.

Due to direct connection being used, the algorithms can make the assumption that a message will be delivered after exactly one step. This makes it possible to drastically reduce the coupled model algorithms, as the only coupled model will be the root model. For example, a coupled model will never receive external input, nor generate external output.

Rationale

Whereas the algorithms presented in the abstract simulators are very readable, an implementation of these algorithms is likely to be very inefficient. For the DEVS abstract simulators, the prominent inefficiency lies in the use of messages to direct all communication. Even though the order in which messages are sent is completely fixed (output creation messages, then transition messages, . . .), the abstract simulators do not take advantage of this. Such messages need to be created, send to the destination, possibly forwarded, . . . As soon as the message arrives at its destination, its type has to be checked, resulting in a big conditional. This is inefficient and impairs modular design as the complete algorithm is represented in a single method.

We also deviate from the parallelism part of the Parallel DEVS abstract simulator for technical reasons, as was also suggested in [26, 25]. Many Python interpreters have a Global Interpreter Lock (GIL), which effectively disables all possibilities for parallelism between threads (except for I/O blocked threads). The only solution is to start different subprocesses, clearly having an even bigger overhead than simple threads. Thus it was decided that the overhead was not worth the slight performance boost in general situations. As we also support distributed simulation, a system with multiple cores could use this functionality to use additional cores instead.

Implementation

Our implementation is fairly straightforward and the core algorithm was already presented in Algorithm 3.1. Not all parts of the implementation are shown, such as the implementation of the atomic DEVS and routing code. The actual code has lots of additions for e.g. tracers and memoization. All message passing was replaced by equivalent, direct method calls, which additionally increased readability of the source code for people that were unfamiliar with the terminology of the abstract simulator. For example, the passing of $*$ messages is replaced by a `outputGeneration` method.

One difficulty in the implementation is that all different DEVS formalisms should be possible with a single algorithm, which has the potential for lots of conditionals. Python doesn't have macros or preprocessing directives, meaning that all these checks should happen at run-time even if they are statically determined. Also, a distributed simulation requires a lot of locking of several parts of the simulator, which is obviously unnecessary in sequential simulation. Since PythonPDEVS requires both, the distributed simulation algorithm is also used for sequential simulation, which causes locking overhead in sequential simulation too.

Additionally, Dynamic Structure DEVS requires checks for *model transitions*, which have a huge impact on performance. As Dynamic Structure DEVS is not used that frequently, we have decided to skip this model transition phase by default. The same is true for other functionality that increases the simulation overhead: they are disabled by default and have to be enabled manually. Enabling Dynamic Structure DEVS through the configuration simply toggles this check on.

4.2.2 Direct connection

Direct connection was presented in [12] as a first step to symbolic flattening. All connections are resolved at simulator initialization time, omitting the intermediate coupled models. As these models no longer have a routing function, they effectively become useless and can be removed from the simulation. The only coupled model that remains is the root node. Consequently, all messages will be transferred between atomic models directly, without passing through a series coupled models. The hierarchy will become (almost) flat, as is shown in Figure 4.1.

It requires some intensive pre-processing of the model and has a severe impact on all parts of the simulation algorithm, specifically those that handle distributed simulation. Even though it was already implemented in previous versions, direct connection is now able to cope with dynamic structure, transfer functions, relocations, . . . and handles distribution a lot better than our previous algorithm.

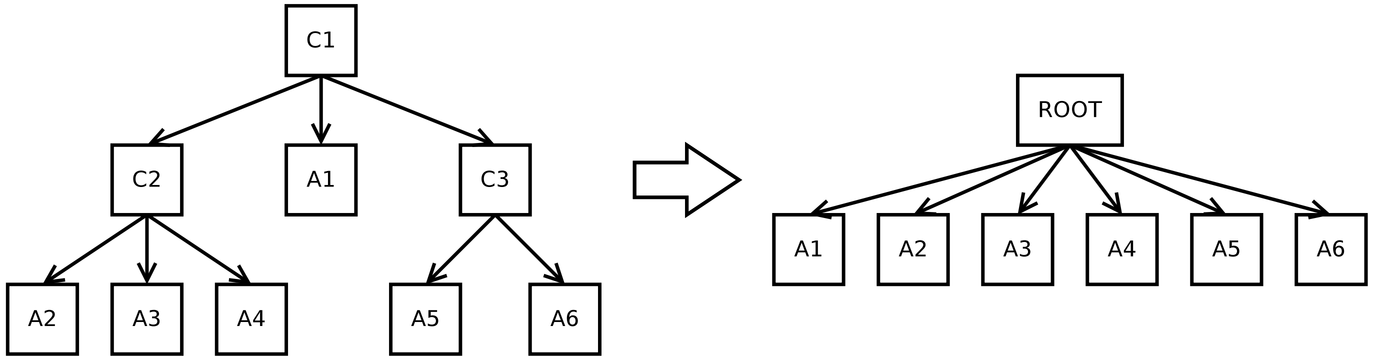

Figure 4.1: Direct connection flattens the hierarchy

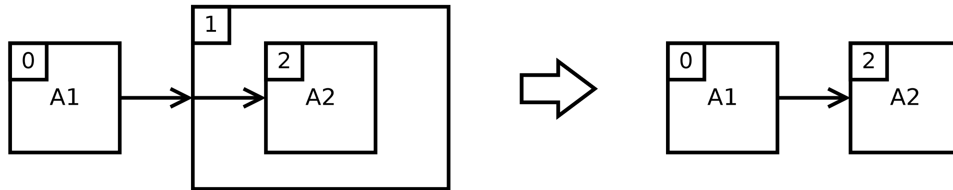

Figure 4.2: Direct connection and its impact on distributed simulation

Rationale

Message passing in coupled models is one of the major simulation overheads in deep models[12, 37, 45]. A coupled model only has a *structural* role in the simulation, meaning that all time spent there is basically routing overhead. Most models are static though, so there is no need to resolve it every simulation step. Even dynamic structure simulations can be thought of as having two phases: simulation of a static model followed by (infrequent) structural changes. In this case, it is required to undo the direct connection before the changes and perform direct connection again afterwards.

Even though it clearly removes the overhead in deep simulations, it has an effect on the scheduler as well. Normally, a scheduler per coupled model would be present, which takes the element that transitions first and passes it to the higher level. However, there is no notion of *levels* anymore, meaning that the root model will contain a scheduler that contains all models simultaneously. This could be either an advantage or a disadvantage, depending on the scheduler. Clearly, it requires a lot less schedulers, saving on space and on complexity. On the other hand, this can have a huge effect on the scheduler's performance as the complexity is defined by how many models are in the scheduler. While naive schedulers might perform worse due to more entries being present, smarter schedulers will be able to exploit this and use this *global* information of the model to their advantage. So even though synthetic models can be built where direct connection might perform slightly worse, most models will have a serious performance boost. Additionally, our code gets a lot more elegant and simple, which will allow for a slight performance increase at other parts of the code too.

The impact in Classic DEVS simulation is slightly reduced if collisions happen, as the coupled models were also responsible for the *select* function. Our direct connection algorithm has to cope with this functionality. In the worst case, the *select* function will have to be called on every coupled model of the hierarchy.

Distributed simulation even requires direct connection to make the anti-message vs. message *race* more efficient, which is why it has become a mandatory optimization. Yet another advantage in distributed simulation is that otherwise we might have to pass through different nodes before we actually reach our destination. An example of this is shown in Figure 4.2. The models are annotated with their location in the distributed simulation. Normally, the message would have been sent to node 1, which immediately forwards it to node 2. Due to direct connection however, the message can be sent to node 2 immediately. This avoids one network message and makes sure that node 1 doesn't have to be rolled back unnecessarily.

Other simulators, such as adevs, use similar approaches, as mentioned in [45].

Implementation

Implementing direct connection is not that difficult, as it merely follows all links at the start of the simulation and creates a new (direct) connection afterwards. It is complicated due to support for transfer functions too, which will be connected through the use of lambda functions.

When using Dynamic Structure DEVS, modifications to the model will mark the model as *dirty*, which causes a new direct

Scheduler	Average case	Worst case
Sorted list	$O(n \cdot log(n))$	$O(n \cdot log(n))$
Minimal list	$O(n)$	$O(n)$
Heap	$O(log(n))$	$O(n \cdot log(n))$
Heapset	$O(log(a))$	$O(n \cdot log(n))$

Table 4.1: Overal complexity of the scheduler

connection pass to happen at the end of the structural changes phase. Should no modifications happen to the model, no direct connection happens. Note that the complete model is recomputed and not only all paths that are related to the altered part. This is because direct connection does not have that big of an overhead and it is sometimes difficult to determine the actual impact on direct connection.

The Classic DEVS simulation algorithm needs to be modified as a result of direct connection. As all coupled models are removed from the simulation, no *select* function is called anymore. For this, the algorithm will first find all models that are colliding (remember that this will now be a list of *atomic models* at *arbitrary depth* and not a mix of *coupled* and *atomic models* at the *same depth*). If multiple models, thus a collision, are found, the hierarchy is iterated top-down as usual and the *select* function is called every time. Some preprocessing is also required, as the *select* function takes its direct children as a parameter, instead of the colliding model list, which contains children at arbitrary depth. The *select* function returns one element, which is used to filter the remaining models: only those that have this model in their hierarchy will be kept. This process resumes until a single atomic model remains, which is the model that will transition.

Direct connection in a distributed setting could be a lot harder if the model would be scattered over the nodes, meaning that not a single node has the complete model. Due to our support for relocation and determinism, we have forced every node to have a complete version of the model (except for states). Direct connection will only have to happen at the controller, after which the direct connected version is broadcast.

4.2.3 Scheduler

The complete simulation kernel is designed to be as modular as possible, which allows the scheduler to be modular too. Thanks to this, there is the possibility to create several *plug-and-play* schedulers from which the user can choose.

The following general-use schedulers are included by default:

1. The **Sorted List** scheduler maintains an eventlist that is kept sorted at all times. All operations simply update the list and then perform the sort operation on it. Due to Python using the *timsort* algorithm (which is also implemented in C), performance is rather good. This is the scheduler used in many simple simulators.

2. The **Minimal List** scheduler is similar, but it doesn't sort the eventlist. For every access to the scheduler, the complete list is iterated to find the required answer.

3. The **Activity Heap** scheduler is based on a heap, but it only contains elements that have a time advance different from infinity. Due to a heap not supporting random access, entries are simply invalidated instead of removed. When too many entries become invalid, the heap is restructured by removing all invalid elements.

4. The **Dirty Heap** scheduler is identical to the activity heap, though it does not clean up the heap if it becomes too dirty. This avoids the bookkeeping of the invalid elements and consequent restructuring of the heap.

5. The **Heapset** scheduler uses a heap too, though it only contains the absolute time values. In addition to the heap, it also contains a hash map, which provides a mapping between an absolute time and the models that transition at that time. Finding the first model requires us to search for the lowest time in the heap, which is used to access the hash map. Collisions only increase the size of the list in the hash map, which is an $O(1)$ operation. It is basically a combination of the heap and list methods, though leans more to the heap method.

An overview of their complexity is shown in Table 4.1. Note that this only shows a general overview, as the complexity of the different scheduler operations also vary drastically.

This is only part of our scheduler-related optimizations, of which the remainder is presented in sections 4.4.4 and 5.2.1.

Rationale

The scheduler is a vital part of a DEVS simulator. Performance-wise it might even be the most vital part. It is nearly always the reason for the complexity of a specific simulation, as it should be the only component that takes into account all models. For this reason, most DEVS simulators will have a built-in scheduler that is specifically optimized for its main problem domain.

Two different approaches are the most frequently used though:

- **heap-based** algorithms will use a heap as data structure, sorted on the absolute time of the model. A schedule operation adds the model to the heap, whereas an unschedule operation removes the model from the heap. When the next time is required, we simply pop the first element from the heap and we automatically have both the time and the model itself. Scheduling has a complexity of $O(k \cdot log(n))$, with k the number of scheduled models and n the number of total models in the simulation (generally $k << n$). Finding the first element has a complexity of $O(k \cdot log(n))$ too, as the first element(s) have to be popped from the heap.
- **list-based** algorithms keep a list of models as their main data structure. Scheduling or unscheduling models doesn't require any change, as the list contains references, so complexity is $O(1)$. Finding the first element requires a complete iteration over the list though, with complexity of $O(n)$. Note that this implementation is a lot easier to implement than the *heap-based* solutions.

Each of these schedulers has its own advantages and disadvantages, but the most obvious difference would be the difference in complexity. Clearly, if k is small, the *heap-based* schedulers are ideal. With a big k (possibly even related to n), the *list-based* schedulers start getting ideal due to their independence of k.

Generally, the simulator has no idea about the value for k and therefore we do not force such assumptions on the user. Even further optimizations to the scheduler are possible (requiring even more domain-knowledge), though these are mentioned in Section 4.4.4. It also allows us to compare schedulers without having to make major modifications to the simulation algorithm.

In previous work[72], we found that the *heap-based* approach is used by (among others) *adevs*[54] and *vle*[57]. The *list-based* approach is used by *CD++*[79], *X-S-Y*[29] and first versions of *PythonDEVS*[9]. These simulators implicitly make an assumption on the value of k, which is not even communicated to the user. If this assumption is correct, their simulator will be efficient, otherwise it will have slow performance.

Implementation

To allow for a modular scheduler, all scheduling operations by the simulation kernel should happen through a well-defined interface. Some restructuring was necessary to the scheduler itself (to receive requests in that way) and to the kernel (to put all of its requests in the desired form).

All calls are as general as possible to offer more flexibility to the scheduler itself. An example is the *reschedule* method, which profiling showed to be the most time consuming in most schedulers. As these calls are made right after the other, it might be more efficient to put a complete request in a single method invocation instead of invoking fine-grained methods on the scheduler.

Several modular schedulers were also required, to clearly show the difference. Writing a new scheduler was not that difficult as our interface is relatively limited: *schedule*, *unschedule*, *reschedule*, *readFirst* and *getImminent*.

4.2.4 Quantized DEVS

Quantized DEVS[81] is a variant of DEVS that will not send output messages if the difference between the new message and the previous message do not exceed a certain threshold. It is often used in Cell DEVS simulation to enhance simulation performance, as in [49, 51].

An example of the use of Quantized DEVS is shown in Figure 4.3. Here, the quantum is defined as 0.25, meaning that a message with value x will not be sent if it is in the range $[y-0.25, y+0.25]$, with y the previously sent value. The leftmost model is one that reduces its internal state with 1 at every simulation step and outputs the new state. The two other models take the received value, divide it by 10 and take this as their new state and output it again. At simulation time 0, all previous messages are initialized as 0, so all messages are being sent. However, the last model should output the value 0.1, which is in the range $[0-0.25, 0+0.25]$ and thus no output happens. As this model is not connected to any model, this does not introduce an error.

In the next simulation step, the first model changes its state to 9, which is outside of the range $[10-0.25, 10+0.25]$. Therefore, the message is sent, and the previous message variable is set to 9. The middle model however, gets new state 0.9, which is within the range $[1-0.25, 1+0.25]$. Consequently, no output message is sent and the previous message variable is kept at 1. Without Quantized DEVS, the model would have outputted the 0.9 and the last model would have altered its state to 0.09. Now we clearly deviated from the normal simulation behavior, because the last model didn't receive the update. However, we have saved one message and a single external transition. This continues and finally at simulation time 3, the message of the middle model is sent again and the previous message variable is updated. In this situation, the error is eventually stabilized, though this is generally not the case.

Some simulators support this by default, such as CD++, though PythonPDEVS does not support this for several reasons:

1. Quantized DEVS introduces an error in the simulated results. This is therefore not a real optimization, as it also alters the simulation results. Simulation should be performed with the threshold as small as possible, while still increasing simulation speed. The introduced error is analyzed in e.g. [51, 46].

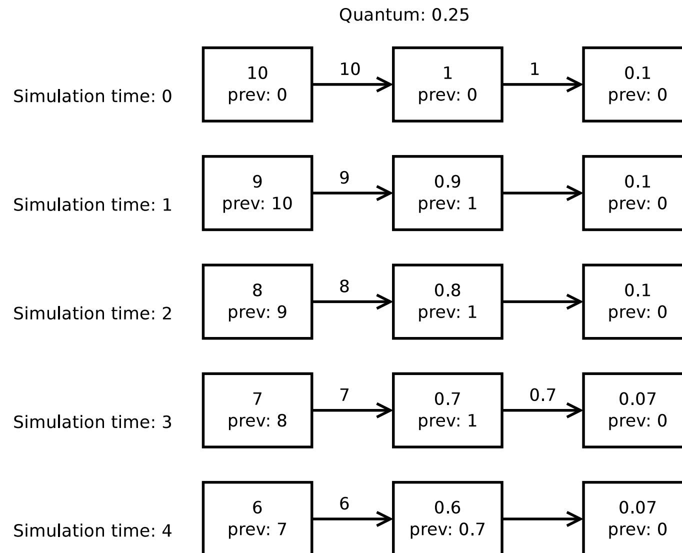

Figure 4.3: Example of Quantized DEVS

2. It is unknown to the simulator how to compare the messages and what this threshold should be. Advanced versions could even have a variable threshold. CD++ does not have this problem, as it only allows single floats to be sent. PythonPDEVS supports arbitrary data(even complete objects), so it is unknown how these should be accessed.

3. There would be a slight overhead for all situations, even if no quantization is used.

Rationale

There are several important advantages to Quantized DEVS, most of them related to distributed simulation:

1. Since less messages are passed, their external transition function is not called;

2. In distributed simulation, every message is a potential straggler, so the number of stragglers should also decrease;

3. In distributed simulation, message passing between two nodes is relatively expensive due to network communication;

4. Models might stay inactive for longer, which can provide additional speedups when combined with activity.

Implementation

Even though Quantized DEVS is not supported at the kernel level, it is possible to write this kind of behavior in the functions of the model. Since this requires manual intervention from the user, we can be certain that the user will be aware of the possible error that is introduced. Actually, one of our benchmarks at the end of this chapter uses Quantized DEVS to better show the effect of our optimizations.

4.3 Distributed simulation

In order to keep models synchronized, some additional code is necessary. This additional code offers some more options for optimization. Our goal in distributed simulation is also slightly different: the simulation algorithms themselves are already rather efficient due to the optimizations for local simulation, so our main focus should go to reducing the amount of network messages.

4.3.1 Allocation

PythonPDEVS requires the user to manually provide the allocation together with the `addSubModel` method. For example, `self.addSubModel(MyModel(), 2)` will place the model created by `MyModel()` at node 2. Contrary to many other simulators, the granularity of allocation is an atomic model instead of a complete subtree. As a result, unrelated atomic models can be assigned to the same node, or a single node can have a number of subtrees.

Main disadvantage of this approach however, is that the location of all nodes need to be specified in the code itself. For simple models, this is unlikely to be a problem, but for more complex models, the allocation strategy might be a lot more involved. One solution to this problem is to use the full functionality of Python and use variables, additional functions, . . . to determine the

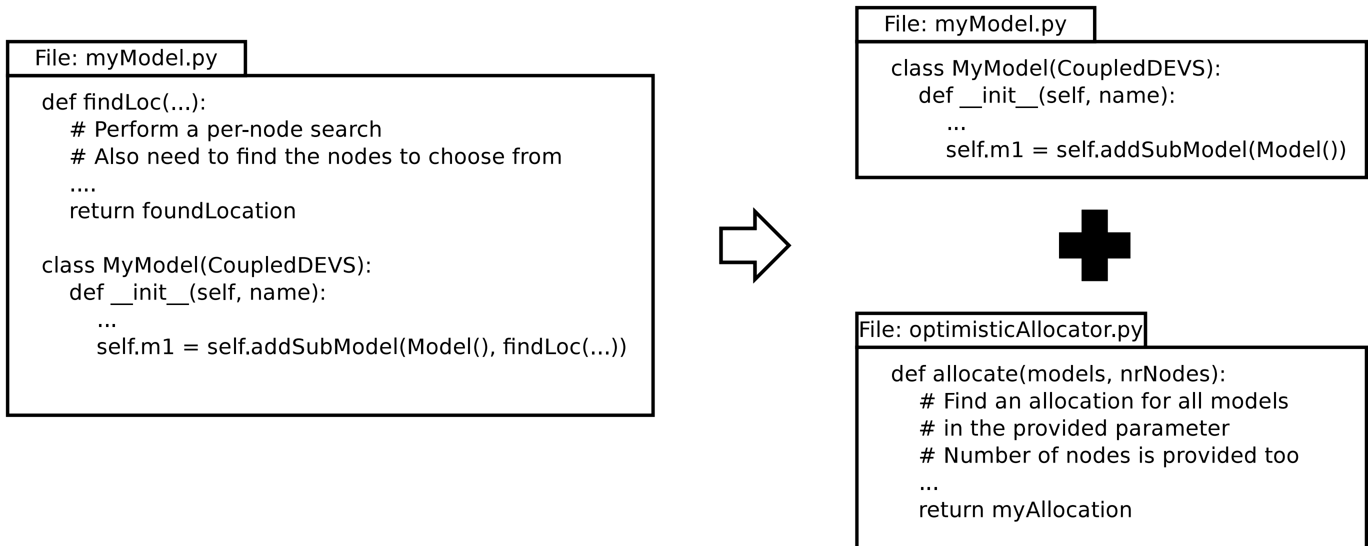

Figure 4.4: Allocators allow more elegant code

location of every submodel in the model initialization code. While it solves the allocation problem, it is very inelegant: the code that specifies the model also has to be involved with the allocation code and more specifically, it needs to know about the details of distributed simulation, how many nodes are used, . . .

Our solution is to introduce a separate component: the allocator. During model construction, the location at which models are simulated can be ignored. At the end, a function is called that performs the allocation using this global knowledge. The function can then really specialize itself for the synchronization protocol and the actual code for the model can stay exactly as it was in sequential simulation.

For performance reasons, the allocation that was found using the allocator will even be saved. As finding a good enough allocation might take an arbitrary amount of time, this will speed up subsequent simulations. Should the allocation code be put directly in the model code, the user would have to write such functionality manually too. The saved files are easily modifiable by hand, as it uses CSV-style notation. To prevent inconsistent files, a small check is also done at the start of every allocation run. Specifically, it is checked whether the names are identical and the requested node actually exists.

In chapter 5, we will further extend the allocator to give even bigger performance improvements. For future reference, we will call these allocators *static allocators*, whereas *dynamic allocators* are introduced in Section 5.2.3.

Rationale

The main use of an allocator is not performance, but because it allows for separation of concerns: the code for the model determines the behavior of the model and the allocator determines the distribution of the model. If sequential simulation is done, the allocator is simply not called and the model will be ran sequential automatically without any involvement of the user, thus offering *true* transparency for distribution in the model. An example is shown in Figure 4.4, where originally the allocation code was mixed in with the model code. By writing an allocator, the model is as elegant as in local simulation and all distribution-specific code is split up. It would also be possible to write multiple allocators to choose from (e.g. `optimisticAllocator`, `conservativeAllocator`, . . .), making it possible to test multiple allocations or switch to different simulation algorithms (should this be supported by PythonPDEVS).

Even though this is mainly to offer more flexibility to the user, it is considered a performance enhancement due to the possibility for more fine-grained control over the model. Additionally, if PythonPDEVS (or other compliant simulators, for that matter) were to offer other types of distributed simulation (like conservative), the distribution is likely to be different. Without such an allocator, the model would need to be extended if good performance is required in both situations. It either requires additional of conditionals in the code, or it would require a complete copy of the model, creating for example the models `MyModel_Conservative` and `MyModel_Optimistic`. With an allocator, two different allocators are written and are simply combined together with the model. Should the ideal allocation strategy change for some reason, the model can be left alone and only the allocator code should be altered.

Implementation

The implementation is not that difficult [1] : we simply allow the user to configure an allocator, which is called with the required arguments. As soon as an allocation is found, it is saved to file for reuse in future simulations. Should a saved allocation already exist, it will be read first and if it matches, the allocation will be done using this file, otherwise the allocator is called.

To test our implementation we have included a general *auto allocator*, which will use a naive heuristic to perform allocation. To offer even more flexibility to the user, it is possible to combine `addSubModel` allocations with a custom allocator: models that don't have a location assigned to them will have `None` as their location attribute, whereas they otherwise have the allocation saved there. Of course, the allocator is able to modify these allocations if it needs to.

In chapter 5, this concept is extended with support for *dynamic allocations*, which are more involved for the simulator and will require a slightly more complex implementation. For this reason, the allocator code has slightly more methods than were introduced here, though this will be explained later on.

Running example

There is no direct usage of a static allocator in the running example, as it will only find a naive allocation. Listing A.4 shows a simple example for a more general case though, so writing a custom static allocator should be fairly simple.

4.3.2 Relocation

Relocation makes it possible for the user to provide *relocation rules* at the start of the simulation. These rules allow the user to alter the distribution at simulation time should it be necessary.

Rationale

While it is true that the initial distribution of a model is extremely important to good performance, it is possible for the ideal distribution to shift during simulation. Without support for relocation, the initial distribution will be kepth throughout the complete simulation run. Relocations can handle these situations better in the sense that they allow the kernel to migrate atomic models between different simulation nodes.

Implementation

For an overview of how the relocation mechanism works, we refer to Section 2.2.3. Performing a relocation requires a rollback to the (newly set) GVT, as discussed in Section 2.2.3. These rollbacks can have a big impact on simulation performance if the models are too far ahead of the GVT. This is not a real concern for the following reasons:

1. Relocations that are performed with optimization in mind will relocate models from the *slow* nodes to the *fast* nodes. The *slow* nodes are also these that are close to the GVT and thus a rollback is very short. Rolling back the *fast* nodes is not that bad either: they were at a high risk of being rolled back by the slow nodes anyway.

2. Big rollbacks are not that problematic if memoization is used. As this is an artificial rollback, the memoization cache will not be invalidated either. Even if the fast nodes are rolled back significantly, they can rely on their memoized states to quickly go over all rolled back computations.

Some mechanisms are added for the user to provide relocation rules to the kernel. These rules are of the form `<time, model, destination>`, where `time` indicates the time of relocation, `model` indicates the model to be migrated and `destination` is the node that should receive the model. After every GVT computation, all these rules are checked for whether or not their corresponding relocation should be performed.

Due to future activity extensions to the relocation component, this rule checking is written within a more general relocation framework.

4.3.3 Message grouping

Message grouping will bundle multiple messages together before sending them over the network. It will be used at several places:

1. Anti-messages will be sent in a batch instead of one message per time. Normally, every message would have one anti-message that can be used to annihilate the previously sent message. In our implementation, an anti-message will simply not exist, but instead we use an *invalidation method* that takes a complete set of IDs to invalidate. This furthermore removes the overhead associated to having anti-messages look similar to normal messages.

[1] Future extensions in Section 5.2.3 make this a lot more difficult.

2. Messages for the same model will be grouped together. When sending multiple messages to the same model, all these messages will be combined into a single message. This is effectively *message merging*, as after the merge there is no possibility to check which part of the message was from which node.
3. Tracer calls will be made in a single (buffered) request to the controller. Clearly only applicable if a tracer is used.
4. Relocation will have all states bundled in a single call.

Of course, for the GVT algorithm it is still required to send a color together with the (anti-)message calls.

Rationale

It is clear that we want to minimize the number of messages between different nodes as much as possible. We will handle the rationale for each case separately:

1. **Anti-messages**:

   Some relation exists with the modified simulation algorithm (Section 4.2.1), as we also remove the *type* of the message from the message itself.

   By grouping all these messages together, a rollback will be much faster, as it only sends out a few calls, instead of an anti-message for every message that was found. The bad news about this is that anti-messages are no longer *messages* and the routing code does not work for them anymore. Thanks to direct connection, this problem is solved: the final destination is already known before the message is send, so no routing for anti-messages is required. All that remains is thus to invalidate the messages at the node that currently hosts that model.

   As the actual payload of the message is also reduced to simply a set of unique identifiers, the total size of the message is further reduced. Further performance enhancements were also possible, as we can use a faster data structure for the message scheduler right now: previously we had to support the efficient removal of single IDs, whereas we now require efficient removal of a complete set of IDs. This is even implemented by default in Python on the set objects, making it even faster.

2. **Messages**:

   If two messages are destined towards the same model, they will be merged at the model itself, so there would be no good reason to delay this merging. By performing the merge as soon as possible, we additionally only put 1 message on the network, which results in only 1 ID. Should the message ever need to be invalidated, only a single message ID needs to be invalidated. As the message is received at the destination as if it were a single message, it is treated as a single message, meaning that the complexity in the message scheduler and message routing parts are also reduced. The naive method is contrasted to our optimized version in Figure 4.5, where the reduction of 3 network messages (and consequently 3 IDs everywhere) to 1 network message is seen.

3. **Tracer calls** and **relocations**:

   These are practically the same, as they simply reduce the amount of round-trip calls in critical parts of the simulation. A tracer call is made for every model that is transitioning, resulting in a massive amount of overhead if we need to wait for each of these messages to be acknowledged. The same is true (to a lesser extent) for the relocations: if multiple models are relocated to the same node, they can be sent at once. No specific optimizations exist by grouping these together and the only reason is to reduce the amount of round-trip calls. Messages did not require optimizations specifically for round-trip calls, as they are sent asynchronously.

We should note that we shouldn't go to far in message grouping. Both anti-messages and normal messages are grouped by model and not by node, for the simple reason that it might be possible that one of the models might have been relocated by now. If we require such checks we would need to split up the received message, acknowledge part of it and forward the rest. This behavior duplicates the complete set of messages, as we create a temporary (split) copy of the message. Should the model be distributed in a decent way, the amount of inter-node links should be small, so the impact would be extremely small. In cases where the grouping would have an impact, performance is inherently bad already due to the many links.

Implementation

The implementation for the 4 different approaches slightly varies. The tracer and relocation are simply done by putting all arguments in a list and passing the list to a wrapper function that will again unpack it and call the methods separately. Message merging was simple as all parts of the simulator already contained support for it.

Only anti-messages required changes to both the saving of the anti-messages, as to the construction of the invalidation call and the actual processing of this invalidation. Due to the altered design of anti-messages, the message scheduler was also modified to efficiently support the simultaneous invalidation of multiple IDs.

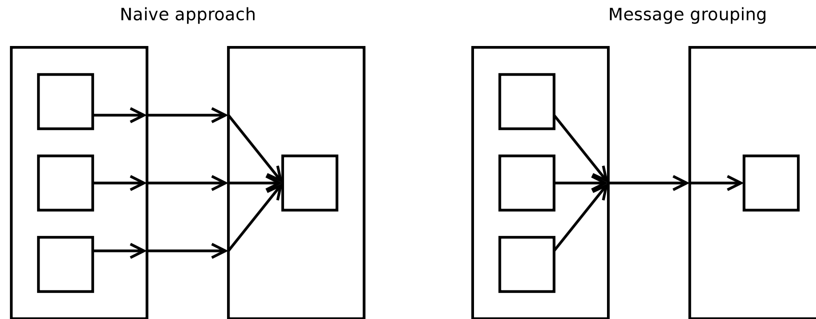

Figure 4.5: Naive messaging vs. message grouping

4.3.4 Per-node scheduler

All nodes are initialized with the *default* scheduler, but it is possible to set a scheduler for a specific node. It would be possible for node 0 to use the *activity heap* scheduler, whereas node 1 uses the *minimal list* scheduler.

This idea would also be applicable in sequential simulation. However, direct connection puts all atomic models in a single model and thus a single scheduler. In distributed simulation however, this doesn't require any additional code apart from configuration and coordination.

Rationale

Section 4.2.3 showed that the choice for the ideal scheduler has an impact on complexity and that there is no clear winner in every domain. In local simulation we are limited to a single scheduler, which will necessarily perform badly if the model consists of multiple parts, all of them having conflicting operation patterns. In distributed simulation, these parts can be divided over multiple nodes such that each node has a single ideal scheduler. The user can then choose the most fitting scheduler at each node, resulting in both parts being simulated as efficiently as possible. The possibility for a performance improvement should be clear, as it is basically an extension of the previous section on selecting the correct scheduler.

Implementation

The implementation is trivial, as there is no link between the scheduler at different nodes and all schedulers use the same interface due to the modularity of our schedulers. All that is required is that the broadcasting of the model includes the scheduler that should be used.

4.3.5 Termination detection algorithm

Several options exist to detect termination in an optimistic, distributed simulation. Most of them rely on the GVT passing over the termination time though. This did not seem like a feasible option to us, as the interval between such GVT computations can be chosen arbitrarily high, which would postpone termination detection.

First, we will explain the basic algorithm that we use and was also used in our previous version. Afterwards, a small optimization is presented that has a massive impact on performance in many situations.

Basic algorithm

Each node will simply start its local simulation loop, which takes into account the remote messages. This algorithm will stop as soon as the termination condition or time is reached at this node and thus the loop will break. The node notifies the controller that it has finished simulation and is waiting for termination. Should the node receive a remote message in the meantime, it will again notify the controller that we are no longer idle. It thus *pushes* its state (*waiting* or *working*) to the controller.

At the controller, we will wait until all nodes have notified that they have finished simulation. As soon as this happens, we start a ring algorithm to make sure that all nodes are still inactive. This additional algorithm is required because it is possible that one of these notifications is transient. A first pass is performed, in which the total number of messages in the complete simulation is computed. Should one of the nodes mention that it became active by now, the ring algorithm will stop early and signal back to the controller that simulation should not yet terminate.

If all nodes state that they are idle, the algorithm will move on to the second phase. This is required as it is possible that one of the nodes got a message while it has already notified termination in the first phase. The second phase will do exactly the same as the first phase: gather the total number of messages in the simulation and stop earlier if one of the nodes is running. At the end of the second phase, the controller now has 2 values: the number of messages found in the first phase and the number of

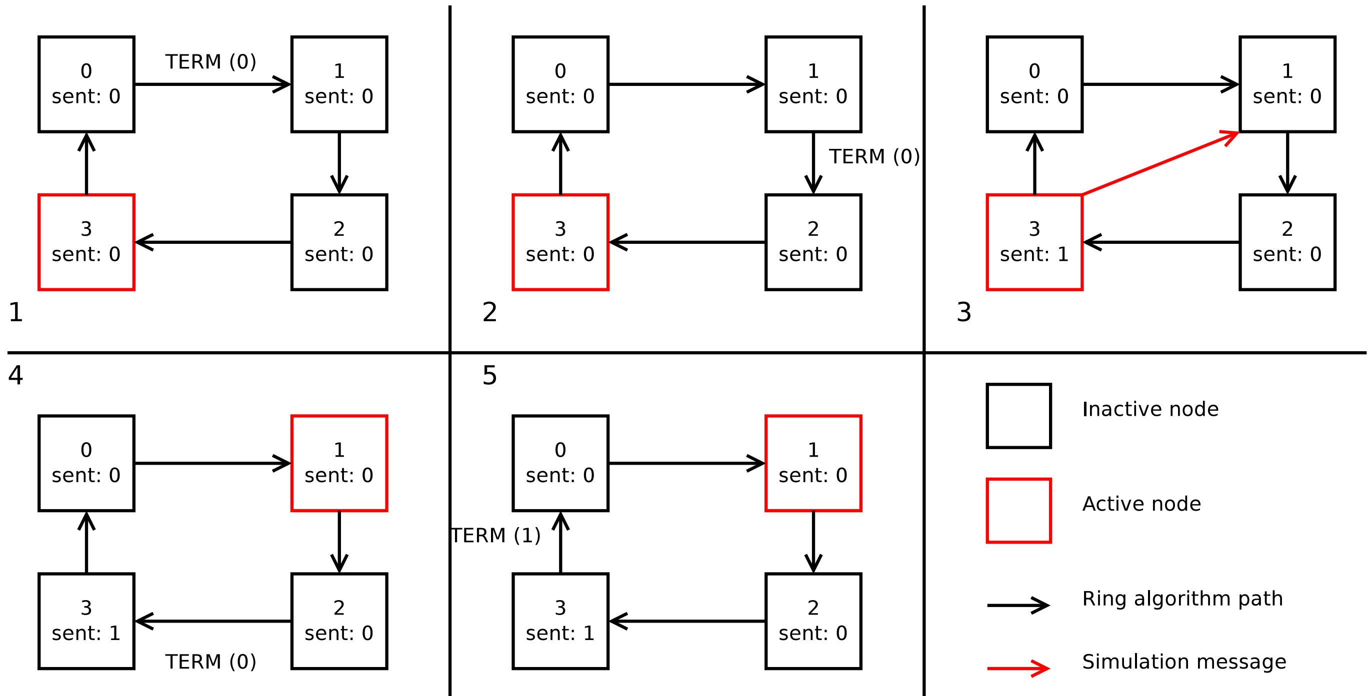

Figure 4.6: Problematic situation in termination algorithm if no second phase is used

messages found in the second phase. If these numbers are equal, it means that all nodes were finished *and* that there was no message passing in between the different phases. As the only way to become active after a termination is to receive a remote message, simulation is effectively terminated.

The need for a second pass is shown in Figure 4.6. One of the nodes is still active when the algorithm starts, which is possible due to the notification still being underway. Right before the termination detection message reaches the active node however, it stops its simulation but first sends a message to a node that has already had the algorithm pass over. The termination algorithm would find nothing suspicious, as node 3 is inactive in step 5, so the controller was notified that all nodes are inactive.

But as can be seen, node 1 is still active from the message that was sent in step 3. Should no second pass happen, the controller would thus incorrectly signal the termination of the algorithm. Three different situations can now occur:

1. Node 1 becomes **inactive** right before the second phase message has arrived and **doesn't send** a message.

   In this case, the termination algorithm will simply pass and conclude that simulation is finished. This is correct, since a node cannot become active on itself.

2. Node 1 becomes **inactive** right before the second phase message has arrived and **does send** a message.

   Now the termination algorithm will run over the ring again. The termination algorithm will detect that something has happened due to the number of sent messages that has increased, even though simulation should be terminated. Consequently, the termination algorithm detects that simulation is still ongoing.

3. Node 1 is still **active** when the second phase message is received.

   The termination algorithm now detects that one of the nodes is still active. The algorithm detects that simulation is still ongoing.

Enhancements

From the previous algorithm, it is clear that the major performance penalty is paid by notifying the controller of our state as soon as it changes. As it happens every time the simulation loop stops, this is problematic if the node is a purely *reactive* node, meaning that it will only perform some actions after it has received an external message. As soon as the message is processed, a *reactive* node would stop its simulation loop and signal the controller that it is finished.

However, it is likely that another message is received slightly later, for which the simulation loop should start again. Our idle notification might have only been just send and we are already notifying that it should be ignored. If this happens at a fast enough pace and at several nodes, the network is flooded by such notifications and the controller becomes a bottleneck. As the

network is flooded, all normal simulation messages are delayed as well, causing very slow processing of simulation messages and consequently slower simulation in general.

Our solution consists of skipping this notification step entirely and simply starting the algorithm every second. This delay should not be too low, as the network would then be flooded by this termination algorithm again. It shouldn't be too high either, as a lot of idling would happen when the simulation is already finished before actually returning. Main disadvantage is that the simulation will again work with fixed times to detect termination.

Starting the algorithm too often is not a performance problem, as the algorithm will immediately detect that simulation is still running. It will quickly return if this is the case, incurring nearly no overhead.

This optimization has a noticeable effect on the performance of, among others, the PHOLD benchmark (Section 4.5.3).

4.4 Domain information

Previous work, such as [72, 73], showed that the availability of domain-information can drastically speed up simulation. One of these examples in [72] was the use of custom message copying, in [73] we found that custom state saving has a similar effect. These additional pieces of code are *domain-specific* information: the defined functions do not work in general situations and can therefore not be integrated in the kernel by default. The possibility to include this kind of information is one of the key features of PythonPDEVS concerning performance.

With domain-specific information it becomes possible to shift from "general purpose, though specifically optimized algorithms" to "domain-specific algorithms". The information being augmented here is used to deviate from *general* solutions to *specific* solutions. Problematic simulations, or even silent-failure, for models where the assumptions are not met thus become possible.

Should the user provide no domain-specific information, our simulation will simply default to *general* solutions to the problem. Some reasons for why no domain-specific information is added include:

1. Performance is no concern;
2. No domain-specific optimizations exist;
3. The achieved performance increase is negligible.

Many situations will fall under the first option: writing all domain-specific code is bothersome, certainly when performance is of no concern. A simulation that runs for 10 seconds wouldn't really benefit from these optimizations, as an improvement of 10% only makes it finish 1 second faster. However, if the model becomes bigger or simulated for a longer time, it might become a difference of 10 days to 9 days. On the other hand, if the same small model is simulated thousands of times, e.g. in an attempt to optimize some parameters, the difference can also become significant.

Domains with absolutely no domain-specific optimizations do not exist. Some of the more advanced ones might be unusable, though the extremely basic ones are applicable in every domain.

The increased performance being negligible is specific to the model. As with the first reason, performance might not be that high of a goal, making a small increase in performance negligible. The actual increase that can be expected is highly dependent on the model that is simulated. If most of the simulation time is spent within the models themselves (e.g. their transition functions), the increase will not be that spectacular as all optimizations are concerned with the actual simulation kernel overhead. Should most time be spent in the actual simulation kernel, then a noticeable performance improvement can be expected.

Because PythonPDEVS can become *more specific* to the simulated model, the kernel is extendable, as shown in Figure 4.7. The *red* part is the basic DEVS simulator, which is still compliant with normal DEVS models. In the *green* part, which is completely optional, the simulation kernel (right side of the figure) has several interfaces to insert domain specific information. In case the model lacks one of these components, the simulator will fall back to general solutions.

This performance increase comes at a cost: the user is required to write domain-specific code, which can cause much slower model development times if this information is expressed as part of the modelling effort. For this reason, it is useful to use the same mindset as when coding, where optimizations are only part of the last phase, when the design and model are already fixed. Should the design be altered, the domain specific parts likely require altering too, as otherwise deviations from the DEVS formalism are possible. Even when the design is unchanged, the user should be aware that such custom functions have the capability to deviate from the DEVS formalism. Should the DEVS formalism be violated, the simulation kernel can not guarantee consistency in e.g. distributed simulation.

A way around this is with the use of *domain-specific* modelling environments, which write out their models to the DEVS formalism for simulation. The assumptions can then be expressed by the program itself, using either static analysis of the model (which fields to copy) or with optimizations that are independent of the model (a domain-specific scheduler). Extending the model with this additional information is then a part of the translation process. Because our design is based on *optional extensions* of the model, other DEVS simulators that do not support such hints will not become unusable.

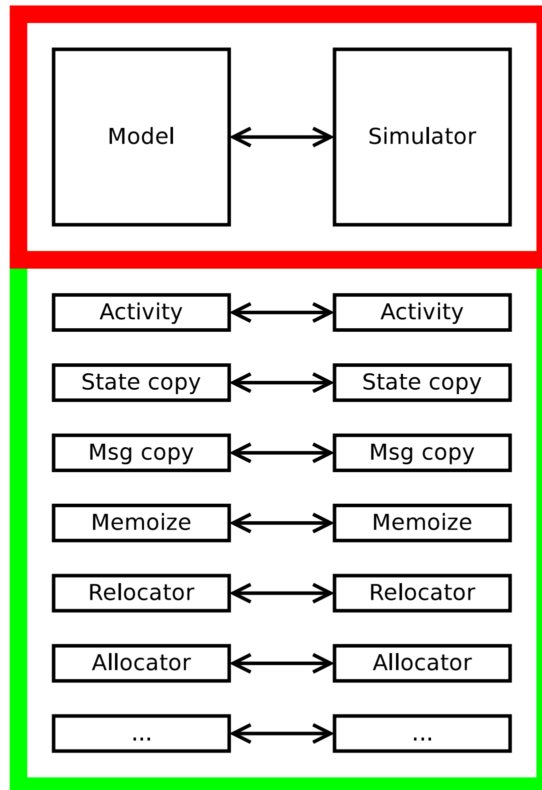

Figure 4.7: The extendability of the PythonPDEVS simulation kernel

Contrary to more general optimizations, domain-specific optimizations are nearly always applicable, making a running example possible. All of the mentioned optimizations are applied in the example model mentioned in the Listing A.1 and presented as a benchmark in Section 5.4.5.

4.4.1 Message copy

The DEVS formalism specifies that all components should be completely modular (although some variants allow partial-modularity[65]). Messages exchanged between models are restricted in two ways:

1. No references/pointers should be passed to data in the state of the model;
2. If multiple models receive the same data, they should not be able to alter each others message by altering the received data.

A possible solution to this problem is by using *copies* of the message instead of the real message. This means that the transition functions will only have access to a local copy of the message, complying with the required restrictions. It should be a *deep* copy, as otherwise it would still be possible to alter other data through references/pointers.

We offer the user the possibility to use a custom function, which defines how messages have to be copied. This introduces a notion of domain-specific information in the message copying methods. The two extreme cases are implemented by default: *deep copy everything* and *no copy*. All cases in between should be handled by the custom copy function that is provided.

Even if a custom copy method is defined, inter-node messaging will still use the `pickle` module, as a string is required for messaging.

Rationale

The basic approach in PythonPDEVS is to create a deep copy of every message that is being passed, which is compliant to the `DEVS` formalism. However, the user might have some knowledge about the message that is being passed and which parts of it don't require copying. Some parts might not require copying if the user does not alter them, or if they are only read once. For this reason, one of our domain-specific optimizations is to allow the user to define the copy function manually.

No copy functionality is implemented to make a fair(er) comparison to adevs feasible, as otherwise we would always lag behind a little because we adhere to the DEVS formalism more strictly. *Deep copy* is provided as it should be done for compliance to the DEVS formalism and can thus be used by default.

Other simulators implement message copying in a varying degree, as creating copies of each and every message is computationally expensive. They therefore place the complete responsibility with the user, which is equivalent to our *no copy* method.

One of the simulators that solves the problem differently is vle[57]. Vle does not create copies, but prevents the user from altering the message through the `const` keyword in C++. This is clearly more elegant, but was not an option for us, as Python does not support constants. Even though it is elegant, the enclosed pointers are not necessarily constant pointers, making modifications still possible. Technically, even if a constant pointer is passed, this allows for one way communication between the models as the value pointed to can change.

Adevs[54] doesn't support modular message passing and assumes that the user doesn't alter the values. This behavior is clearly a lot faster then always creating a deep copy.

Implementation

For the implementation, we only need to make a copy of the message as soon as it received. The actual method used to create the copy is chosen using a conditional. Default copy methods use the *cPickle* module that is provided by Python itself, *no copy* functionality simply skips the creation of the copy.

Note that the general copy method is slow, as it uses the *cPickle* module provided by Python, which effectively serializes the value to a string and parses it again. Pickling copies a lot of unnecessary things (internal Python attributes) and requires 2 phases: pickling and unpickling of the message.

Empirical results

The performance improvement when selecting *no copy* should be obvious, considering that copying can be skipped altogether. Figure 4.8 shows that defining a custom copy method is faster than using the default methods, even if the custom method simply copies all attributes manually. This can be explained due to the *cPickle* method using a string as an intermediate format, which also needs to be parsed again afterwards.

Running example

A custom message copy function can be found in our running example in Listing A.1. The `copy` methods of messages show this part of the code. For messages that have a simple structure, a simple assign for each attribute separately suffices. More complex messages might require some kind of recursion to create such a copy.

4.4.2 State saving

Every time warp implementation requires the possibility to create copies of the state of the models. These states are accessed when a rollback is performed and such accesses should return copies again. Several state saving methods are supported, some of them more general than others at the cost of lower performance. Others are plainly unusable in certain situations, whereas others will raise an exception if they are used outside of their domain. The following state saving methods are included:

1. **Deepcopy**: use the `deepcopy` module to create a copy. It can be used in every situation, though it is rather slow.
2. **Pickle**: use the `pickle` module to create a copy. It is a lot faster than the *deepcopy* method, but is unusable for constructs such as locks, threads,... However, such constructs should not be used in most DEVS models anyway. Note that the `cPickle` module is used if it is available.
3. **Pickle highest state**: identical to the `pickle` module, but force the highest pickling protocol.
4. **Copy**: use the `copy` module to create a copy. It only creates a shallow copy, so can cause erroneous simulation if a deeper copy is required.
5. **Marshal**: use the `marshal` module to create a copy. This is almost identical to the `copy` method, but will fail if a perfect copy couldn't be made.
6. **Assign**: simply assign the states. It is inherently unsafe, unless the state is of a primitive type.
7. **Custom copy**: use a custom method that must be defined on every class that is used as a message. Defining a custom method puts all the responsibility with the user and is the fastest in most simulations (apart from a simple assign of course). Also, this is the only method in which *domain-specific* information can be augmented.

Custom copy is the state saving method that allows the user to define custom state saving, possibly including domain knowledge. Similar to the custom message copy, the user might wish to copy only several parts of the state and thus have increased performance, both for simulation time and for memory consumption.

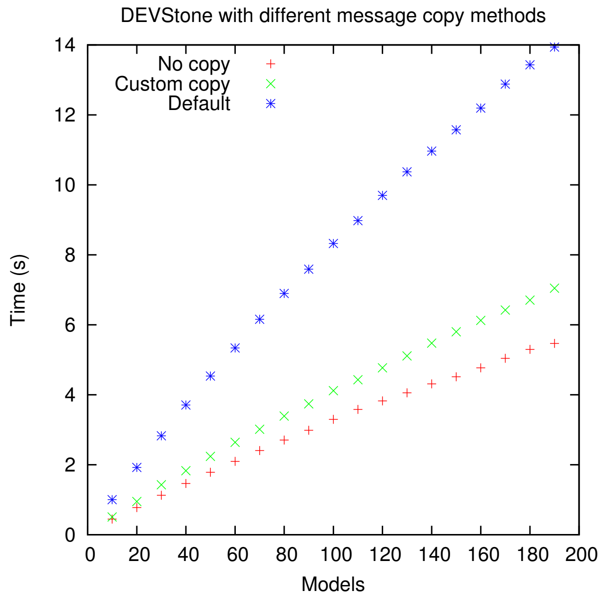

Figure 4.8: Comparison of message copy methods

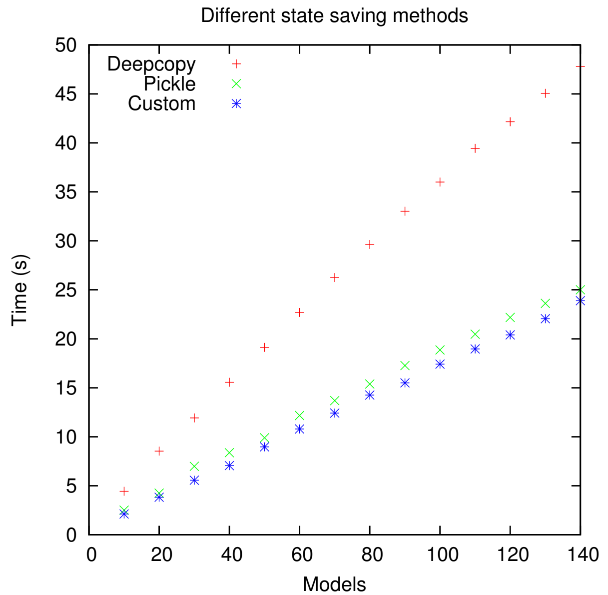

Figure 4.9: Comparison of state saving methods

Rationale

State saving is one of the requirements for a time warp implementation and it also poses the biggest overhead in a time warp implementation. It is thus the part that should require the most optimization to get an efficient distributed simulation. The number of rollbacks is also important, though inefficient state saving will make even perfectly parallelizable models run slower then is to be expected. So whereas rollbacks are meant to be the exception, state saving is guaranteed to happen after every transition function.

Message copying was a similar situation: a copy of a specific object is required, which might be created more efficiently with domain knowledge. As was the case for message copying, even a really basic manual copy method will be a lot faster than the one that is used by default.

Implementation

Implementation is straightforward, as we only need to make a distinction between all different options. To keep the complexity of the selection independent of the amount of possibilities, the state saving class is assigned to an attribute of the simulator. The different state saving methods are modular, though not meant to be extended by the user. User extensions have to be provided through the *custom copy* method, which calls the `copy` method on the state.

Empirical results

For our comparison we will only compare the most important methods: *deepcopy*, *pickle highest state* and *custom copy*. Figure 4.9 clearly shows a difference in performance, where *deepcopy* is slowest (as it is even more general) and *pickle highest state* being relatively fast. A *custom copy* function is still slightly better, even if all attributes are simply copied as is the case here. Note though that the custom copy is made twice: once when creating the saved state and once when loading the saved state. Whereas with pickling, the copy can be made partially by saving the intermediate string representation.

Running example

State saving is very similar to the message copy functionality, with the exception that it is defined on states instead of messages. Our running example again contains such examples, as shown in Listing A.1, with the `copy` methods that are defined on the states. Note that state saving methods are generally more complex than message copy methods, as states are generally more complex.

4.4.3 Memoization

Memoization is a frequently used technique in computer science, where the result of a function is stored together with the input values of the function. This requires deterministic functions: the return value should always be identical for identical inputs.

For DEVS simulation, the *transition functions* are the only parts that qualify for these conditions due to the design of the DEVS formalism. When memoization is used, a transition function with a specific combination of input values (in our case: state) will only be executed once. Later calls with the same input values will be redirected to the memoization data structure, which will return the cached value.

Domain information is used during the comparison of the different states. In Python, object comparison always defaults to *False*, making general memoization impossible for complex structures. While it is not completely *domain-specific*, it is required to use knowledge of the model to compare all attributes (at the required depth). Optionally, some parts of the state might be unnecessary to compare and could then be skipped for memoization. Be aware that, if some attributes are ignored, that attribute will be overwritten by the previously memoized value too.

Rationale

In sequential simulation, memoization is unlikely to yield any benefit for several reasons:

1. Chances for identical input values are rather slim in the general case
2. Memoization requires both states and return values to be saved, which is not done by default in sequential simulation. This causes all overhead associated with time warp to be introduced in sequential simulation too.
3. There is no simple heuristic to clean up the memoized values. This would be comparable to a time warp implementation without fossil collection: eventually memory will run out. Even though memoization is not vital to simulation, defining a good heuristic is important for performance.

Memoization is thus not implemented in local simulation. However, the user is free to write its own memoization code in the transition functions themselves.

Distributed simulation is different in all previous 3 points as follows:

1. Identical input values will always happen right after a rollback.

   When a straggler arrives, every model at the node is rolled back, even though it only affects one of them. As only a single model has changed, the states of all other models should still be identical, making memoization interesting. This single model could influence other models, making them too deviate from their memoized history.
2. States are already saved due to time warp, we simply need to make an additional reference to the created data.

   Time warp requires state saving to be done, so there is no additional overhead for creating the copy. However, states are saved slightly longer: as soon as a rollback occurs, the state history is normally removed, but now it is moved to the memoized state history. The sum of the state history and of the memoized states will never become larger then the maximal state history.
3. It is possible to assume that the chain of function calls that will happen is identical to the chain of calls in the past.

   As soon as this assumption fails, the memoization cache is cleaned because the chances for a hit have become too small. We do not use a hash map as a data structure, as cleaning up such a data structure not simple. Should one of the assumptions be violated for a model, the memoization history is cleared until another rollback occurs. This is not that hard of a requirement, as it means that we have deviated from the *known* path. Since we always assume that the same path is taken, deviations guarantee future misses and thus the memory can be freed earlier. Even though this is kind of a pessimistic approach, space needs to be taken into consideration, certainly in a distributed simulation.

A performance improvement is to be expected in distributed simulation, whereas in sequential simulation it would nearly always lower the performance. Certainly in combination with relocation, this causes very long rollback sequences. In later sections, we will use relocations to boost performance, which will only be possible if memoization is used to be able to quickly *roll forward* to the original state.

Implementation

Some small additions were necessary to the algorithm, but only to the place where the transition functions are called. Our changes are shown in Algorithm 4.1. The rollback algorithm needs to be altered too, by saving the state history in the memoization history instead of clearing it (not shown as it is a trivial modification).

As with other time consuming features, memoization is disabled by default. Enabling it by default would not give any benefit, as the user has to define a comparison method for it to work. If no such method is defined, the comparison would always fail and consequently simulation would only be delayed further.

Algorithm 4.1 Transition processing algorithm with memoization

```
newstate ← NULL
if memoization history not empty then
    if current state matches first memoized state then
        if internal transition then
            newstate ← memoizedstate
        else if external input values matches memoized input values then
            if confluent transition then
                newstate ← memoizedstate
            else if elapsed time matches elapsed time then
                if external transition then
                    newstate ← memoizedstate
                end if
            end if
        end if
    end if
end if
if newstate = NULL then
    clear memoization history
    execute normal transition code
else
    remove first memoized value
end if
```

Empirical results

Actual results are highly dependent on the amount of rollbacks that are done, combined with the time it takes for the computation of a transition function. If the transition function takes only a few microseconds, whereas the comparison takes even longer, memoization will never give any performance benefit. On the other hand, complex transition functions are likely to profit from memoization in most situations.

Figure 4.10 shows the influence of memoization for a synthetic model. As the artificial load increases, the effects of memoization become more apparent. Results without memoization contain slightly more jitter. Rollbacks, which happen nondeterministically, will have a higher influence on the total simulation time.

Running example

The running example contains memoization code for the most computationally intensive atomic models only, as it doesn't really matter for the others. Memoization code is simply the definition of a custom `__eq__` method on the state, as shown in Listing A.1. Most of the time, this method will simply compare every attribute for equality (possibly calling other equality functions).

4.4.4 Scheduler

The choice between different schedulers was already introduced in Section 4.2.3. While these schedulers are optimized for a specific domain, they are not yet domain-specific as they still work outside of their domain. That is, they do not utilize shortcuts or use simpler data structures than generally required.

As the scheduler is completely modularized, it is possible for the user to define custom schedulers. These custom schedulers can be virtually anything, as long as its external behavior is identical to the default schedulers. A domain that never uses Dynamic Structure DEVS for example, can skip the `schedule` and `unschedule` operations.

Rationale

Section 4.2.3 showed that the choice between different schedulers can have a significant impact on performance. If even general-purpose schedulers have such an impact, it is likely that a domain-specific scheduler would be even faster.

Optimizations can be done based on assumptions on the actual scheduler calls and the content of these calls. A simple example would be a form of *boolean* condition that indicates whether or not a model is *active* or *inactive*. All *active* models will transition at a specific time, whereas all *inactive* models will not transition in the future. Such assumptions could clearly simplify the scheduler code, making it a lot faster.

This possibility for higher performance comes at a cost again, as the correctness of the scheduler is vital for a correct simulation.

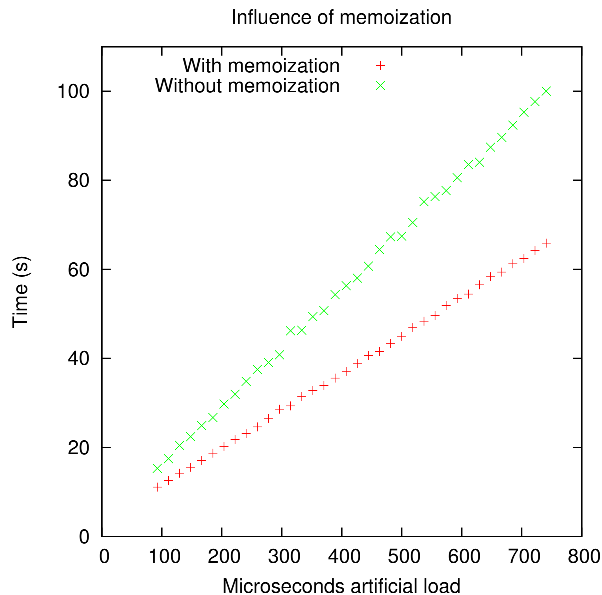

Figure 4.10: Performance comparison with and without memoization

When wrong assumptions are made, or even a slight bug is introduced, simulation will likely go wrong.

Implementation

No additional implementation is necessary, because all general-purpose schedulers are loaded the same way as a custom scheduler. In this respect, there is no difference between a custom scheduler or one that is provided by default. Only a small configuration method is required to allow the user to manually inject its own scheduler.

Empirical results

Creating a custom scheduler is more involved and requires more information about the domain. Therefore, empirical results for a custom scheduler are delayed until the fire spread benchmark in Section 4.5.1.

Running example

No domain-specific knowledge can be used to make a more efficient scheduler for our running example, as there is no specific pattern in these calls that can be optimized. A custom scheduler is therefore not present in the running example. Listing A.3 shows the implementation of the minimal list scheduler and should give enough information to write a custom scheduler.

4.5 Benchmarks

This section will present 3 benchmarks, of which one is a realistic model. The other two are synthetic benchmarks that are frequently used in the literature. Comparison to other simulators is difficult as no other popular DEVS simulator has results for these two benchmarks. Nonetheless, the complexity and performance of our simulator is tested in a variety of situations.

The next section will attempt to make a comparison with another DEVS simulator, called adevs.

All of these benchmarks will use all performance enhancements available, such as custom message copy, custom state saving, . . .

4.5.1 Fire spread

The first benchmark is a realistic one: a fire spread model. This model is often used in the literature in the context of activity, for example in [49, 50, 51, 47, 56, 46]. We will not yet tackle the activity component of this model, which is left for the next chapter.

The fire spread model boils down to a rectangular grid of atomic models. Each of these can have different characteristics, to model the different kinds of vegetation. All cells start of in the *not burning* state, except for a few cells that are in the *burning* state. Cells will emit their heat to neighboring cells. As soon as the temperature of a cell reaches a certain temperature, the cell itself will transition to the *burning* state. After some time, a *burning* cell will start cooling down and eventually transition to the *burned* state. It is thus clear that if the parameters are configured decently, that fire will spread throughout the cells, forming a

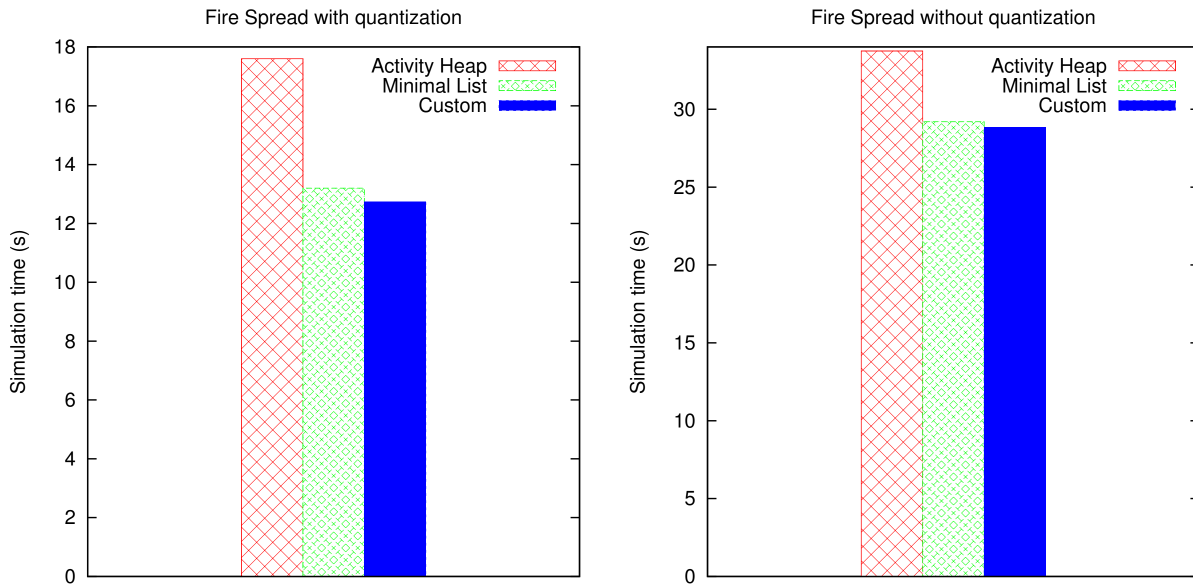

Figure 4.11: Fire spread benchmark with quantum 1.0   Figure 4.12: Fire spread benchmark without quantization

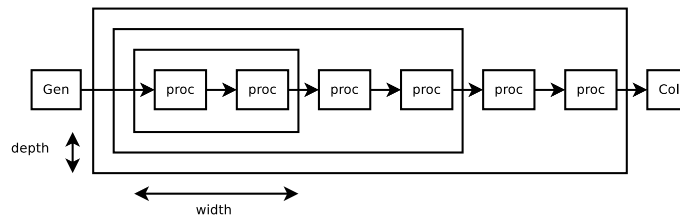

Figure 4.13: A DEVStone model with depth 3 and width 2

fire front. A more complete problem definition, we refer to [44], which is additionally the source of our used formulas.

As is common practice with this kind of models, we will use Quantized DEVS (Section 4.2.4). The same benchmark will also be ran without Quantized DEVS, to clearly show the difference.

The main objective of this benchmark is to show the effectiveness of a custom domain-specific scheduler. For the custom scheduler, we observe that simulation progresses in fixed time slices. Consequently, the scheduler can make a shortcut concerning the different times of transition. Our assumption is that "if the model is scheduled, it will transition at the same time as all other scheduled models". This allows for a scheduler that simply keeps a list of scheduled models and returns it afterwards.

Figure 4.11 (with quantization) and figure 4.12 (without quantization) show that this custom scheduler is slightly faster than even the fastest general-purpose scheduler, both with and without quantization. Note that the relative difference is bigger when Quantized DEVS is used, as the custom scheduler implicitly uses activity.

4.5.2 DEVStone

DEVStone[23] is a synthetic benchmark for local DEVS simulators. It is a synthetic benchmark, as it simply creates a chain of atomic models through which messages pass. An example is shown in Figure 4.13.

As the time advance value of the benchmark can be tuned, the used scheduler has a significant impact on performance. To have consistently good performance, it should thus be possible to choose between different schedulers. But even though this choice is vital to good performance, most DEVS simulators do not support such a choice. This indicates that there is a need for a tunable scheduler in other simulators, as otherwise simulator comparisons can be biased by chosing a specific number of collisions.

With collisions

Our first version of this benchmark is one with a lot of collisions. Such a situation can be caused simply by setting the time advance of every processor to a fixed value, e.g. 1.0. Every transition will be a confluent transition, which happens at every simulation step. Results are shown in Figure 4.14.

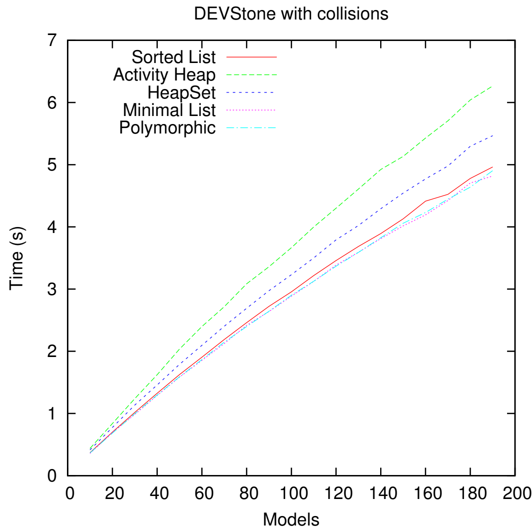

Figure 4.14: DEVStone results with collisions

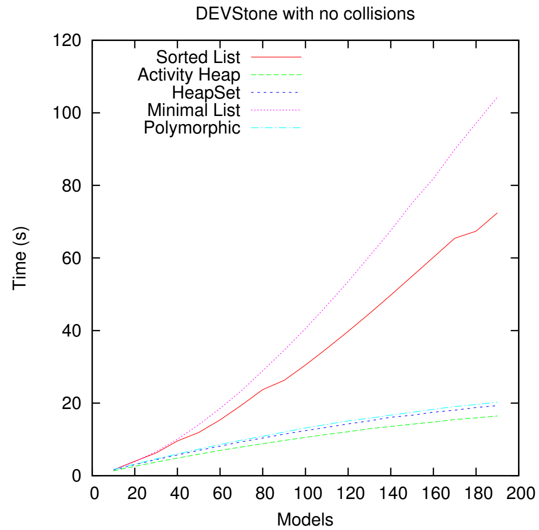

Figure 4.15: DEVStone results without collisions

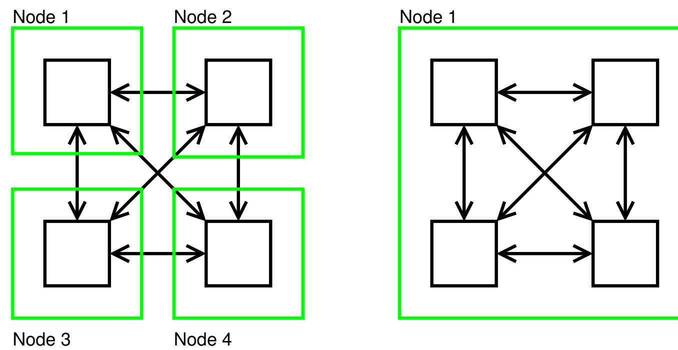

Figure 4.16: PHOLD model, both the distributed and local version

Ignore the *polymorphic* scheduler in these results for now. As it is partially activity-based, it is only presented in Section 5.2.1. However, to prevent repetition of the same figure later on, its performance is already included here.

The *list-based* approaches clearly perform best when a lot of collisions happen, confirming our intuition. Every scheduling operation has a complexity of $O(n)$, as the list is simply iterated. Heap-based approaches however, need to pop all elements (as all events are colliding) of the heap and push them back on somewhat later. Such an individual operation has a cost of $O(log(n))$, which is performed for all n models. The total complexity is thus $O(n \cdot log(n))$.

This confirms our rationale for choosing a list-based scheduler when collisions happen frequently.

Without collisions

To ensure that (nearly) no collisions happen, the time advance of the model is set to a random value. This random value is generated using the included random generator to ensure determinism. Messages are queued in the processor if the processor is still processing another message.

From Figure 4.15, it becomes clear that the *heap-based* approaches are far better when very few collisions happen.

With no collisions, every model will transition at a different time. Contrary to the previous benchmark, there are n times as much transitioning phases, but each phase only performs a single transition. The list-based approaches iterate over the list for every model, resulting in a complexity of $O(n)$ for every transitioning phase. Heap-based approaches only accesses $log(n)$ elements now, as they only pop and push a single model in a single phase.

This again confirms our rationale for choosing a heap-based scheduler when collisions happen infrequently.

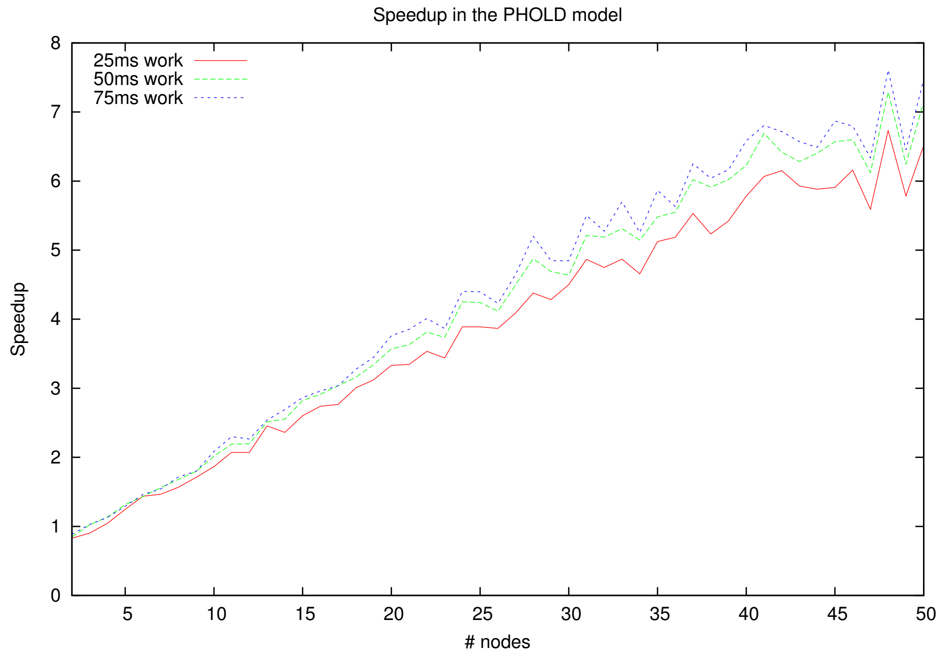

Figure 4.17: Performance for the PHOLD model on a cluster

4.5.3 PHOLD

In the literature, the PHOLD[21] model is often used as a distributed (DEVS) benchmark for different simulators[15, 53]. Other benchmarks, which are often more suitable for the implementation being benchmarked (and are less synthetic), include models for fire spreading[66, 41] or the game of life[24].

In the PHOLD model, a single atomic model is placed at every different simulation node. An example is shown in Figure 4.16. Each model starts with a single event that it has to process and sends it to a randomly chosen node afterwards. All models are therefore tightly connected to each other. The special case where a node sends the event to itself is not allowed, in which case a new random node is selected. Processing times of messages are also determined randomly. Of course, all randomness is again completely deterministic thanks to our random generator. In the equivalent local simulation, the same number of atomic models is simulated locally.

Some versions of the PHOLD benchmark run with multiple atomic models at a single node. This gives slightly more favorable results, as some message will stay at the same node and thus not cause roll backs. We have opted for the original approach, where only a single model is present at every node. This means that every message is a remote message, requiring the complete message passing path to be followed. Additionally, every message can be a straggler, causing frequent roll backs. Such a model is clearly a bad situation for a time warp implementation, as a lot of external communication happens.

The model is synthetic though, so performance in real situations should be a lot better. However, results are still valuable as it shows how our implementation behaves is non-ideal situations.

Results are shown in Figure 4.17, both for $25ms$, $50ms$ and $75ms$ of computation in the transition function. The actual speedup in these kinds of models is dependent on the amount of computation that is required in every model. If the computation takes nearly no time, the time warp overhead will become too big, which is confirmed by [66]. This also indicates that no speedup is to be expected when light computations are performed.

The occasional spike in the results is not due to the number of runs that is performed. Instead, the model is different every time and some of them simply distribute better due to a variety of reasons.

Compared to the previous version of PythonPDEVS, the speedup has actually decreased. Many more optimizations to the sequential simulation algorithm were performed. In absolute time, performance of the distributed version has increased though.

4.6 Comparison to adevs

This section will compare PythonPDEVS to adevs[54] using the DEVStone benchmark. We explicitly chose adevs, as adevs was found to be the fastest simulator in our previous comparisons[72].

Adevs is written completely in C++ and is basically only a set of header files, as it is completely based on the use of C++

templates. Adevs is compiled together with the model, creating a single executable at the end. The big advantage is that the compiler will have knowledge of both the simulator and the model for optimizations. The disadvantage is that every change to the model requires complete recompilation of everything, even the simulator. PythonPDEVS is completely interpreted [2] , making it ideal for fast prototyping.

A small adevs (atomic) model can be seen in Listing 4.1. The equivalent PythonPDEVS model is shown in Listing 4.2 (with comments omitted). The following differences are immediately visible:

- Adevs uses constant integers as ports instead of fully featured objects. Due to this, there is no real distinction between input and output ports. It also makes printing the names of ports more difficult, as they are not saved internally.
- In adevs, transition functions do not return the new state, but update the current object. Additionally, there is no separate *state* attribute, which makes it unclear which attributes are part of the *model state*.
- Adevs does not contain any tracers, so the user is required to do this manually using `cout` calls. All information that can be output like this, can also be output using our verbose tracer. It is also possible to write a custom tracer, which provides even more flexibility.
- The output function does not return the generated bag, but must update a provided bag that is passed by reference.
- Due to the implementation in C++, the user needs to do all object destruction manually.
- Due to the implementation in C++, all messages need to be strictly typed. This could be a nuisance with message passing.

4.6.1 Functionality comparison

In terms of functionality, adevs also differs significantly from PythonPDEVS. Some of the features that adevs doesn't provide are:

- **Tracing support**: the user has to write all tracers himself, which is a slight usability problem for quick prototyping and for new users. In parallel simulation, adevs uses conservative synchronization and thus it is not necessary to have kernel-support for tracing. Optimistic synchronization requires kernel support for this anyhow, as the kernel is the only entity that knows when an action is safe to process.
- **Termination conditions**: termination conditions are partially supported, in the sense that the user is required to manually call the *execNextEvent()* method on the simulator. This shifts the while loop to the user, which (in a sense) gives even more freedom to the user. On the other hand, parallel simulation only supports a termination time. PythonPDEVS supports the termination condition in both simulation methods.
- **Realtime simulation**: PythonPDEVS allows the same model to be executed in 3 different ways, completely transparent to the model itself.
- **Classic DEVS**: adevs only supports Parallel DEVS, in combination with Dynamic Structure DEVS

Adevs does provide some additional features that are not (or only to some extent) implemented in PythonPDEVS:

- **Modelica support**: adevs supports continuous models, which can be written in Modelica[19] and later be translated to adevs models.
- **Cell DEVS**: adevs has a specific class for Cell DEVS[80] models, which can be used for a specific set of problems. PythonPDEVS does not support Cell DEVS, though behavioraly equivalent models can be modelled using either Classic and Parallel DEVS. PythonPDEVS does have a cell tracer that can be used for manageable tracing output in these situations.
- **Listener objects**: mostly equivalent to our tracers, though it only intercepts exchanged messages and doesn't have insight in the state of the models that easily.
- **Routing methods**: PythonPDEVS has the user define all couplings using the *connectPorts* method, which can be passed a transfer function if required. Adevs on the other hand, has the user define a *route* method on the coupled model. It can be an arbitrary method that takes a bag as input and should return another bag as output. This method is then required to perform all routing manually and possibly perform some kind of transfer function. Static connections are not enforceable in this way, neither is direct connection possible. It also prevents the simulation kernel from having more information about the actual connections.

Clearly, PythonPDEVS is more aimed at *ease-of-use* by having default tracers, simple switching to Classic DEVS, simpler termination conditions, simple connections, ... Also, PythonPDEVS uses Python as its programming language, which greatly simplifies the implementation of both the simulation kernel and of the simulated models. Of course, an implementation in C++ has a performance advantage as the model is compiled and possibly optimized by the compiler.

[2] It is possible to compile the PythonPDEVS kernel using Cython[8], while retaining the interpreted nature of the models. Such a translation script is provided in the source code too.

```
#include "Clerk.h"
using namespace adevs;

// Assign locally unique identifiers to the ports
const int Clerk::arrive = 0;
const int Clerk::depart = 1;

Clerk::Clerk():
Atomic<IO_Type>(), // Initialize the parent Atomic model
t(0.0), // Set the clock to zero
t_spent(0.0) // No time spent on a customer so far
{}

void Clerk::delta_ext(double e, const Bag<IO_Type>& xb) {
    // Update the clock
    t += e;
    // Update the time spent on the customer at the front of the line
    if (!line.empty()) t_spent += e;
    // Add the new customers to the back of the line.
    Bag<IO_Type>::const_iterator i = xb.begin();
    for (; i != xb.end(); i++)
    {
        // Copy the incoming Customer and place it at the back of the line.
        line.push_back(new Customer(*((*i).value)));
        // Record the time at which the customer entered the line.
        line.back()->tenter = t;
    }
}

void Clerk::delta_int() {
    // Update the clock
    t += ta();
    // Reset the spent time
    t_spent = 0.0;
    // Remove the departing customer from the front of the line.
    line.pop_front();
}

void Clerk::delta_conf(const Bag<IO_Type>& xb) {
    delta_int();
    delta_ext(0.0,xb);
}

void Clerk::output_func(Bag<IO_Type>& yb) {
    // Get the departing customer
    Customer* leaving = line.front();
    // Set the departure time
    leaving->tleave = t + ta();
    // Eject the customer
    IO_Type y(depart,leaving);
    yb.insert(y);
}

double Clerk::ta() {
    // If the list is empty, then next event is at inf
    if (line.empty()) return DBL_MAX;
    // Otherwise, return the time remaining to process the current customer
    return line.front()->twait-t_spent;
}

void Clerk::gc_output(Bag<IO_Type>& g) {
    // Delete the outgoing customer objects
    Bag<IO_Type>::iterator i;
    for (i = g.begin(); i != g.end(); i++) delete (*i).value;
}

Clerk::~Clerk() {
    // Delete anything remaining in the customer queue
    list<Customer*>::iterator i;
    for (i = line.begin(); i != line.end(); i++) delete *i;
}
```

Listing 4.1: Example adevs model from its documentation slightly shortened to fit on a single page

```python
from DEVS import AtomicDEVS
from customer import Customer

class ClerkState(object):
    def __init__(self):
        self.t = 0.0
        self.t_spent = 0.0
        self.line = []

class Clerk(AtomicDEVS):
    def __init__(self):
        AtomicDEVS.__init__(self, "Clerk")
        self.arrive = self.addInPort("arrive")
        self.depart = self.addOutPort("depart")

    def extTransition(self, inputs):
        self.state.t += self.elapsed
        if len(self.state.line) > 0:
            self.state.t_spent += self.elapsed
        for message in inputs[self.arrive]:
            self.state.line.append(Customer(message.value, self.state.t))
        return self.state

    def intTransition(self):
        self.state.t += self.timeAdvance()
        self.state.t_spent = 0.0
        self.state.line.pop(0)
        return self.state

    def confTransition(self, inputs):
        self.state = self.intTransition()
        return self.extTransition(inputs)

    def outputFnc(self):
        leaving = self.state.line[0]
        leaving.tleave = self.state.time + self.timeAdvance()
        return {self.depart: [leaving]}

    def timeAdvance(self):
        return INFINITY if len(self.state.line) == 0 else self.state.line[0].t_spent
```

Listing 4.2: Example PythonPDEVS model that is semantically equal to Listing 4.1

4.6.2 Performance comparison

In terms of performance, we will only compare the *local* simulation performance. Several reasons exist why a comparison of the *parallel and distributed* features would be worthless:

- Adevs only supports shared-memory simulation, whereas PythonPDEVS does not make shared-memory assumptions. This has a drastic influence on the used algorithms, and corresponding performance. However, an adevs simulation only scales up to a single machine, whereas PythonPDEVS can run on several machines without shared-memory.
- Adevs uses conservative synchronization, rendering a comparison of parallel performance into a comparison between conservative and optimistic synchronization. A huge difference is possible between both approaches, even if exactly the same model is used[22]. Furthermore, conservative synchronization requires the lookahead value to be known and provided to the simulator for decent performance.
- Adevs uses a different middleware for communication. Due to the different assumptions about shared-memory, PythonPDEVS uses MPI as its middleware, while adevs uses OpenMP. OpenMP effectively uses the available shared-memory for communication, whereas MPI uses the network.

DEVStone is chosen as a benchmark due to its simplicity. The same distinction between lots of collisions and no collisions will be made again. For PythonPDEVS, the most efficient scheduler will be used, depending on the model. For adevs, we do not have a choice for which scheduler to use. As adevs is written in C++, we have the possibility to compile it with different compiler directives.

We have used *adevs 2.7* in our comparison, compiled with *GCC 4.7.3*, with compiler optimizations enabled (*-O2*). To get more comparative results, PythonPDEVS was executed with *PyPy 2.2.1*. PyPy is an alternative Python interpreter that features a JIT compiler and is primarily aimed at high performance.

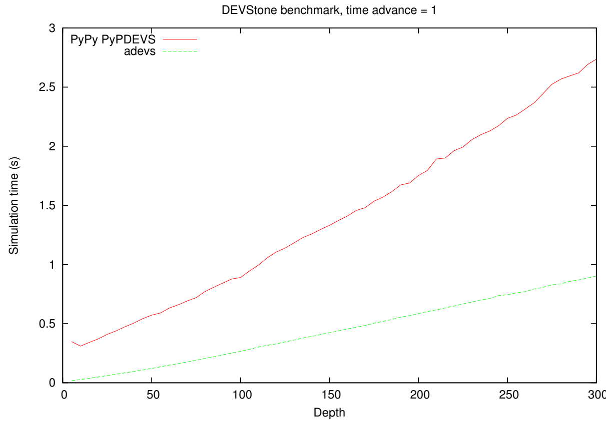

Figure 4.18: Comparison to adevs with *ta* of 1

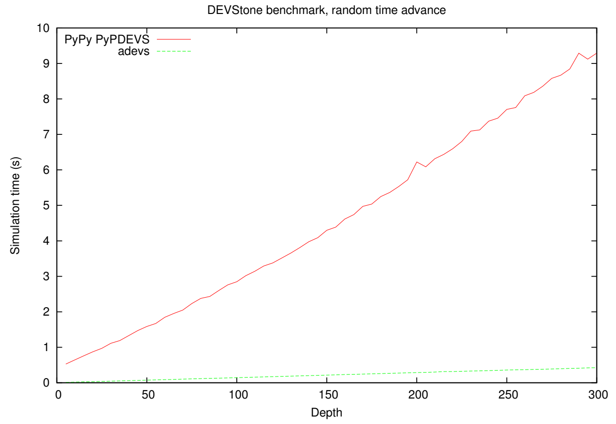

Figure 4.19: Comparison to adevs with random *ta*

The results with many collisions are shown in Figure 4.18, whereas results for no collisions are shown in Figure 4.19. It is clear that adevs is still the fastest simulator in both situations. With no collisions happening, both PythonPDEVS and adevs use the same scheduler (activity heap), resulting in much higher performance in adevs. However, with lots of collisions happening, adevs will still use this scheduler, whereas PythonPDEVS can switch over to a list-based scheduler. Even though adevs is still the fastest, the gap has narrowed significantly due to the modularity of the scheduler, which is a form of a domain-specific hint.

All in all, adevs remains unbeaten in terms of performance. The most important consideration is that the order of complexity is the same for both adevs and PythonPDEVS.

5
Activity

In this chapter, *activity* is introduced in a variety of forms. The primary use of activity will be to enhance simulation performance. By making PythonPDEVS aware of activity and having it use it for increased performance, PythonPDEVS becomes an *activity-aware* simulator. Activity is the pinnacle of *domain specific* information in PythonPDEVS.

First of all, the different definitions of *activity* are presented. Our application of activity is shown afterwards, together with several methods of extracting it from the simulation. To conclude, several benchmarks are shown, which exploit the measured activity to obtain higher performance.

5.1 Definitions

The term "*activity*" has been used for a variety of things, of which only several are relevant to this thesis. An excellent overview is given by [48], which forms the basis for this section. Even though many definitions are provided in said work, our main focus will be on the "*Activity in discrete-event systems*" and "*Determination of activity configurations in CA*" sections.

5.1.1 Activity scanning

Earlier use of *activity* was in the context of *activity scanning*, which is one of the four major frameworks[4]. Activity scanning is a two-phase approach: the first phase is dedicated to simulation time management, and the second phase to the execution of conditional activities. This is visualized in Figure 5.1.

A similar conceptual framework is the three-phase approach: the first phase is identical to the two-phase approach, thought the second phase is split up in finding unconditional activities and finding conditional activities.

Activity is defined as "An activity is an operation that transforms the state of a system over time" in these approaches.

5.1.2 Activity in resources

We will now introduce the definitions of activity that have recently emerged and which are used in the remainder of this chapter.

Qualitative

The first definition only distinguishes between *active* and *inactive* models. In this context, there is no numerical value (quantity) linked to the activity. An example of such a distinction is that *inactive* models have a time advance that returns ∞. *Active* models have a time advance which returns a finite real number.

A system is *qualitatively inactive* when no events occur and *qualitatively active* otherwise. The horizon signifies the period in simulated time in which the (in)active model is consistently (in)active.

Quantitative

Quantitative activity will link the activity to a numerical value, turning it into a measure. In this context, the horizon implies the period in simulated time over which the quantity is measured.

The *quantitative activity* is defined as the sum of *quantitative internal activity* and *quantitative external activity*. *Quantitative*

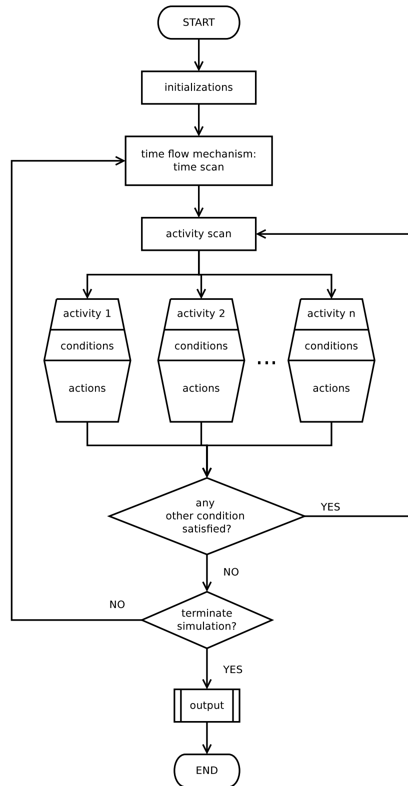

Figure 5.1: Activity scanning in the two-phase approach

internal activity corresponds to the number of internal discrete events in the horizon. It provides information about the quantity of internal computations within atomic models. *Quantitative external activity* corresponds to the number of external discrete events in the horizon. It provides information about the quantity of messages exchanged between atomic models.

We will take a slight deviation from these definitions in order to allow for domain-specific hints. Additionally, our deviation allows for more fine-grained control over the notion of activity. We emphasize the distinction between *resource usage* (quantitative internal activity) and *connection usage* (quantitative external activity) more clearly.

Quantitative internal activity In our definition, quantitative internal activity corresponds to the consumption of resources caused by the state transitions. Internal, external and confluent transitions all can cause state transitions, so all of them are measured.

This allows for the following changes:

1. We deviate from a simple transition counter, to a more fine grained representation that is configurable by the user.
2. Different forms of activity can be considered, such as the time spent in the transition functions, the memory consumed by the transition function, or even a domain-specific notion of activity.
3. External and confluent transition functions are included too, as they too are responsible for resource consumption.

Quantitative external activity Our definition of quantitative external activity corresponds to the amount of exchanged events over a specific connection between different models. Even though this activity value is still a counter, information is no longer limited to the model performing the external transition. By only taking the model with the external transition into account, no information about the source is saved.

The most important consequence is that frequently used connections can be detected, instead of only the most frequently receiving model.

Resemblance to atomic and coupled models This distinction has similarities to the distinction between roles of atomic and coupled models: atomic models contain the *behavior* of the model, whereas coupled models contain the *structure* of the model.

Just like a model being composed of both atomic and coupled models, our internal and external activity can also be composed to give a complete activity overview. Internal and external activity are thus orthogonal in our definition, as shown in Figure 5.2. Simply adding them up (as is done in the original definition of quantitative activity) is useless.

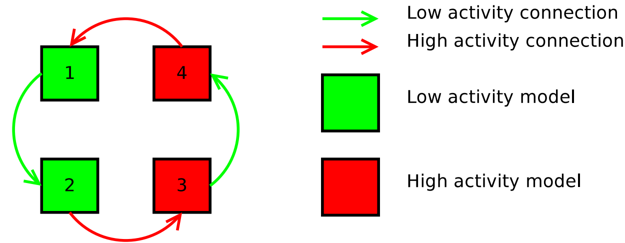

Figure 5.2: Different activity definitions

Others

As activity can be defined more generally as a *consumption of resources*, several other definitions might seem fitting. The *resources* that are being used are not necessarily related to the processing time or number of events, but could also be extended to e.g. energy consumption[28] or memory consumption.

Activity is used as a concept outside of the computer science field too, for example in biology, economics and epistemology[48].

5.2 Applications

To achieve our goals, we need to be able to use the activity with the intention of increasing performance. PythonPDEVS uses all three definitions of activity: qualitative, quantitative internal and quantitative external activity. In our applications, the first two can be modified by the user to add in a domain-specific notion. All of these activities serve a different goal in our implementation, which makes it possible to use all of them simultaneously.

5.2.1 Qualitative

Qualitative activity is the most straightforward definition and is thus treated first. A model is either *active* or *inactive*, so no distinction is made between *highly active* or *barely active* nodes. An *inactive* model cannot influence the course of the simulation, so it can be completely ignored in the simulation algorithms. The model can only become active by receiving an external input.

By removing models from the simulation, the complexity will decrease as it becomes dependent on the number of active models instead of the total number of models. The scheduler is the only component in the simulator that necessarily has to take into account all models. In order to influence the complexity, activity should be used to speed up the scheduler. Our optimizations are not that difficult, and are even provided by default in simulators such as adevs[54] and vle[57]. It basically consists of *not scheduling* inactive models, that is, models with a time advance equal to $+\infty$.

This is a simple observation, as these models will never be returned by the scheduler. The only possibility to have these models activated is through an external input message, which is independent of the scheduler. Inactive models don't even need to be remembered, additionally saving on space.

The simplest implementation would be in an *activity heap* implementation of the scheduler: a normal *heap* scheduler is simply extended by performing a check for the `timeNext` of the model. If the `timeNext` is equal to ∞, scheduling the model is skipped. No additional complexity is introduced: models were rescheduled on a 1-by-1 basis anyway.

A model that can profit from this is shown in Figure 5.3. This is the same fire spread model from a previous benchmark (Section 4.5.1). All inactive models are visualized as *blue*, whereas even the slightest hint of *red* indicates activity. An activity-aware scheduler will take into account only the red models (r^2 models), whereas a normal scheduler would take all of them (n^2 models) into account. Therefore, the complexity of the scheduler is reduced from $O(n^2)$ to $O(r^2)$, with $r \leq n$.

Not all schedulers are able to profit from inactivity though. For example the *minimal list* scheduler: it iterates over the complete list as soon as an operation is performed, ignoring all kinds of activity. The design of this scheduler simply isn't fit for activity: adding it in will have a negative effect. Inactivity needs to be detected, which requires an additional $O(k)$ operation, and the inactive models have to be removed from the list, which is an additional $O(n)$ operation per inactive model. Should no model be inactive, all this time is wasted. As if that wasn't enough, all models that are rescheduled would need to be checked for whether or not they are already present in the list, which requires additional checks. The only function that gets (slightly) faster is the function that searches for the first model to transition: only the *active* models have to be checked. All advantages of the *minimal list* scheduler, such as performance being irrelevant to the number of collisions, are lost due to the activity check.

With a modular scheduler, it is possible to introduce a *domain-specific* concept of activity too. Our approach up until now was perfectly general and in widespread use. However, the user might have some additional knowledge about models that will never transition again, even if they were to have a finite time advance. Such models too can be removed from the scheduler, further reducing complexity. An example might be a model that has a finite time advance, but is always interrupted by other external

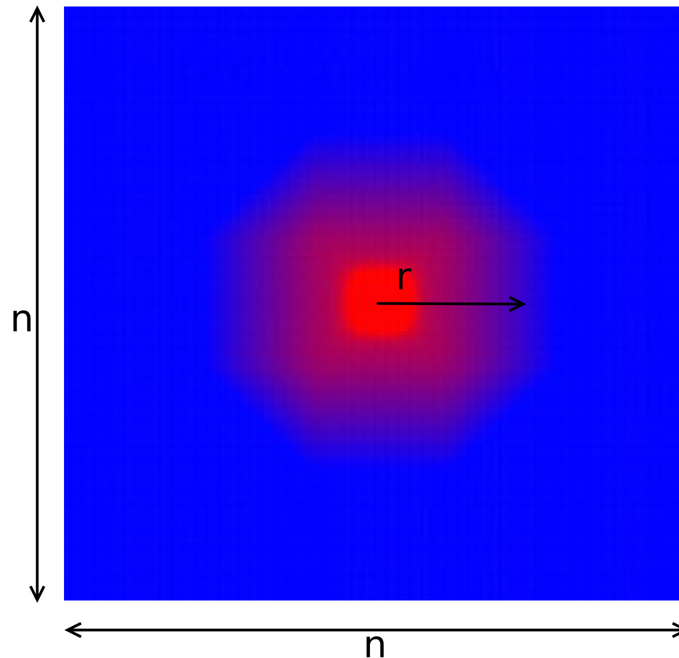

Figure 5.3: Application of qualitative activity can reduce complexity from $O(n)$ to $O(r)$

interrupt. As a consequence, the model never performs an internal transition, making scheduling unnecessary.

Polymorphic scheduler

Partially related to activity-aware optimizations is the use of a polymorphic scheduler, which is a scheduler that alters its data structure at run-time. The decision on whether or not to switch, and to what data structure, is made using heuristics based on the tracked access patterns. A similar approach was also taken in Meijin++[52], which adapted its data structure to current access patterns.

It doesn't really use qualitative activity as defined before, though it is largely similar: a polymorphic scheduler *tracks patterns* and tries to *optimize for them*. The speedup is gained by alternating between heap-based and list-based data structures (Section 4.2.3), depending on the situation.

At first sight, a polymorphic scheduler might seem unnecessary if the ideal scheduler is defined before simulation starts. Such choices are static however, while the ideal scheduler might vary throughout the simulation. The polymorphic scheduler addresses this issue by continuously monitoring the scheduling operations and possibly altering the data structure if it becomes unfit.

A polymorphic scheduler is clearly the most advanced usage of a modularized scheduler. A basic version is provided, which is also the one that is benchmarked. This basic version chooses between two different schedulers: one of them list-based, whereas the other is heap-based. If lots of collisions are detected, the scheduler will switch to a list-based scheduler. On the other hand, if nearly no collisions are detected, the heap-based scheduler will be chosen. While this should be sufficient in most situations, it is possible to write your own polymorphic scheduler, based on the provided one. By customizing this scheduler, different heuristics can be implemented, or a different set of schedulers can be used.

5.2.2 Quantitative internal

Recall that in our definition in Section 5.1.2, quantitative internal activity was the amount of resources spent within the transition functions. This activity measure will be used to perform automated, activity-driven relocations that are used for load-balancing[38] of the activity.

Measurement

In order to find the activity measure, the transition function needs to be wrapped with some tracing code. In the *activity tracking plug-in* in DEVSimPy[11, 60], this activity measure was hard-coded to be the amount of CPU time spent within the function. By default, the PythonPDEVS activity measure will also be an estimation of this time.

Several possibilities exist to estimate this metric:

1. Using the `time` python module.

```
yentl ~ $ python -m timeit -s "import␣time" -- "time.time()"
10000000 loops, best of 3: 0.0801 usec per loop
yentl ~ $ python -m timeit -s "import␣time" -- "time.clock()"
10000000 loops, best of 3: 0.199 usec per loop
yentl ~ $ python -m timeit -s "import␣resource" -- "resource.getrusage(resource.RUSAGE_SELF).ru_utime"
1000000 loops, best of 3: 0.642 usec per loop
yentl ~ $ python -m timeit -s "import␣psutil" -- "psutil.cpu_times().user"
10000 loops, best of 3: 25.9 usec per loop
```

Listing 5.1: Comparison of different activity measuring methods

method	time per loop	times slower than `time.time()`
`time.time()`	0.08 usec	1x
`time.clock()`	0.199 usec	2.5x
`resource`	0.642 usec	8x
`psutil`	25.9 usec	324x

Table 5.1: Comparison of different activity measuring methods

This library is only concerned with the wall clock time. If no additional computation happens at the nodes, this gives a crude estimate to the CPU time. It requires the assumption that nearly no context switches happen. However, time warp performs badly if lots of context switches happen, making the lack thereof a fair assumption. On the bright side, the wall clock time can be accessed very rapidly using built-in functions. All in all, the results are fast, though are only a crude estimate when a lot of context switching happens.

2. Using the `resource` python module.

   This library returns information on the resource usage of the currently running interpreter. One of the returned metrics is the CPU time spent within this program. However, PythonPDEVS is a multi-threaded application and therefore CPU time will also advance in different threads of the simulator code. Measurements are thus also estimates, though they are unaffected by context switches caused by the operating system.

   More information is available, like the amount of memory consumed, number of context switches, ... All of them are irrelevant to our general strategy though. Should other resource consumption metrics be desired, this module might be the only possibility.

   Even though it is more accurate, it takes a lot longer to access these values.

3. Using external modules, like `psutil`.

   The `psutil` library is similar to the `resource` library, but it is not provided in default Python distributions. It can return even more information, though this is again unnecessary as we are only interested in the CPU time spent within a single thread. However, usage of this library is even slower than using the `resource` library, making it unsuited for performance sensitive applications. This is the implementation chosen in DEVSimPy.

A comparison of the invocation time required to call each of the functions is given in table 5.1. Measurements were conducted as specified in Listing 5.1. The `psutil` module has an enormous overhead compared to the others.

Because our goal is to optimize for performance, having very expensive tracking methods will not pay of: the overhead imposed by the information gathering would be higher than the potential gains in most situations. We will opt to use the `time` module by default due to its low overhead. If few context switches happen, the passed wall-clock time will be a good estimate and provide decent results.

Custom definition

The CPU time spent provides a decent, general estimate for the computational load in the transition functions. This general metric is insufficient in several cases for the following reasons:

1. Results are inaccurate and unreliable in case of context switching.

   As was previously mentioned, the `time` module returns the difference in wall clock time, not in CPU time. Results can be very inaccurate in the presence of context switching, thread switching, I/O operations, ... While it is true that these situations are rare with time warp, even periodic extra load can have a huge impact on the measured values. In the worst case, only a select number of nodes have an overloaded CPU, causing these nodes to spread incorrect results to other nodes.

```
preValue = model.preActivityCalculation()
model.inputValues = deepcopy(model.inputValues)
model.state = model.transition()
activity = model.postActivityCalculation(preValue)
```

Listing 5.2: Pseudo-code in Python indicating the place of both activity calls

2. The obtained activity values are non-deterministic and unrepeatable.

   This is partially linked to the previous remark, though still sufficiently different. Even if the CPU was otherwise idling and simulation goes smoothly with minimal thread switching, the execution time of a specific function is non-deterministic. Several low-level mechanisms cause such non-deterministic results, like disk I/O, caching effects, . . . Deterministic results are helpful during debugging or when optimizing for a specific situation. It is likely that these non-deterministic values are relatively close to each other, though the problem remains.

3. Results don't use domain-specific information and consequently, the results don't have any real meaning to the user.

   Take for example the activity measure: "Model *A* spent 3 seconds and model *B* spent 1 second in the transition functions". This is likely not that helpful when performing load balancing, as it only states that model *A* is 3 times as active as model *B*. No actual *insight* in the model is given, only a statement about how long it took. Depending on the model, some more specific knowledge could be combined with this. A more valuable statement might be "Model *A* had an average temperature of 41 degrees in the past time period, whereas model *B* had temperature 20". Apart from being much closer to the domain, it also provides insight in the state, making activity predictions a lot easier.

4. Slightly faster metrics might be possible.

   This is not really a problem, as the default happens almost instantaneous. However, it is possible that an even faster metric is already present in the state itself, like the temperature at the current simulation time. So in addition to being more accurate and deterministic, it is also slightly faster.

Specifying a custom *activity concept* is thus advised in many situations, though most users don't want to think about activity. Such functionality should be *completely transparent* to the end user. If a custom definition is given, that one will be used, otherwise the default (wall clock time-based) is used. This is simply a case of polymorphism: every atomic model is an instance of a subclass of the `AtomicDEVS` class. The `AtomicDEVS` class has the functions defined with the default. The user's model is always a subclass of the `AtomicDEVS`, so inheritance is done automatically. If some models have a custom definition and others have the default, the user needs to take this into account at all places where the values are used.

Our implementation puts two methods around the actual transition call, as shown in Listing 5.2. The default implementation of the `preActivityCalculation()` method, called right before the transition, simply returns the current system time. In the `postActivityCalculation(preValue)` method, which is called after the transition, this time is taken and subtracted from the current system time. At the end, the passed wall clock time is returned and used as activity.

What is special is the split in a *pre-* and *post-*transition part. This way, the user can access both the original and new state (in `preActivityCalculation` and `postActivityCalculation` respectively). Information can be passed from the first part to the second part in the form of a parameter.

A single method with access to both the original state and the new state is impossible for two reasons:

1. Some activity measures want to compare the system state (e.g. resource consumption) before and after the transition. This effectively requires two calls: one before the transition and one after.

2. It is possible to update the state instead of creating a completely new state. Therefore, there is no instance of the original and new state at the same time. Creating a copy of the state first would be a complete waste of time.

Whatever activity value is returned, its usage is perfectly safe in the sense that simulation will stay correct. The only negative impact caused by incorrect activity values is slower simulation. This is kind of logical, as the simulator will perform optimizations for the returned activity. Simulation will always be correct, no matter how wrong the prediction was, as otherwise the user might be reluctant to try out custom activity definitions.

On the other hand, custom schedulers are potentially unsafe towards the user and should be avoided unless the user is absolutely sure about what is going on.

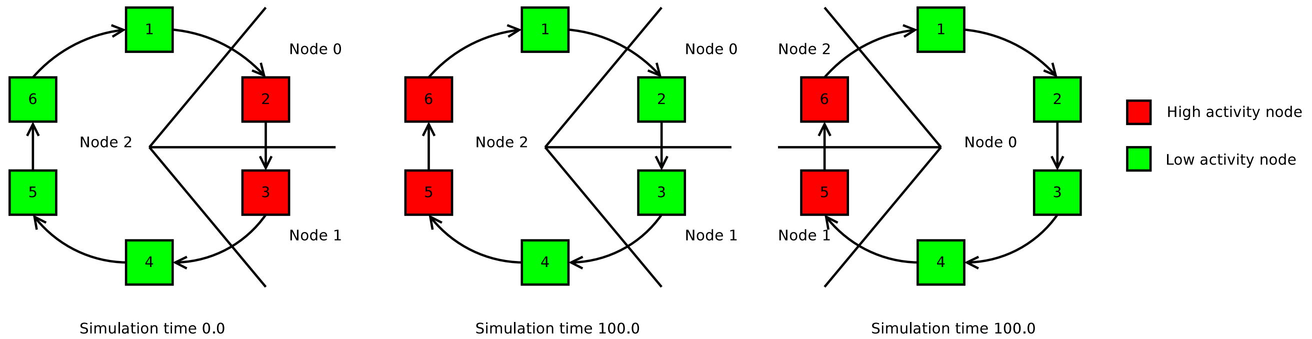

Figure 5.4: Activity relocation on a synthetic ring model

Management

The gathered activity should still be saved somewhere. In sequential simulation, the activity could be added to a counter stored inside the model. Should the activity be more complex, adding it in a list is possible too. When the activity gets used, the counter (or list) is simply returned and reset afterwards.

In distributed simulation, using time warp, this approach is clearly insufficient. First of all, it is possible for a computation to be rolled back. When this happens, the associated activity also has to be removed. Otherwise the results will always be non-deterministic even if a deterministic method is used. The activity information is paired with the state history that is saved as part of the time warp algorithm. The tuple to be saved is extended from `<timeLast, timeNext, state>` to `<timeLast, timeNext, state, activity>`.

Secondly, it might be possible for the user to define activity as something different from a numerical value (a tuple perhaps). Should they simply be accumulated, this is impossible as only numbers can be added. Activity values are thus saved per model and are only accumulated as soon as they are required. Accumulation is fairly simple, as the state history is already traversed during fossil collection. Therefore, the complexity of the simulation algorithm doesn't change as a result of activity tracking and management.

Usage

Contrary to qualitative activity, which is usable in both local and distributed simulation, quantitative internal activity can only be used in distributed simulation. This is logical, as we use it for load balancing, which only has a use in distributed simulation. The *relocator* is extended and becomes completely user-configurable for this purpose. Instead of simply processing relocation rules, the relocator will call a method with the found activities as its parameters. It is then requested to return a list of modifications that should be performed in order to optimize the simulation.

Even though PythonPDEVS supports manual relocations, their usefulness is not at the same level as dynamic relocations. Examples of such models are models with moving *activity regions* such as fire spread and city layout. With activity-based relocations, the model is constantly analyzed and can use smarter relocation strategies. The use of manual relocations is an option if the model and its behavior is clearly understood. However, altering a small parameter of the model might drastically alter this behavior, rendering manual relocations insufficient.

An example of an activity relocation is shown in Figure 5.4. At first, the initial allocation is chosen in such a way that there is a perfect distribution of the load. Consequently, time warp will perform very well at the start of the simulation. After some time however, different models become highly active, whereas previously very active models might become almost inactive. The initial allocation was static and thus doesn't adapt to this situation. As a result, simulation slows down as simulation progresses. With activity relocation, the distribution of activity (load) is monitored throughout the simulation. The relocator will detect the imbalance and move the highly active nodes to their own node again.

As the behavior is simple here, manual relocations could be a solution too, but this requires manual work from the user: when to perform the relocations, which relocations, . . . If for some reason the speed at which the activity passes through the models changes, such manual relocations have to be adapted too. On the other hand, activity-based relocations will automatically search for relocations that have a positive effect on the simulation performance. It offers all the advantages of manual relocations, with the added benefit that these rules are found automatically. As they are found automatically, they are automatically inferred from the model and require no model-specific user information. The relocator will be able to adapt itself to changing situations.

The domain-information related to activity is distributed over two places: the activity definition and the activity relocator. Whereas the activity definition can be used in its most general form, the relocator requires some domain information most of the time. For example, one activity relocator is enclosed in the PythonPDEVS distribution, which simply tries to exchange

models between different nodes. This is ideal for some models (like the benchmark in Section 5.4.4), but very bad for others (like the fire spread model from Section 4.5.1). The user is encouraged to write a custom relocator for maximal performance.

5.2.3 Quantitative external

Recall that in our definition, quantitative external activity corresponds to the amount of exchanged events over a specific connection between different models. We will combine quantitative external activity with quantitative internal activity to determine a good *initial allocation*. This will be called a *dynamic allocator*, being the dynamic variant of the static allocator (Section 4.3.1).

Measurement

The measurement of external activity is done by incrementing a counter $A \rightarrow B$ as soon as an event is passed from port A to port B. Whereas quantitative internal activity measured the amount of resources consumed, this is not necessary for external activity. Every exchanged event incurs the same computations and consequently the same amount of used resources. Knowing the amount of exchanged messages is sufficient to know the resource usage of this connection.

Custom definition

Contrary to quantitative internal activity, there is no need for a custom definition here. When a transition happens at model A, it is unknown what the internal activity is. The transition might contain a conditional that causes long computations some times, but very short computation otherwise. As this conditional is only dependent on the state (as is enforced by DEVS), information about the state is necessary.

The processing of messages at the simulator level is a lot more distinctive in itself. Most likely, the kind of events passed over a specific connection is fixed. If this is not the case, splitting up this connection in multiple connections, by using more ports, offers a sufficient solution. This change might seem invasive as it changes the model, though it is most likely already present in well-designed models. Additionally, using the same connection for unrelated messages is misuse of the dynamic typing allowed by Python. Simulators that use statically typed languages enforce this automatically.

Note that the allocator will be able to combine both external and internal activity. The internal activity is still customizable as was previously mentioned. Even though the measurement of quantitative external activity isn't customizable, the allocator is still completely customizable.

Management

As was the case with quantitative internal activity, a rollback has to remove the activity that was caused by the messages being rolled back. The main reason for this is to prevent artificially high activity in specific models due to frequent rollbacks. This metric is again computed over different simulation nodes, but if one of them rolls back frequently, all of its links will seem a lot more active even if this is not the case. Due to artificially high values, wrong decisions are likely to be made.

External activity measurement is only performed at the start of the simulation and only on *local, sequential* simulation. The use of local simulation might seem strange, as we are searching for a good distribution. Simulations that use quantitative external activity will run in *forced local* mode. This is enforced until the dynamic allocator states that it has enough information to make a good allocation. Rollbacks are clearly avoided and activity measurements will be accurate.

Sequential simulation is forced instead of relying on a similar mechanism as the state history, because no "link history" is in place. Such a link history would have no use outside of quantitative external activity tracking, inducing a severe overhead.

Saving the measured activity now becomes trivial as no rollbacks occur and every message will simply increase a counter by 1. This counter is stored in the form of a two-dimensional array, but uses a hash map instead. For one, hash maps are more frequently used than raw arrays in Python. Secondly, connections between models are most likely sparse so that using a hash map saves a lot of space overhead. The external activity measured from port A and B is easily accessed using `activity[A][B]`. As ports can only be either input or output ports, `activity[B][A]` will always be zero.

Usage

Quantitative external activity is used by the dynamic allocator. This allocator has the same function as the static allocator (Section 4.3.1): finding a good allocation to start the simulation with. Two major differences exist though:

1. The allocator will have a termination time.

   For the static allocator, this was always required to be zero. A static allocator is thus a special case of the dynamic allocator, in which no activity measurement is required. The name is different though, as a termination time of zero will prevent any activity tracking and perform the allocation immediately If the termination time is different from zero, simulation will run in forced-sequential mode until this termination time is reached. When the GVT passes over the termination time, the

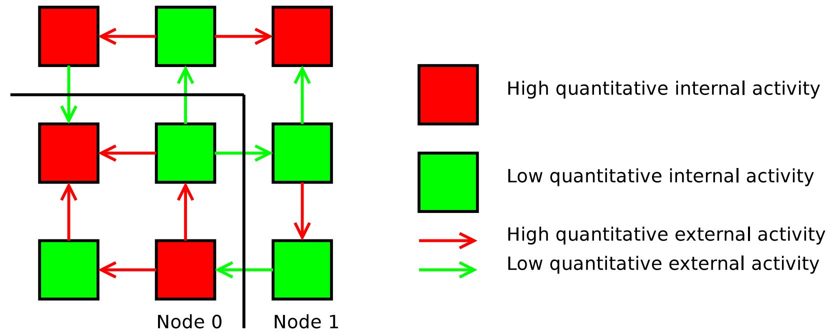

Figure 5.5: Quantitative external example

allocator will be invoked with all gathered activity values. The allocator should return an allocation, just as was the case with the static allocator. This allocation will be used in the rest of the simulation, which will be distributed.

2. Two additional arguments become available for the allocator to use.

   These arguments are the *internal* and *external* activity values that were measured before the termination time. These can be used to make decisions based on the activity of either models, links, or a combination of both.

Measured activity can thus be used by the allocator to base its decisions on. An example is shown in Figure 5.5. At the start of the simulation, all 9 models run locally on the controller. When the termination time is reached, the allocator will be invoked with the activity information that is shown. The allocator has access to both quantitative internal and external activity. It can therefore balance out both the activity of the nodes and make sure that the split happens over connections with low activity. The resulting split is shown too: both nodes have 2 highly active nodes and no connection with high activity passes the network.

Simulation performance itself is not increased, though it is a useful addition to an *activity-aware* simulator. The rationale is mostly identical to that of the static allocator: split up the allocation from the model. From another point of view, its rationale is closely linked to activity relocations: optimize the allocation with activity information.

Normally, the user would have to write the initial allocation manually by embedding this information in the model, or by adding a static allocator. Static allocators have the same weakness as manual relocator: slight modifications to the model require user intervention. A dynamic allocator is able to adapt itself to a variety of circumstances, depending on how general the used allocator was.

During the external activity gathering, the relocator will never be executed as it would interfere with the allocator. As was the case with the static allocators, the allocation will be saved to disk and can be used in subsequent simulations.

Dynamic allocator with only internal activity

Our previous presentation of the dynamic allocator might make it seem mandatory for the allocator to use external activity. This is not the case, as dynamic allocators are passed the internal activity too. As such, it is possible to ignore external activity and focus on internal activity only. While this might seem useless in combination with an internal-activity-only relocator, it offers some distinct advantages:

1. The allocator is only called once at the start of the simulation, thus imposing no overhead during subsequent simulation;

2. The allocator searches for an initial allocation in the assumption that no models are allocated. A relocator can clearly not make this assumption, as it must try to minimize the number of relocations.

A dynamic allocator that only uses internal activity is a worthwhile *complement* to a relocator instead of a replacement. The allocator finds a good starting allocation, on which the relocator can perform small changes to match the situation at hand.

An example of this is in the synthetic activity benchmark in Section 5.4.4. A naive allocation ($\frac{\text{total models}}{\text{total nodes}}$ models per node) is so bad that the relocator is unable to fix this with small changes only. A complete restructuring of the allocation would solve this problem instantly. The relocator does not have any memory though, making it impossible to detect whether it is the first run or not. Should the relocator always try to come up with a completely new allocation, the number of relocations would be immense. Complexity of the relocation will rise as it has to take into account all models. By introducing a dynamic allocator, the allocator will search for a decent allocation once at the beginning of the simulation. This allocation is then used by the relocator as the starting point of relocations.

An example of this is shown in Listing A.5.

	scheduler	**allocator**	**relocator**
static	manually chosen scheduler	static allocator	manual relocator
activity-based	polymorphic scheduler	dynamic allocator	activity relocator

Table 5.2: Activity enhanced components

5.2.4 Overview

Table 5.2 shows an overview of all applications of activity. It is shown that all activity components are also able to function without activity. Activity is only used to enhance these components and makes dynamic rules possible. Without the use of activity, use of these components is tedious for the user and very dependent on the model being simulated.

A scheduler without the use of activity has to be handpicked by the user. This choice might not be too difficult, but it is an additional worry for the user. Should a slightly different model be used with a different number of collisions, the user has to manually change this configuration parameter. Additionally, a handpicked scheduler is static throughout the simulation. Sudden changes in the scheduling operations will not be caught and simulation will continue with a sub-optimal scheduler. An activity-enhanced scheduler monitors the number of collisions and adapts itself throughout the complete simulation.

An allocator without activity information has to resort to static information to pick a decent allocation. Whereas currently all distributed DEVS simulators require such static declarations, this has problems if the model is altered. An activity-enhanced allocator component will monitor the activity up to a certain simulation time. If this time is reached, all activity measures are passed to the allocator. This way, the allocator can make an educated guess on how to distribute the activity from the start. After the initial monitoring phase, no additional overhead is involved.

A manual relocator will blindly follow the static rules passed by the user, such as "move model A to node B at time C". Creating such rules is tedious and becomes almost undoable for models with thousands of models. Simulating a slightly different model might cause all such rules to become invalid. The activity-enhanced relocator is able to monitor the load and allows for dynamic rules. These dynamic rules are encoded in the form of a relocator, which offers all constructs available in Python. Custom activity definitions can be used too, in an attempt to optimize specific metrics.

5.3 Computation

Orthogonal to our different definitions of internal activity (general or custom), a computation interval can be defined. We differentiate between four different types, all of which are supported by PythonPDEVS.

The activity that is passed to the relocator is always based on tracked values. These values are tracked in the interval $[GVT - horizon, GVT]$, even though some nodes can be further ahead in simulated time. This does not form a problem, as only the activity from the specified interval will be passed to the relocator.

Native support exists for both *activity tracking* and *last state*. By writing a domain-specific relocator, it is possible to use *activity prediction* and *future state* too.

5.3.1 Activity tracking

The simplest (and default) method is *activity tracking*. All activity values within the interval $[GVT - horizon, GVT]$ are accumulated into a single value per model. The relocator has access to these accumulated values only and tries to optimize for these values.

A relocator that uses activity tracking creates relocations to optimize the model for the interval that has just happened. It is also takes the horizon as an argument, which is the length of the interval in which it is computed. This is often a good estimate despite the fact that it optimizes for the past. All values are known to be correct, so it is ideal if the activity only changes sporadically. It requires no domain-specific knowledge either, as the values are used as-is.

The reaction time of the relocator is quite slow though: it optimizes for the past, which might be different from the current (and future) situation.

5.3.2 Activity prediction

Activity prediction builds on top of the values found during activity tracking. Basically, results are preprocessed in the relocator to make a guess on future activity. If this guess is correct, relocation will optimize the model for the near future, which is ideal. If the guess is incorrect, relocation will optimize for a non-existing situation. Depending on how wrong it is, the relocations might have nearly no influence on simulation performance, or slow it down. However, simulation will always be correct, even if the prediction is incorrect.

Such predictions clearly require domain-specific knowledge, with as main question: "How will the activity evolve towards

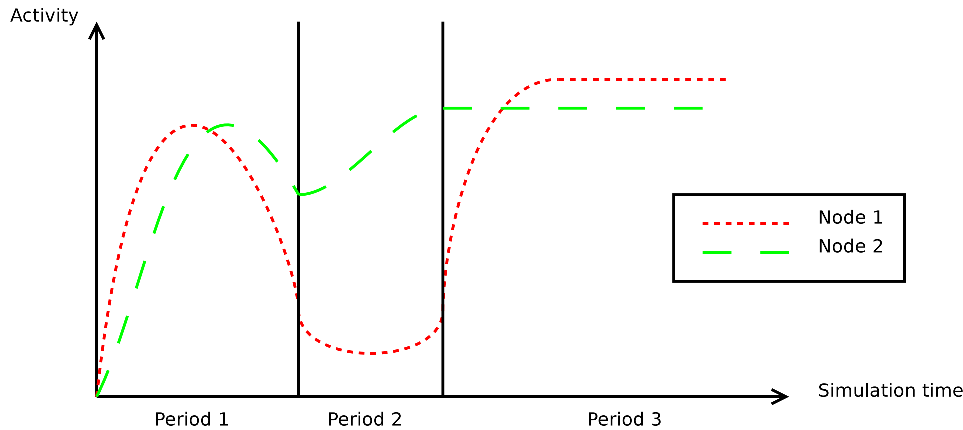

Figure 5.6: Activity prediction to filter out unnecessary relocations

simulation time *GVT* + *horizon*?". The value of the *horizon* is also an unknown factor, as it is based on the pace of the simulation.

It is possible for the custom definition of activity to give predictions of future activity instead of the current activity. It might seem to make activity prediction as a seperate phase useless. However, in-line activity prediction is clearly not desired as it will contain hard-coded assumption and make modularity impossible. Activity prediction has another advantage over such custom definitions: a custom definition only has access to the local state of the model. Some kinds of prediction require global information about all activities to make accurate predictions. An example would be a generator and processor coupled to each other. The processor cannot predict how active the generator will be and cannot make predictions based on its own state alone. With activity prediction, the relocator can detect that the generator is highly active, predicting that the processor will become highly active in the future.

In many situations, nearly no gain is to be expected when using prediction, due to inaccuracy. A situation in which prediction can be useful, is when there are huge fluctuations that are known to be automatically corrected due to the design of the model. An example is shown in Figure 5.6. In period 1, nodes have approximately equal distribution, so no relocations are performed. In period 2 though, node 1 has nearly no activity, whereas node 2 is highly active. A naive relocator that is using activity tracking would try to equalize this by moving some models from node 2 to node 1. However, due to our domain knowledge, we know that node 1 becomes as active as node 2 within foreseeable time. Should the naive relocation be performed in period 2, it would have to be reverted again in period 3, causing 2 relocations and corresponding rollbacks. A relocator with domain knowledge can ignore this temporary problem, knowing that it will be solved automatically.

5.3.3 Last state

The *last state* method is different from activity tracking, in that it bases its computation on the activity in a single state. This single state is the last state of the interval used by activity tracking, thus the state of the model at time GVT.

For some models, accumulation of activity values is unnecessary. An example is the fire spread model: the activity can be defined as the temperature at the state. With accumulation though, this merely returns the sum of all temperatures seen throughout the interval. Using the horizon it is possible to obtain an average temperature in the interval, assuming that the number of transitions is known. When using the last state only, the activity will be the last temperature within the interval. In this case, this is a better estimation of the activity.

Even though only some activity values are actually used, all states still have their activity computed. It is unknown which state will be the one that was the last at the GVT. Moreover, performing activity tracking induces nearly no overhead.

5.3.4 Future state

Of course, the user is free to combine *last state* with *prediction*, where a prediction will be made based on a single state, for example for time $GVT + horizon$. This might prove simpler than activity prediction of a complete interval. As it is the dual of activity prediction, all remarks about making predictions apply too.

Because it makes predictions for a single state in the future, performance is not guaranteed to be increased immediately. However, such optimizations are not such a bad idea knowing that several nodes of the simulation might already be far ahead in simulated time.

5.3.5 Comparison

All of these different methods try to optimize for a different period in simulation time.

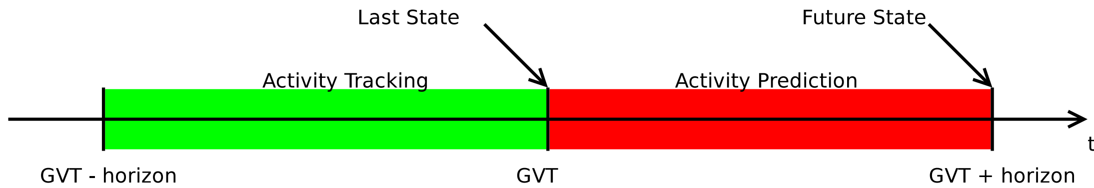

Figure 5.7: Different activity computation methods

Activity tracking optimizes for the past, but has the advantage that its data is known to be correct. Activity prediction optimizes for the future, but has the disadvantage that its data is likely to be incorrect and requires more domain knowledge. Last state optimizes for the current state, for when this information is more helpful. Future state optimizes for a future state, with the disadvantage that it might be incorrect and requires more domain knowledge.

A graphical representation of the interval in time for which is being optimized is seen in Figure 5.7.

5.4 Benchmarks

In this section, several benchmarks will be presented that use the activity concept. As before, both synthetic and realistic models are used. The synthetic models represent artificial models, in which the use of activity has a significant effect. In realistic models, some effect is still seen by using activity, though at a much smaller scale.

Quantitative external activity is not benchmarked, as this is not really used for increased performance. All benchmarks will have a good starting allocation, so there is no use finding such an allocation automatically. Of course, if no good starting allocation was provided, then quantitative external activity will have an impact.

5.4.1 Synthetic qualitative

Qualitative activity is the simplest form of activity, both to implement and to illustrate. The model contains 1000 atomic models, of which only a fixed number of models return a time advance different from ∞. No connections exist between the different models, so an inactive model will stay inactive indefinitely. Simulations are ran with an increasing number of active models, starting from no active models and going to all models being active. This is done using both a normal heap implementation (which schedules all elements) and an activity heap (which only schedules only active models).

Results are shown in Figure 5.8. The plot shows the relative performance of the *total simulation*, which is why the activity heap isn't much faster. If less than 90% of the models are active, using an activity-based scheduler is more efficient. On the other hand, the activity check causes a slight overhead when all models are active, as no models can be skipped. While the overhead is insignificant, it shows that the use of activity is not guaranteed to speed up simulation. Even when only 50% of the models are active, simulation time is only decreased by 5%.

5.4.2 Fire spread

The model from the fire spread benchmark in Section 4.5 is reused. This time, we will increase the size of the model well beyond the speed at which the fire spreads. After a certain number of cells is reached, all subsequent cells will stay inactive forever. It should be clear that this is the case depicted in Figure 5.3, which was used as rationale for qualitative activity in the scheduler.

Again, Quantized DEVS was used to make sure that the fire (and thus the computation) doesn't spread too fast. This yields a lot more inactive models, providing an additional benefit with an activity-aware scheduler.

Figure 5.9 (and a zoomed in view in Figure 5.10) shows how the different schedulers behave. The *heapset* and *activity heap* use activity, whereas the others don't. With only a few models, all of them will be active and colliding, so a list-based scheduler is ideal. However, the activity-based schedulers start flattening out when these additional cells no longer become active, whereas naive schedulers keep increasing. The slight increase in simulation time for the activity-aware schedulers is due to the increased model construction time. This effect is therefore avoided in long running simulations.

Results contain some jitter, even though these are the averages of 20 runs. The time spent during simulation is simply too short for more stable results. This was done on purpose for several reasons:

1. We mainly want to measure the time of the scheduler and not of the simulation itself. Therefore, the model itself should have very fast transition functions, making artificial load an unfit solution.
2. The increased performance caused by activity stems from the number of inactive models. If the termination time is increased, simulation would take longer but the number of cells becoming active would increase too.

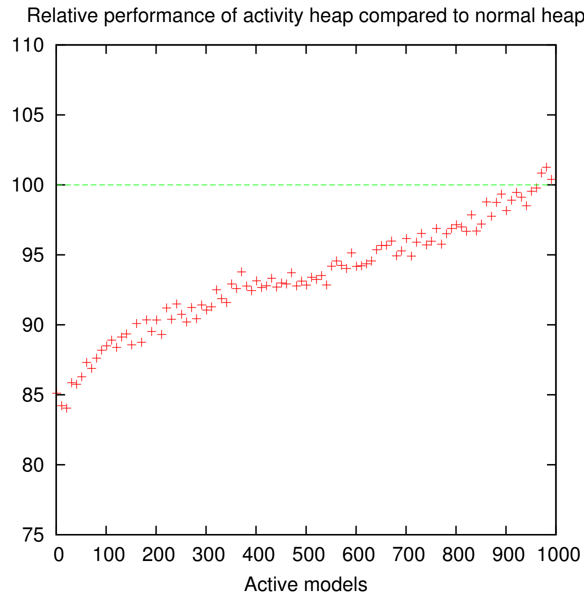

Figure 5.8: Synthetic benchmark for qualitative activity

5.4.3 Polymorphic scheduler

For the polymorphic scheduler benchmark, a model is created with several atomic models, none of them inter-connected. These atomic models come in two varieties: fixed-time and random-time models. All fixed-time models will transition at the same time by returning a fixed time advance. These models will thus all collide with each other. The random-time models return a (pseudo-)random number as time advance, which results in no collisions.

Figure 5.11 shows the results, where the number of colliding models refers to the number of fixed-time models. After the amount of colliding models has grown big enough, the difference is not easily noticeable due to the scale. Figure 5.12 shows a rescaled version of Figure 5.11.

A heap-based scheduler is a lot better when nearly no collisions happen, so the polymorphic scheduler uses this scheduler internally. When the amount of collisions increases, the total simulation time drastically decreases due to the design of the model. In Figure 5.12, it can be seen that the polymorphic scheduler changes its internal data structure as soon as about 30 models collide. A little later, the list-based scheduler becomes the best one, as predicted by the polymorphic scheduler. This difference doesn't seem that drastic as with few collisions, though the relative difference is comparable.

In another benchmark, a model with varying behavior is simulated. After a period of 2000 simulation time units, the behavior changes from many-collisions to few-collisions. After yet another 2000 simulation time units, this behavior switches again from few-collisions to many-collisions. Using a statically defined scheduler is insufficient here, as not a single scheduler is good in both situations. Figure 5.13 shows the results of this benchmark, where the time taken in each period is visualized. It is clear that the polymorphic scheduler is never the fastest, though it is consistently fast by switching the data structure multiple times during the simulation. Overall, the polymorphic scheduler will be the fastest as it adapts its internal data structure.

5.4.4 Synthetic quantitative

Our synthetic benchmark for quantitative internal activity closely resembles the model used in the rationale in Section 5.2.2, shown in Figure 5.4. A ring is built of atomic models, of which only 10% is highly active, whereas all others perform nearly no computation. The activity also migrates throughout the model, so a good static allocation will be of no use. Memoization is enabled to drastically reduce the overhead associated to relocations.

Figure 5.14 shows that activity relocation helps to achieve a decent speedup. This benchmark was done using a dynamic allocator to find an ideal allocation at the start of the simulation. Models that are highly active at the start are placed on separate nodes, whereas lightly active models are combined. Speedup increases slightly when no relocations are performed, though this is only due to the favorable situation created by the dynamic allocator. Adding more nodes has no significant influence on performance without the use of relocations. With relocations, speedup increases almost linearly. This comes as no surprise, as the relocator balances the load, achieving as much speedup as possible.

Figure 5.15 and Figure 5.16 show the same model, but with a varying amount of artificial computation in the transition functions.

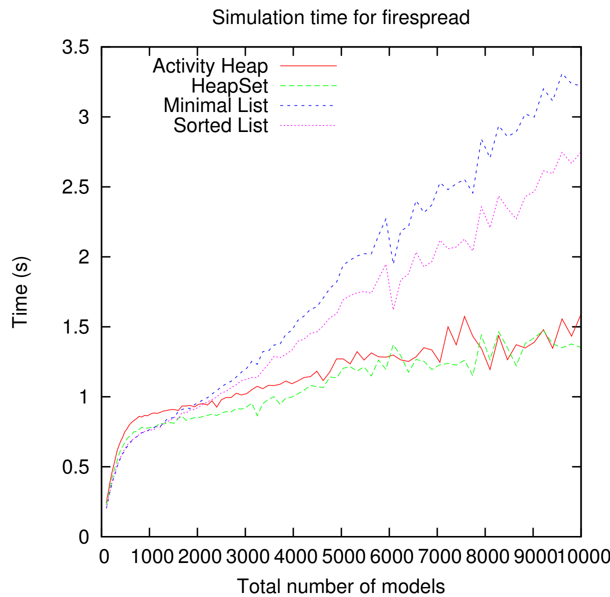

Figure 5.9: Fire spread benchmark for qualitative activity

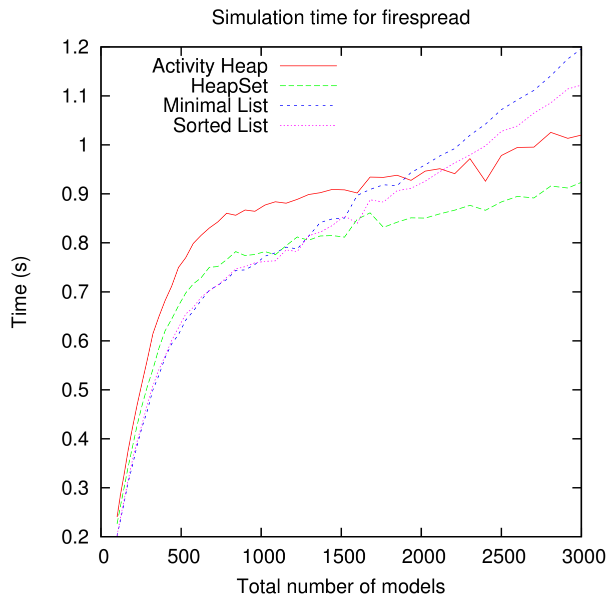

Figure 5.10: Figure 5.9 zoomed in

Normally, increasing the amount of computation in the transition functions increases the speedup. If the model is badly balanced though, nearly no speedup will be noticeable as the most active node will hold back simulation. Even if the load becomes high, unbalanced models will perform equally bad, whereas balanced models will get an additional speedup. This shows that our technique scales well too, both in terms of number of nodes and computation at the transition functions.

Apart from simply showing that activity tracking is faster, we also collected activity measurements from a 3-node simulation run. The variation of the activity throughout the simulation is shown in Figure 5.17 and Figure 5.18. With activity tracking the load of all nodes stays approximately equal, though with the occasional peaks that are quickly resolved. The average activity seems to be about 0.13 per node. Without activity tracking, the activity of all nodes is always very different: at first only node 2 and 3 are active. After some time, node 2 becomes the only active node as all highly actives models are at that node. One node has an activity of 0.45, whereas all other nodes have an activity of (almost) 0. Near the end of the simulation, the activity starts to move to node 1. This causes node 2 to become inactive, but node 1 now becomes highly active.

5.4.5 City Layout

The city layout model is a more realistic model, which was used as a case study in [55]. A small example is shown in Figure 5.20, which contains 2 example routes. A Manhattan-style city layout is constructed using an Python script, shown in Appendix B. Roads are unidirectional and always start and end at intersections. These intersections contain traffic lights, which toggle after a fixed delay. Every road segment contains either a residential building, outputting a car that needs to go to a commercial building, or a commercial building. Each road segment is part of a district, which is the atomic entity used in relocations. A district can be either residential or commercial, determining which type of buildings is constructed on its road segments. Residential and commercial districts are respectively the source and sink of events (cars). The route that is followed is computed in the script, so there is no path finding overhead within the actual simulation. The implementation of this model is the example model in Appendix A.1.

The communication between different road segments is shown in Figure 5.19 and is based on the specification at `http://msdl.cs.mcgill.ca/people/hv/teaching/MoSIS/assignments/DEVS/` and [55]. Residential buildings have the function of a generator and commercial buildings that of a collector. At an intersection, queries are forwarded to the destination if the light is green, or are immediately rejected if the light is red. As soon as the light becomes green again, all previously rejected queries are forwarded immediately.

While this is a realistic model, several problems remain when it is used in distributed simulation:

1. Atomic models have very light computation, as nothing really happens apart from computing the new velocity. This is quite a simple formula, so the achieved speedups are likely to be quite low.

2. The state of the models is relatively complex. States contain cars, queued queries, processing queries, acknowledgments, ... State saving will impose a significant overhead in simulation. The state saving overhead is even higher than the compu-

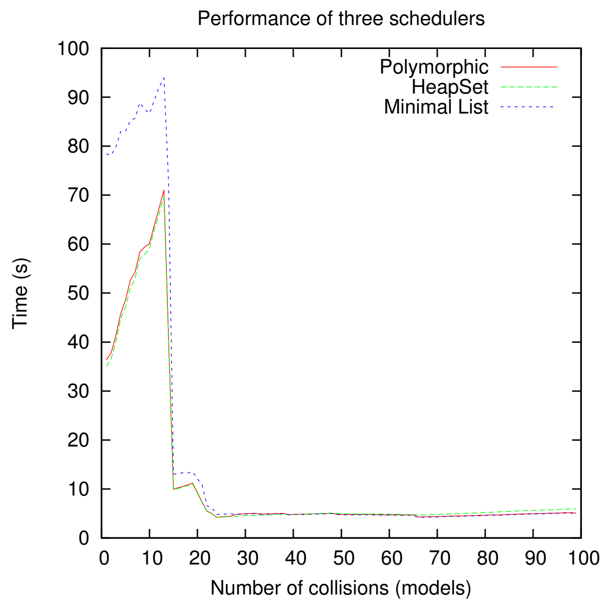

Figure 5.11: Polymorphic scheduler performance

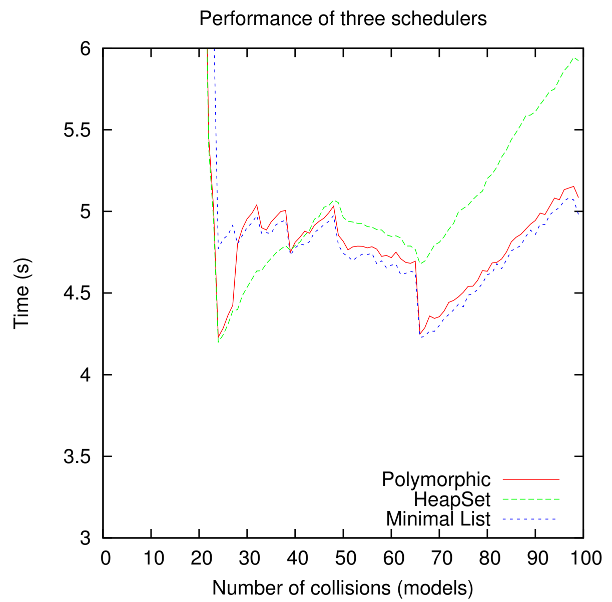

Figure 5.12: Figure 5.11 zoomed in

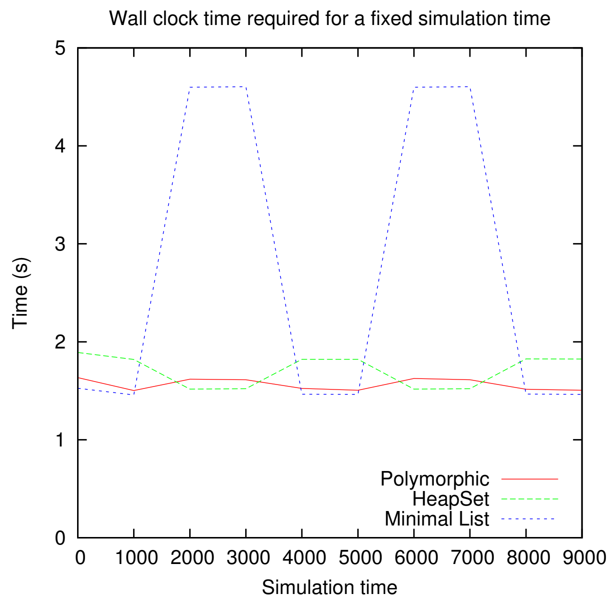

Figure 5.13: Scheduler performance during step-wise simulation

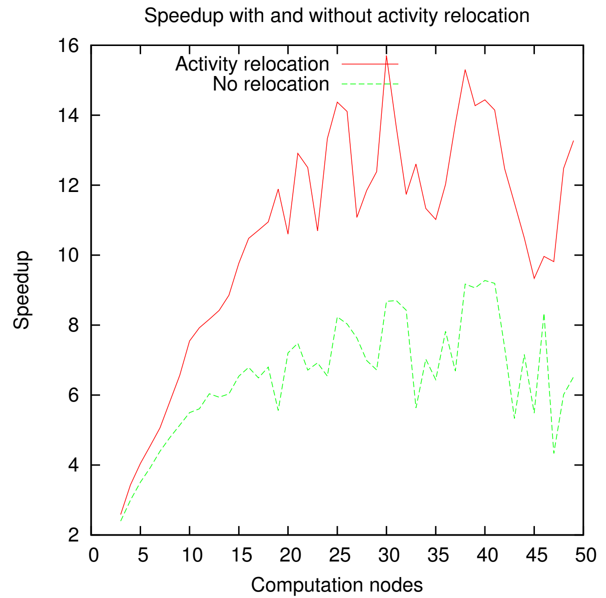

Figure 5.14: Speedup with and without activity tracking for varying number of nodes

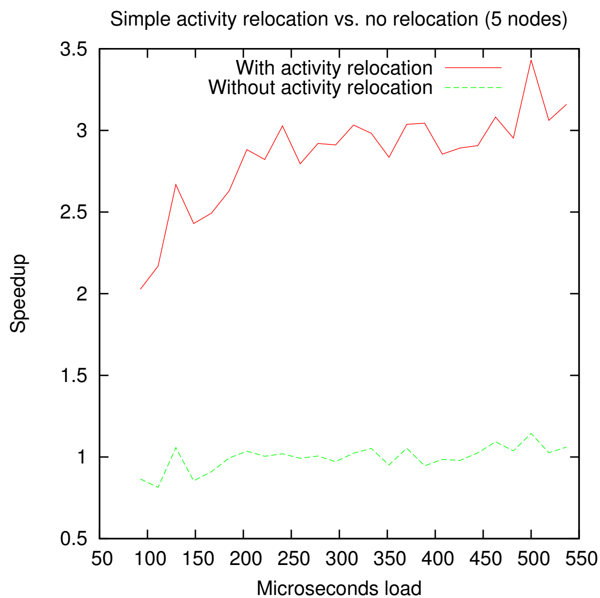

Figure 5.15: Activity tracking on a 5-node synthetic model

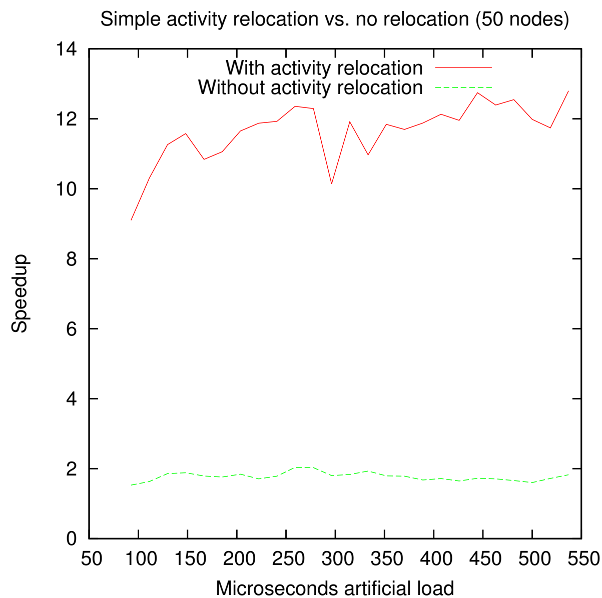

Figure 5.16: Activity tracking on a 50-node synthetic model

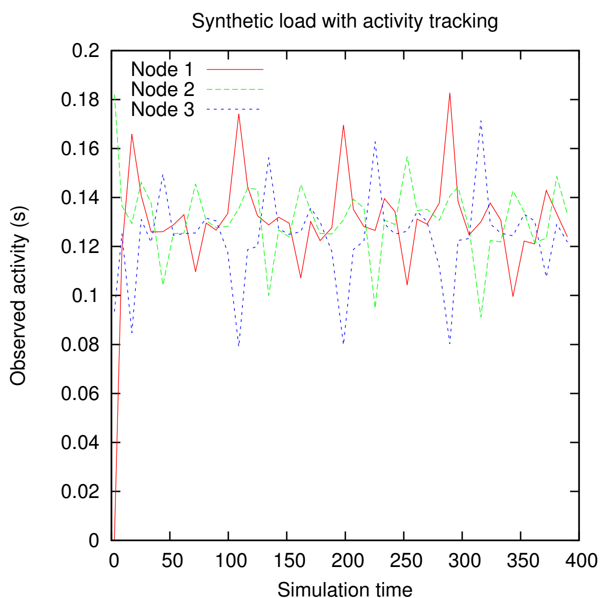

Figure 5.17: Activity distribution with activity tracking (3 nodes)

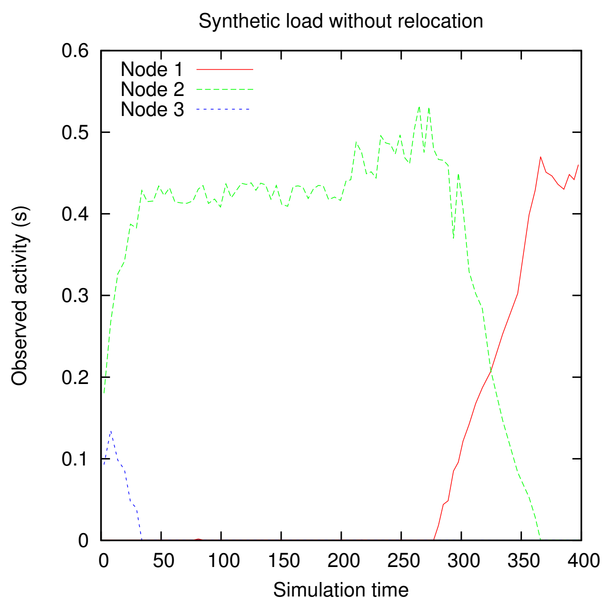

Figure 5.18: Activity distribution without activity tracking (3 nodes)

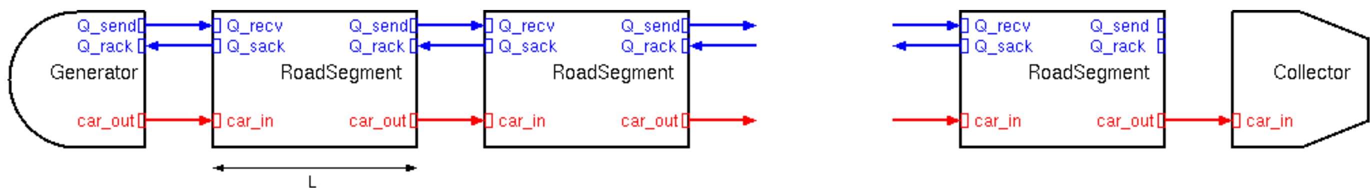

Figure 5.19: A road stretch (source: `http://msdl.cs.mcgill.ca/people/hv/teaching/MoSIS/assignments/DEVS/`)

tation done in the transition functions.

3. Time warp requires as few exchanged messages as possible between different nodes. In the city layout model, all cars are explicitly assumed to pass through all of the nodes. In addition to the cars themselves, there is also a need to pass queries and acknowledgments. This passing has to happen multiple times due to the presence of traffic lights. Furthermore, queries are answered nearly immediate, meaning that the lookahead is very small.

Distributing this model slowed down simulation immensly due to these problems, so our results will only show the absolute execution time. This observation does not invalidate the results observed concerning activity. To circumvent the problem that transition functions have nearly no computational load, some artificial load will be introduced. As the model is too difficult to parallelize, simulating a model that is large enough to offer some possibilities to the relocator would take way too long if it was run on a lot of nodes. We have thus chosen to simulate the model with only 3 nodes (local, with multiple cores) and 5 nodes (distributed, over a network).

Activity is used to relocate districts over different nodes during the actual simulation. Intuitively, activity can have an impact here as the cars (and thus the activity) move through the model. At the start of the simulation, the residential districts will be highly active, whereas the commercial districts will be inactive. Near the end of the simulation, the commercial districts will become highly active, whereas the residential districts will become inactive.

Figure 5.21 and Figure 5.22 show the results for a simulation with 3 and 5 nodes respectively. Four different methods were compared:

1. **No relocation**: The basic case without any activity information being used.

2. **Activity Tracking**: Activity tracking is the basic activity method. The wall clock time spent in the transition functions is used as the activity of the model. Relocations try to balance this time spent by migrating complete districts over to other nodes.

3. **Custom activity tracking**: Custom activity tracking uses the number of cars in the district as its internal activity metric. This is a deterministic value, which is of great help compared to the unreliable wall clock time. As this is based on a single state, the last state method is used instead of normal activity tracking.

4. **Custom activity prediction**: Custom activity prediction is almost the same as custom activity tracking, but there is a small preprocessing step before the activity values are used. This preprocessing will try to guess the activity at time `GVT + horizon` instead of using the activity at time `GVT`. Our prediction is very basic: it is assumed that 20% of the cars will have exited the district. In commercial sections, we further assume that there is a chance that the car will have arrived and be removed from the simulation.

From the results, it is clear that both custom activity tracking and custom activity prediction are always faster than no relocation. Both custom methods are nearly identical and the difference between both is neglible. The main cause is that prediction is unable to ignore inequally distributed loads, as was our main rationale.

The difference between activity tracking and custom activity tracking that is visible with 5 nodes is an important observation. As simulation takes a lot of time, the period between different GVT computations shrinks if even more load is added. This period between the GVT is equal to the horizon, which is therefore rather small. The horizon might be so small that a model only transitions a few times, making the measured activity statistically irrelevant. As models do not transition at exactly the same time, some models will transition more frequently than others in the specified horizon. These models will have an higher activity in the horizon, but the models will actually not be more active. This problem is thus caused by a too small horizon. While increasing the horizon will alleviate this problem, it will cause slower reaction times. Custom activity methods use the last state only, which is invulnerable to this problem.

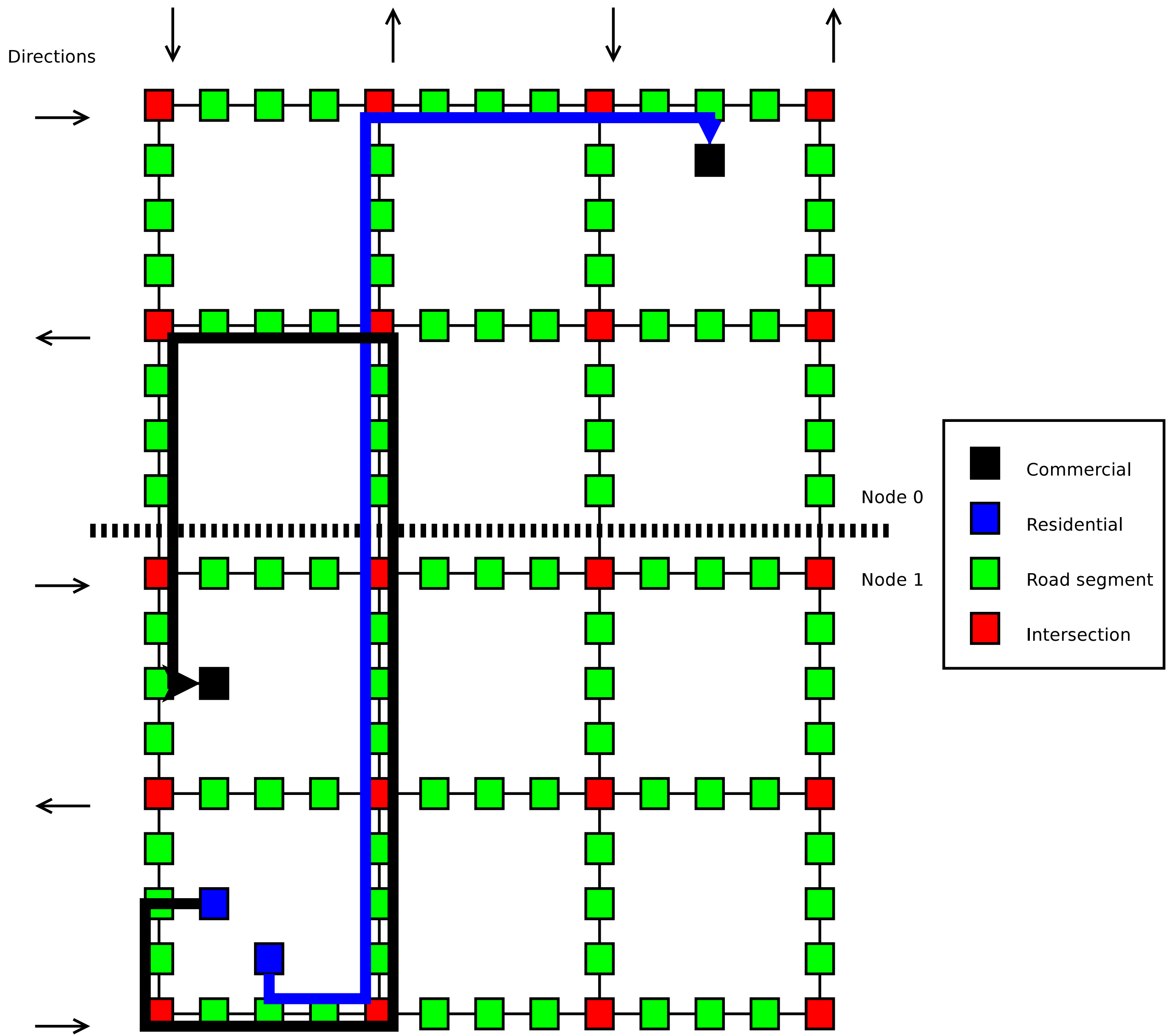

Figure 5.20: Example city layout model for 2 cars over 2 nodes

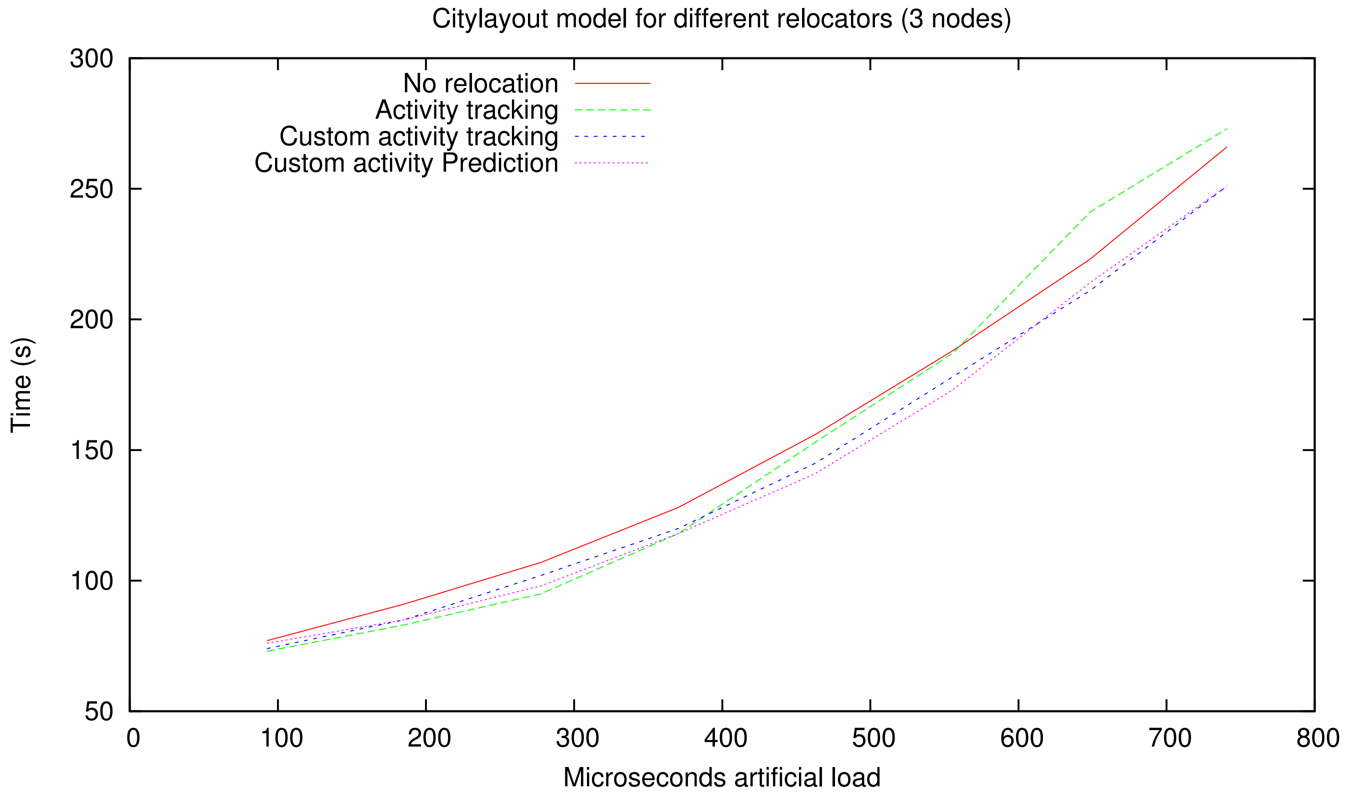

Figure 5.21: City layout benchmark for quantitative internal activity (3 nodes)

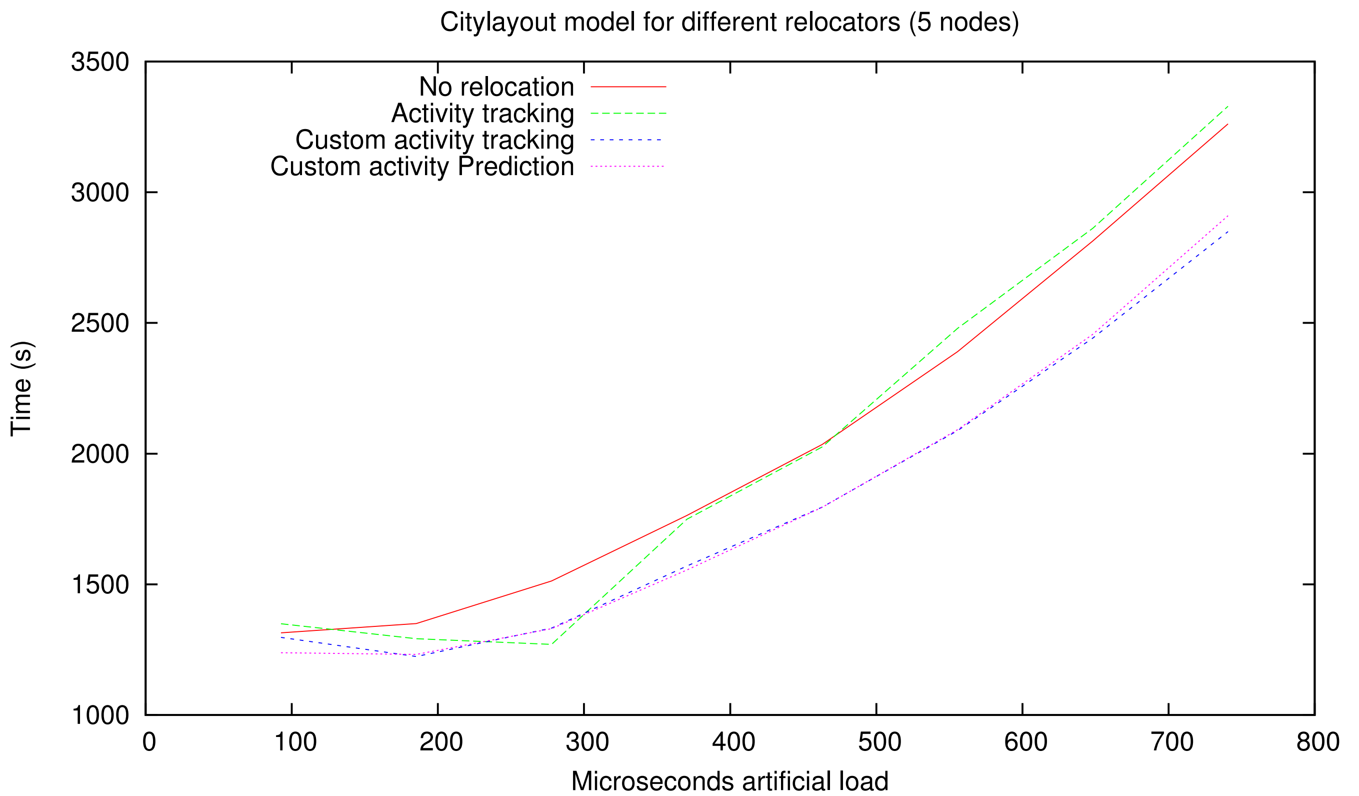

Figure 5.22: City layout benchmark for quantitative internal activity (5 nodes)

6 Conclusion

In this thesis we elaborated on the PythonPDEVS simulation kernel. In Chapter 1, a small introduction to several DEVS formalisms and two distributed synchronization protocols was given.

Chapter 2 presented the high-level design of the kernel by presenting several UML diagrams, indicating its modular design. Additionally, we discussed some slight modifications that were necessary to the GVT algorithm. One of our other contributions that is related to design, is the possibility to simulate a stripped down version of the kernel with itself.

All of our functionality was discussed in Chapter 3. Each feature had a discussion about how it is used, its rationale, and its implementation. Some of this functionality is unique to PythonPDEVS, whereas others are inspired by the work of others. Even other features are implemented by other groups, but use PythonPDEVS as their simulation kernel.

The performance side of the kernel was presented in Chapter 4. Optimizations for different situations were presented, again with explanation about how it is used, rationale and implementation. Domain-specific hints can be added to the simulator to make it perform faster in nearly every situation. These performance claims were supported by empirical results where applicable. More general benchmarks followed and finally a comparison to adevs was performed. In terms of performance, PythonPDEVS is a worthy opponent of adevs.

Finally, Chapter 5 presented the main contribution of this thesis: adding (optional) activity to the PythonPDEVS simulation kernel. Several definitions were discussed and our usage of each of them is presented. First there was qualitative activity, which was used to enhance the scheduler. Quantitative activity was split up in quantitative internal activity, used for relocations, and quantitative external activity, used for allocations. All of these enhancements were possible without activity, though activity allowed for automation. These optimizations were again benchmarked in both a synthetic and a more realistic model.

In conclusion, PythonPDEVS was written as an efficient simulation kernel, without compromising on usability and functionality.

The documentation that is present within every PythonPDEVS distribution is an additional source of information. Its main focus is on the actual usage of all these features and some examples.

6.1 Recap

Our algorithms have been shown to give a significant speedup in the sequential simulation of a wide variety of models. These modifications, and in particular the use of activity in the scheduler, make it possible to achieve a lower complexity than even the most efficient simulators. Furthermore, such optimizations to the scheduler are compatible with every implemented feature.

With the use of distributed simulation, activity can be used for finding a good initial allocation, or for optimizing this allocation using on-the-fly load balancing.

While all of these optimizations would also be possible without activity, including activity makes it more flexible and requires less user intervention. In this way, activity is not inherently linked to the possibilities for improved performance, though makes it much easier to exploit.

Our approach has the major advantage that it allows users to increase simulation performance by providing some hints about the model being simulated. These hints range from fairly trivial (message copy functions) to more advanced (migration component), offering something for every kind of user and domain.

The major disadvantage is that it requires user information, which can be wrong, depending on the skills of the user. Luckily, most of these methods can cope with wrong information in the sense that simulation will still be correct, though with lower efficiency.

6.2 Contributions

The following contributions were made to the design and implementation of DEVS simulators:

1. PythonPDEVS's synchronization algorithms were modelled and simulated using PythonPDEVS itself.

   This allowed the distributed simulation of a distributed simulation. Due to deterministic simulation runs, several bugs could be found and fixed more easily. Performance and behavior in different situations can be assessed easily, even if the target platform is unavailable.

2. PythonPDEVS is a family of simulators.

   A lot of (mostly orthogonal) features are supported. There is support for realtime simulation, as-fast-as-possible simulation, Classic DEVS, Parallel DEVS, optional dynamic structure functionality, . . . All of these options require a flexible and modular scheduler.

3. The advantages of a modular scheduler were introduced.

   Modular schedulers offer an easy way to increase the performance of many simulators. A polymorphic scheduler was presented, which can find the ideal scheduler automatically throughout the simulation.

4. The use of an allocator was introduced.

   Most DEVS simulators force the user to specify the initial distribution in the model itself. PythonPDEVS on the other hand, makes it possible to extract this knowledge and write a specifically designed component: the allocator. Both static and dynamic versions of an allocator were introduced.

5. PythonPDEVS supports the processing of domain-specific hints.

   Domain-specific hints are used to take specific shortcuts in the simulation, offering higher performance. Some possible optimizations are discussed that are able to use these hints. Different techniques were benchmarked to show that these hints can really have an influence on simulation performance.

6. Activity was introduced in the simulation kernel, together with domain-specific activity definitions.

   In addition to simply tracking the activity, our activity is automatically used during simulation to achieve simulation speedups. Some more advanced methods were also made possible, like custom definitions of activity and the prediction of activity.

7. We add the concept of last state activity computation.

   Normal activity tracking measures the activity over a specified interval, called the horizon. With last state computation, only the current state is used for the relocators. This method is only useful when using custom activity definitions though.

Parts of this thesis were published in [74, 76, 75].

6.3 Future work

Future work is possible in most directions:

- **Design**
  Our PythonPDEVS model only supports a stripped-down version of the simulator, as we only focused on the synchronization and relocation protocol. In future work this model can be extended to the complete simulation kernel with all of its features. Some of the missing features are modular tracers, modular schedulers and support for different DEVS formalisms.
- **Functionality**
  Some more functionality could be added, mainly with distributed simulation. These additional features are not really that useful in the general sense due to their huge impact on performance. While it might not be necessary from a performance point of view, orthogonality of the features would be a lot better. Examples of unsupported combinations are realtime distributed simulation and distributed dynamic structure DEVS.
  More functionality might be possible too, like the integration of agent-based simulation within a general DEVS simulator.

- **Usability**
  Future work here is mainly done by the DEVSimPy project, which uses PythonPDEVS as its simulation kernel. Some remaining problems are how this distributed simulation and the new interface will be implemented and made usable from inside the IDE. Other future work might update the $AToM^3$ DEVS formalism to create PythonPDEVS compliant models. All new functionality should also be made available in these interfaces.
- **Performance**
  PyPy seems to be one of the most important bottlenecks. Interpreted code is still slower than normal CPython and JIT compilation happens at simulation time. The garbage collection phase is also seen to be quite time consuming.
  More advanced simulation algorithms might be possible too. A plethora of possible speedups to the time warp protocol are presented in the literature.
  Another direction might be to port PythonPDEVS to C++ or another compiled and high-performance programming language. This makes a fair comparison to adevs possible, though at the cost of the tedious work of porting PythonPDEVS to a much stricter language. For an implementation in C++, template metaprogramming[68] would become an option.
  Using the power of the GPU for massive parallelism[63] is another option.
- **Activity**
  Different applications of activity are still being considered by several research groups. Some other work might be beneficial to include in the PythonPDEVS kernel by default, like analytical estimates of activity[43] which could be done before simulation starts. Other directions include further optimization or additional exploitation of activity in different parts of the simulation kernel.

Examples

Previous chapters introduced several enhancements to the model. No explanation on how this is actually implemented is shown though, for which it is included here. For more explanations about the practical use of these features, we refer to the Python-PDEVS documentation.

A.1 Model

This is the exact code that is used for the city layout benchmark in Section 5.4.5. Explanation about the purpose of the model is provided in the same section. It is also used as a running example for all domain-specific optimizations. This code includes code for several extensions to the DEVS formalism, such as:

1. memoization in the `__eq__` method
2. tracing in the `__str__` method
3. custom message copy in the `copy` method for events
4. custom state saving in the `copy` method for states
5. custom activity in the `preActivityCalculation` and `postActivityCalculation` methods

```
import sys
sys.path.append("../../../src/")
from infinity import *
from DEVS import *
import random

north = 0
east = 1
south = 2
west = 3
vertical = "02"
horizontal = "13"
dir_to_int = {'n': north, 'e': east, 's': south, 'w': west}
int_to_dir = {north: 'n', east: 'e', south: 's', west: 'w'}

class Car:
    def __init__(self, ID, v, v_pref, dv_pos_max, dv_neg_max, departure_time):
        self.ID = ID
        self.v_pref = v_pref
        self.dv_pos_max = dv_pos_max
        self.dv_neg_max = dv_neg_max
        self.departure_time = departure_time
        self.distance_travelled = 0
        self.remaining_x = 0
        self.v = v

    def __eq__(self, other):
        return (self.ID == other.ID and
            self.v_pref == other.v_pref and
            self.dv_pos_max == other.dv_pos_max and
            self.dv_neg_max == other.dv_neg_max and
            self.departure_time == other.departure_time and
            self.distance_travelled == other.distance_travelled and
            self.remaining_x == other.remaining_x and
            self.v == other.v)

    def __str__(self):
        return "Car:␣ID␣=␣" + str(self.ID)

    def copy(self):
        car = Car(self.ID, self.v, self.v_pref, self.dv_pos_max, self.dv_neg_max, self.departure_time)
        car.distance_travelled = self.distance_travelled
        car.remaining_x = self.remaining_x
        car.path = list(self.path)
        return car

class Query:
    def __init__(self, ID):
        self.ID = ID
        self.direction = ''

    def __str__(self):
        return "Query:␣ID␣=␣" + str(self.ID)

```

```
55      def __eq__(self, other):
56          return (self.ID == other.ID and self.direction == other.direction)
57
58      def copy(self):
59          query = Query(self.ID)
60          query.direction = self.direction
61          return query
62
63  class QueryAck:
64      def __init__(self, ID, t_until_dep):
65          self.ID = ID
66          self.t_until_dep = t_until_dep
67
68      def __str__(self):
69          return "Query␣Ack:␣ID␣=␣" + str(self.ID) + ",␣t_until_dep␣=␣%.3f" % self.t_until_dep
70
71      def __eq__(self, other):
72          return (self.ID == other.ID and self.t_until_dep == other.t_until_dep)
73
74      def copy(self):
75          return QueryAck(self.ID, self.t_until_dep)
76
77  class BuildingState:
78      def __init__(self, IAT_min, IAT_max, path, name):
79          self.currentTime = 0
80          from randomGenerator import RandomGenerator
81          seed = random.random()
82          self.randomGenerator = RandomGenerator(seed)
83          self.send_query_delay = self.randomGenerator.uniform(IAT_min, IAT_max)
84
85          self.send_car_delay = INFINITY
86          self.path = path
87          if path == []:
88              self.send_query_delay = INFINITY
89
90          self.name = name
91          self.send_query_id = int(name.split("_", 1)[1].split("_")[0]) * 1000 + int(name.split("_", 1)[1].
                split("_")[1])
92          self.send_car_id = self.send_query_id
93          self.next_v_pref = 0
94          self.sent = 0
95
96      def copy(self):
97          new = BuildingState(0, 0, list(self.path), self.name)
98          new.currentTime = self.currentTime
99          new.send_query_delay = self.send_query_delay
100         new.send_car_delay = self.send_car_delay
101         new.send_query_id = self.send_query_id
102         new.send_car_id = self.send_car_id
103         new.next_v_pref = self.next_v_pref
104         new.randomGenerator = self.randomGenerator.copy()
105         new.sent = self.sent
106         return new
107
108     def __str__(self):
109         if self.path != []:
110             return "Residence:␣send_query_delay␣=␣%.3f,␣send_query_id␣=␣%s,␣send_car_delay␣=␣%.3f,␣
                    send_car_id␣=␣%s" % (self.send_query_delay, self.send_query_id, self.send_car_delay, self.
                    send_car_id)
111         else:
112             return "Commercial:␣waiting..."
113
114 class Building(AtomicDEVS):
115     def __init__(self, generator, district, path = [], IAT_min = 100, IAT_max = 100, v_pref_min = 15,
            v_pref_max = 15, dv_pos_max = 15, dv_neg_max = 150, name="Building"):
116         # parent class constructor
117         AtomicDEVS.__init__(self, name)
118
119         # copy of arguments
120         self.IAT_min = IAT_min
121         self.IAT_max = IAT_max
122         self.v_pref_min = v_pref_min
```

```
123         self.v_pref_max = v_pref_max
124         self.dv_pos_max = dv_pos_max
125         self.dv_neg_max = dv_neg_max
126 
127         # output ports
128         self.q_sans = self.addOutPort(name="q_sans")
129         self.q_send = self.addOutPort(name="q_send")
130         self.exit = self.addOutPort(name="exit")
131         self.Q_send = self.q_send
132         self.car_out = self.exit
133 
134         # input ports
135         self.q_rans = self.addInPort(name="q_rans")
136         self.Q_rack = self.q_rans
137         self.entry = self.addInPort(name="entry")
138 
139         # set the state
140         self.state = BuildingState(IAT_min, IAT_max, path, name)
141         self.state.next_v_pref = self.state.randomGenerator.uniform(self.v_pref_min, self.v_pref_max)
142         self.send_max = 1
143 
144         self.district = district
145 
146     def intTransition(self):
147         mintime = self.timeAdvance()
148         self.state.currentTime += mintime
149 
150         self.state.send_query_delay -= mintime
151         self.state.send_car_delay -= mintime
152 
153         if self.state.send_car_delay == 0:
154             self.state.send_car_delay = INFINITY
155             self.state.send_car_id = self.state.send_query_id
156             self.state.next_v_pref = self.state.randomGenerator.uniform(self.v_pref_min, self.v_pref_max)
157         elif self.state.send_query_delay == 0:
158             self.state.send_query_delay = INFINITY
159 
160         return self.state
161 
162     def outputFnc(self):
163         mintime = self.timeAdvance()
164         currentTime = self.state.currentTime + self.timeAdvance()
165         outputs = {}
166 
167         if self.state.send_car_delay == mintime:
168             v_pref = self.state.next_v_pref
169             car = Car(self.state.send_car_id, 0, v_pref, self.dv_pos_max, self.dv_neg_max, currentTime)
170             car.path = self.state.path
171             outputs[self.car_out] = [car]
172         elif self.state.send_query_delay == mintime:
173             query = Query(self.state.send_query_id)
174             outputs[self.Q_send] = [query]
175         return outputs
176 
177     def timeAdvance(self):
178         return min(self.state.send_query_delay, self.state.send_car_delay)
179 
180     def extTransition(self, inputs):
181         self.state.currentTime += self.elapsed
182         self.state.send_query_delay -= self.elapsed
183         self.state.send_car_delay -= self.elapsed
184         queryAcks = inputs.get(self.Q_rack, [])
185         for queryAck in queryAcks:
186           if self.state.send_car_id == queryAck.ID and (self.state.sent < self.send_max):
187             self.state.send_car_delay = queryAck.t_until_dep
188             if queryAck.t_until_dep < 20000:
189                 self.state.sent += 1
190                 if self.state.sent < self.send_max:
191                     self.state.send_query_delay = self.randomGenerator.uniform(self.IAT_min, self.IAT_max)
192                     self.state.send_query_id += 1000000
193         return self.state
194 
```

```
195     def preActivityCalculation(self):
196         return None
197
198     def postActivityCalculation(self, _):
199         return 0 if self.state.send_car_delay == float('inf') else 1
200
201 class Residence(Building):
202     def __init__(self, path, district, name = "Residence", IAT_min = 100, IAT_max = 100, v_pref_min = 15,
            v_pref_max = 15, dv_pos_max = 15, dv_neg_max = 15):
203         Building.__init__(self, True, district, path=path, name=name)
204
205 class CommercialState(object):
206     def __init__(self, car):
207         self.car = car
208
209     def __str__(self):
210         return "CommercialState"
211
212     def copy(self):
213         return CommercialState(self.car)
214
215 class Commercial(Building):
216     def __init__(self, district, name="Commercial"):
217         Building.__init__(self, False, district, name=name)
218         self.state = CommercialState(None)
219         self.toCollector = self.addOutPort(name="toCollector")
220
221     def extTransition(self, inputs):
222         return CommercialState(inputs[self.entry][0])
223
224     def intTransition(self):
225         return CommercialState(None)
226
227     def outputFnc(self):
228         return {self.toCollector: [self.state.car]}
229
230     def timeAdvance(self):
231         if self.state.car is None:
232             return INFINITY
233         else:
234             return 0.0
235
236     def preActivityCalculation(self):
237         return None
238
239     def postActivityCalculation(self, _):
240         return 0
241
242 class RoadSegmentState():
243     def __init__(self):
244         self.cars_present = []
245         self.query_buffer = []
246         self.deny_list = []
247         self.reserved = False
248         self.send_query_delay = INFINITY
249         self.send_query_id = None
250         self.send_ack_delay = INFINITY
251         self.send_ack_id = None
252         self.send_car_delay = INFINITY
253         self.send_car_id = None
254         self.last_car = None
255
256     def __eq__(self, other):
257         if not (self.send_query_delay == other.send_query_delay and
258               self.send_ack_delay == other.send_ack_delay and
259               self.send_car_delay == other.send_car_delay and
260               self.send_query_id == other.send_query_id and
261               self.send_ack_id == other.send_ack_id and
262               self.send_car_id == other.send_car_id and
263               self.last_car == other.last_car and
264               self.reserved == other.reserved):
265             return False
```

```
266         if self.query_buffer != other.query_buffer:
267             return False
268         if len(self.cars_present) != len(other.cars_present):
269             return False
270         for c1, c2 in zip(self.cars_present, other.cars_present):
271             if c1 != c2:
272                 return False
273         if len(self.deny_list) != len(other.deny_list):
274             return False
275         for q1, q2 in zip(self.deny_list, other.deny_list):
276             if q1 != q2:
277                 return False
278         return True
279 
280     def copy(self):
281         new = RoadSegmentState()
282         new.cars_present = [c.copy() for c in self.cars_present]
283         new.query_buffer = list(self.query_buffer)
284         new.deny_list = [q.copy() for q in self.deny_list]
285         new.reserved = self.reserved
286         new.send_query_delay = self.send_query_delay
287         new.send_query_id = self.send_query_id
288         new.send_ack_delay = self.send_ack_delay
289         new.send_ack_id = self.send_ack_id
290         new.send_car_delay = self.send_car_delay
291         new.send_car_id = self.send_car_id
292         new.last_car = self.last_car
293         return new
294 
295     def __str__(self):
296         string = "Road␣segment:␣cars_present␣=␣["
297         for i in self.cars_present:
298             string += str(i.ID) + ",␣"
299         return string + "]␣,␣send_query_delay␣=␣%.3f,␣send_ack_delay␣=␣%.3f,␣send_car_delay␣=␣%.3f,␣
              send_ack_id␣=␣%s" % (self.send_query_delay, self.send_ack_delay, self.send_car_delay, self.
              send_ack_id)
300 
301 class RoadSegment(AtomicDEVS):
302     def __init__(self, district, load, l = 100.0, v_max = 18.0, observ_delay = 0.1, name = "RoadSegment"):
303         AtomicDEVS.__init__(self, name)
304 
305         # arguments
306         self.l = float(l)
307         self.v_max = v_max
308         self.observ_delay = observ_delay
309         self.district = district
310         self.load = load
311 
312         # in-ports
313         self.q_rans = self.addInPort(name="q_rans")
314         self.q_recv = self.addInPort(name="q_recv")
315         self.car_in = self.addInPort(name="car_in")
316         self.entries = self.addInPort(name="entries")
317         self.q_rans_bs = self.addInPort(name="q_rans_bs")
318         self.q_recv_bs = self.addInPort(name="q_recv_bs")
319         # compatibility bindings...
320         self.Q_recv = self.q_recv
321         self.Q_rack = self.q_rans
322 
323         # out-ports
324         self.q_sans = self.addOutPort(name="q_sans")
325         self.q_send = self.addOutPort(name="q_send")
326         self.car_out = self.addOutPort(name="car_out")
327         self.exits = self.addOutPort(name="exits")
328 
329         self.q_sans_bs = self.addOutPort(name="q_sans_bs")
330         # compatibility bindings...
331         self.Q_send = self.q_send
332         self.Q_sack = self.q_sans
333 
334         self.state = RoadSegmentState()
335 
```

```
336     def extTransition(self, inputs):
337         queries = inputs.get(self.Q_recv, [])
338         queries.extend(inputs.get(self.q_recv_bs, []))
339         cars = inputs.get(self.car_in, [])
340         cars.extend(inputs.get(self.entries, []))
341         acks = inputs.get(self.Q_rack, [])
342         acks.extend(inputs.get(self.q_rans_bs, []))
343 
344         self.state.send_query_delay -= self.elapsed
345         self.state.send_ack_delay -= self.elapsed
346         self.state.send_car_delay -= self.elapsed
347 
348         for query in queries:
349             if (not self.state.reserved) and not (len(self.state.cars_present) > 1 or (len(self.state.
                    cars_present) == 1 and self.state.cars_present[0].v == 0)):
350                 self.state.send_ack_delay = self.observ_delay
351                 self.state.send_ack_id = query.ID
352                 self.state.reserved = True
353             else:
354                 self.state.query_buffer.append(query.ID)
355                 self.state.deny_list.append(query)
356                 if self.state.send_ack_delay == INFINITY:
357                     self.state.send_ack_delay = self.observ_delay
358                     self.state.send_ack_id = query.ID
359             self.state.last_car = query.ID
360         for car in self.state.cars_present:
361             car.remaining_x -= self.elapsed * car.v
362 
363         for car in cars:
364             self.state.last_car = None
365             car.remaining_x = self.l
366             self.state.cars_present.append(car)
367 
368             if len(self.state.cars_present) != 1:
369                 for other_car in self.state.cars_present:
370                     other_car.v = 0
371                 self.state.send_query_delay = INFINITY
372                 self.state.send_ack_delay = INFINITY
373                 self.state.send_car_delay = INFINITY
374             else:
375                 self.state.send_query_delay = 0
376                 self.state.send_query_id = car.ID
377                 if self.state.cars_present[-1].v == 0:
378                     t_to_dep = INFINITY
379                 else:
380                     t_to_dep = max(0, self.l/self.state.cars_present[-1].v)
381                 self.state.send_car_delay = t_to_dep
382                 self.state.send_car_id = car.ID
383 
384         for ack in acks:
385             if (len(self.state.cars_present) == 1) and (ack.ID == self.state.cars_present[0].ID):
386                 car = self.state.cars_present[0]
387                 t_no_col = ack.t_until_dep
388                 v_old = car.v
389                 v_pref = car.v_pref
390                 remaining_x = car.remaining_x
391 
392                 if t_no_col + 1 == t_no_col:
393                     v_new = 0
394                     t_until_dep = INFINITY
395                 else:
396                     v_ideal = remaining_x / max(t_no_col, remaining_x / min(v_pref, self.v_max))
397                     diff = v_ideal - v_old
398                     if diff < 0:
399                         if -diff > car.dv_neg_max:
400                             diff = -car.dv_neg_max
401                     elif diff > 0:
402                         if diff > car.dv_pos_max:
403                             diff = car.dv_pos_max
404                     v_new = v_old + diff
405                     if v_new == 0:
406                       t_until_dep = INFINITY
```

```
407                     else:
408                       t_until_dep = car.remaining_x / v_new
409
410                 car.v = v_new
411
412                 t_until_dep = max(0, t_until_dep)
413                 if t_until_dep > self.state.send_car_delay and self.state.last_car is not None:
414                     self.state.send_ack_delay = self.observ_delay
415                     self.state.send_ack_id = self.state.last_car
416                 self.state.send_car_delay = t_until_dep
417                 self.state.send_car_id = ack.ID
418
419                 if t_until_dep == INFINITY:
420                   self.state.send_query_id = ack.ID
421                 else:
422                   self.state.send_query_id = None
423                 self.state.send_query_delay = INFINITY
424         return self.state
425
426     def intTransition(self):
427         for _ in range(self.load):
428             pass
429         mintime = self.mintime()
430         self.state.send_query_delay -= mintime
431         self.state.send_ack_delay -= mintime
432         self.state.send_car_delay -= mintime
433         if self.state.send_ack_delay == 0:
434             self.state.send_ack_delay = INFINITY
435             self.state.send_ack_id = None
436
437         elif self.state.send_query_delay == 0:
438             # Just sent a query, now deny all other queries and wait until the current car has left
439             self.state.send_query_delay = INFINITY
440             self.state.send_query_id = None
441         elif self.state.send_car_delay == 0:
442             self.state.cars_present = []
443             self.state.send_car_delay = INFINITY
444             self.state.send_car_id = None
445
446             # A car has left, so we can answer to the first other query we received
447             if len(self.state.query_buffer) != 0:
448                 self.state.send_ack_delay = self.observ_delay
449                 self.state.send_ack_id = self.state.query_buffer.pop()
450             else:
451                 # No car is waiting for this segment, so 'unreserve' it
452                 self.state.reserved = False
453         if self.state.send_ack_id == None and len(self.state.deny_list) > 0:
454             self.state.send_ack_delay = self.observ_delay
455             self.state.send_ack_id = self.state.deny_list[0].ID
456             self.state.deny_list = self.state.deny_list[1:]
457
458         return self.state
459
460     def outputFnc(self):
461         outputs = {}
462         mintime = self.mintime()
463         if self.state.send_ack_delay == mintime:
464             ackID = self.state.send_ack_id
465             if len(self.state.cars_present) == 0:
466                 t_until_dep = 0
467             elif (self.state.cars_present[0].v) == 0:
468                 t_until_dep = INFINITY
469             else:
470                 t_until_dep = self.l / self.state.cars_present[0].v
471             ack = QueryAck(ackID, t_until_dep)
472             outputs[self.Q_sack] = [ack]
473             outputs[self.q_sans_bs] = [ack]
474         elif self.state.send_query_delay == mintime:
475             query = Query(self.state.send_query_id)
476             if self.state.cars_present[0].path != []:
477                 query.direction = self.state.cars_present[0].path[0]
478                 outputs[self.Q_send] = [query]
```

```
479         elif self.state.send_car_delay == mintime:
480             car = self.state.cars_present[0]
481             car.distance_travelled += self.l
482             if len(car.path) == 0:
483                 outputs[self.exits] = [car]
484             else:
485                 outputs[self.car_out] = [car]
486         return outputs
487 
488     def mintime(self):
489         return min(self.state.send_query_delay, self.state.send_ack_delay, self.state.send_car_delay)
490 
491     def timeAdvance(self):
492         delay = min(self.state.send_query_delay, self.state.send_ack_delay, self.state.send_car_delay)
493         # Take care of floating point errors
494         return max(delay, 0)
495 
496     def preActivityCalculation(self):
497         return None
498 
499     def postActivityCalculation(self, _):
500         # If a car has collided, no activity is here
501         return 1 if len(self.state.cars_present) == 1 else 0
502 
503 class Road(CoupledDEVS):
504     def __init__(self, district, load, name="Road", segments=5):
505         CoupledDEVS.__init__(self, name)
506         self.segment = []
507         for i in range(segments):
508             self.segment.append(self.addSubModel(RoadSegment(load=load, district=district, name=(name + "_
                    " + str(i)))))
509 
510         # in-ports
511         self.q_rans = self.addInPort(name="q_rans")
512         self.q_recv = self.addInPort(name="q_recv")
513         self.car_in = self.addInPort(name="car_in")
514         self.entries = self.addInPort(name="entries")
515         self.q_rans_bs = self.addInPort(name="q_rans_bs")
516         self.q_recv_bs = self.addInPort(name="q_recv_bs")
517 
518         # out-ports
519         self.q_sans = self.addOutPort(name="q_sans")
520         self.q_send = self.addOutPort(name="q_send")
521         self.car_out = self.addOutPort(name="car_out")
522         self.exits = self.addOutPort(name="exits")
523         self.q_sans_bs = self.addOutPort(name="q_sans_bs")
524 
525         self.connectPorts(self.q_rans, self.segment[-1].q_rans)
526         self.connectPorts(self.q_recv, self.segment[0].q_recv)
527         self.connectPorts(self.car_in, self.segment[0].car_in)
528         self.connectPorts(self.entries, self.segment[0].entries)
529         self.connectPorts(self.q_rans_bs, self.segment[0].q_rans_bs)
530         self.connectPorts(self.q_recv_bs, self.segment[0].q_recv_bs)
531 
532         self.connectPorts(self.segment[0].q_sans, self.q_sans)
533         self.connectPorts(self.segment[-1].q_send, self.q_send)
534         self.connectPorts(self.segment[-1].car_out, self.car_out)
535         self.connectPorts(self.segment[0].exits, self.exits)
536         self.connectPorts(self.segment[0].q_sans_bs, self.q_sans_bs)
537 
538         for i in range(segments):
539             if i == 0:
540                 continue
541             self.connectPorts(self.segment[i].q_sans, self.segment[i-1].q_rans)
542             self.connectPorts(self.segment[i-1].q_send, self.segment[i].q_recv)
543             self.connectPorts(self.segment[i-1].car_out, self.segment[i].car_in)
544 
545 class IntersectionState():
546     def __init__(self, switch_signal):
547         self.send_query = []
548         self.send_ack = []
549         self.send_car = []
```

```
550            self.queued_queries = []
551            self.id_locations = [None, None, None, None]
552            self.block = vertical
553            self.switch_signal = switch_signal
554            self.ackDir = {}
555
556        def __str__(self):
557            return "ISECT␣blocking␣" + str(self.block)
558            return "Intersection:␣send_query␣=␣" + str(self.send_query) + ",␣send_ack␣=␣" + str(self.send_ack)
                   + ",␣send_car␣=␣" + str(self.send_car) + ",␣block␣=␣" + str(self.block)
559
560        def copy(self):
561            new = IntersectionState(self.switch_signal)
562            new.send_query = [q.copy() for q in self.send_query]
563            new.send_ack = [a.copy() for a in self.send_ack]
564            new.send_car = [c.copy() for c in self.send_car]
565            new.queued_queries = [q.copy() for q in self.queued_queries]
566            new.id_locations = list(self.id_locations)
567            new.block = self.block
568            new.switch_signal = self.switch_signal
569            new.ackDir = dict(self.ackDir)
570            return new
571
572    class Intersection(AtomicDEVS):
573        def __init__(self, district, name="Intersection", switch_signal = 30):
574            AtomicDEVS.__init__(self, name=name)
575            self.state = IntersectionState(switch_signal)
576            self.switch_signal_delay = switch_signal
577            self.district = district
578
579            self.q_send = []
580            self.q_send.append(self.addOutPort(name="q_send_to_n"))
581            self.q_send.append(self.addOutPort(name="q_send_to_e"))
582            self.q_send.append(self.addOutPort(name="q_send_to_s"))
583            self.q_send.append(self.addOutPort(name="q_send_to_w"))
584
585            self.q_rans = []
586            self.q_rans.append(self.addInPort(name="q_rans_from_n"))
587            self.q_rans.append(self.addInPort(name="q_rans_from_e"))
588            self.q_rans.append(self.addInPort(name="q_rans_from_s"))
589            self.q_rans.append(self.addInPort(name="q_rans_from_w"))
590
591            self.q_recv = []
592            self.q_recv.append(self.addInPort(name="q_recv_from_n"))
593            self.q_recv.append(self.addInPort(name="q_recv_from_e"))
594            self.q_recv.append(self.addInPort(name="q_recv_from_s"))
595            self.q_recv.append(self.addInPort(name="q_recv_from_w"))
596
597            self.q_sans = []
598            self.q_sans.append(self.addOutPort(name="q_sans_to_n"))
599            self.q_sans.append(self.addOutPort(name="q_sans_to_e"))
600            self.q_sans.append(self.addOutPort(name="q_sans_to_s"))
601            self.q_sans.append(self.addOutPort(name="q_sans_to_w"))
602
603            self.car_in = []
604            self.car_in.append(self.addInPort(name="car_in_from_n"))
605            self.car_in.append(self.addInPort(name="car_in_from_e"))
606            self.car_in.append(self.addInPort(name="car_in_from_s"))
607            self.car_in.append(self.addInPort(name="car_in_from_w"))
608
609            self.car_out = []
610            self.car_out.append(self.addOutPort(name="car_out_to_n"))
611            self.car_out.append(self.addOutPort(name="car_out_to_e"))
612            self.car_out.append(self.addOutPort(name="car_out_to_s"))
613            self.car_out.append(self.addOutPort(name="car_out_to_w"))
614
615        def intTransition(self):
616            self.state.switch_signal -= self.timeAdvance()
617            if self.state.switch_signal <= 1e-6:
618                # We switched our traffic lights
619                self.state.switch_signal = self.switch_signal_delay
620                self.state.queued_queries = []
```

```
621             self.state.block = vertical if self.state.block == horizontal else horizontal
622 
623             for loc, car_id in enumerate(self.state.id_locations):
624                 # Notify all cars that got 'green' that they should not continue
625                 if car_id is None:
626                     continue
627                 try:
628                     if str(loc) in self.state.block:
629                         query = Query(car_id)
630                         query.direction = int_to_dir[self.state.ackDir[car_id]]
631                         self.state.queued_queries.append(query)
632                 except KeyError:
633                     pass
634         self.state.send_car = []
635         self.state.send_query = []
636         self.state.send_ack = []
637         return self.state
638 
639     def extTransition(self, inputs):
640         # Simple forwarding of all messages
641         # Unless the direction in which it is going is blocked
642         self.state.switch_signal -= self.elapsed
643         for direction in range(4):
644             blocked = str(direction) in self.state.block
645             for car in inputs.get(self.car_in[direction], []):
646                 self.state.send_car.append(car)
647                 # Got a car, so remove its ID location entry
648                 try:
649                     del self.state.ackDir[car.ID]
650                     self.state.id_locations[self.state.id_locations.index(car.ID)] = None
651                 except (KeyError, ValueError):
652                     pass
653                 self.state.queued_queries = [query for query in self.state.queued_queries if query.ID !=
                        car.ID]
654             for query in inputs.get(self.q_recv[direction], []):
655                 self.state.id_locations[direction] = query.ID
656                 self.state.ackDir[query.ID] = dir_to_int[query.direction]
657                 if blocked:
658                     self.state.send_ack.append(QueryAck(query.ID, INFINITY))
659                     self.state.queued_queries.append(query)
660                 else:
661                     self.state.send_query.append(query)
662             for ack in inputs.get(self.q_rans[direction], []):
663                 self.state.ackDir[ack.ID] = direction
664                 if (ack.t_until_dep > self.state.switch_signal) or ((ack.ID in self.state.id_locations)
                        and (str(self.state.id_locations.index(ack.ID)) in self.state.block)):
665                     try:
666                         self.state.id_locations.index(ack.ID)
667                         t_until_dep = INFINITY
668                         nquery = Query(ack.ID)
669                         nquery.direction = int_to_dir[direction]
670                         self.state.queued_queries.append(nquery)
671                     except ValueError:
672                         continue
673                 else:
674                     t_until_dep = ack.t_until_dep
675                 self.state.send_ack.append(QueryAck(ack.ID, t_until_dep))
676         return self.state
677 
678     def outputFnc(self):
679         # Can simply overwrite, as multiple calls to the same destination is impossible
680         toSend = {}
681         new_block = vertical if self.state.block == horizontal else horizontal
682         if self.state.switch_signal == self.timeAdvance():
683             # We switch our traffic lights
684             # Resend all queries for those that are waiting
685             for query in self.state.queued_queries:
686                 if str(self.state.id_locations.index(query.ID)) not in new_block:
687                     toSend[self.q_send[dir_to_int[query.direction]]] = [query]
688             for loc, car_id in enumerate(self.state.id_locations):
689                 # Notify all cars that got 'green' that they should not continue
690                 if car_id is None:
```

```
691                     continue
692                 if str(loc) in new_block:
693                     toSend[self.q_sans[loc]] = [QueryAck(car_id, INFINITY)]
694                 else:
695                     try:
696                         query = Query(car_id)
697                         query.direction = int_to_dir[self.state.ackDir[car_id]]
698                         toSend[self.q_send[self.state.ackDir[car_id]]] = [query]
699                     except KeyError:
700                         pass
701         # We might have some messages to forward too
702         for car in self.state.send_car:
703             dest = car.path.pop(0)
704             toSend[self.car_out[dir_to_int[dest]]] = [car]
705         for query in self.state.send_query:
706             toSend[self.q_send[dir_to_int[query.direction]]] = [query]
707         for ack in self.state.send_ack:
708             # Broadcast for now
709             try:
710                 toSend[self.q_sans[self.state.id_locations.index(ack.ID)]] = [ack]
711             except ValueError:
712                 pass
713         return toSend
714 
715     def timeAdvance(self):
716         if len(self.state.send_car) + len(self.state.send_query) + len(self.state.send_ack) > 0:
717             return 0.0
718         else:
719             return max(self.state.switch_signal, 0.0)
720 
721     def preActivityCalculation(self):
722         return None
723 
724     def postActivityCalculation(self, _):
725         return len(self.state.send_car)
726 
727 class CollectorState(object):
728     def __init__(self):
729         self.cars = []
730 
731     def copy(self):
732         new = CollectorState()
733         new.cars = list(self.cars)
734         return new
735 
736     def __str__(self):
737         ## Print your statistics here!
738         s = "All␣cars␣collected:"
739         for car in self.cars:
740             s += "\n\t\t\t%s" % car
741         return s
742 
743 class Collector(AtomicDEVS):
744     def __init__(self):
745         AtomicDEVS.__init__(self, "Collector")
746         self.car_in = self.addInPort("car_in")
747         self.state = CollectorState()
748         self.district = 0
749 
750     def extTransition(self, inputs):
751         self.state.cars.extend(inputs[self.car_in])
752         return self.state
753 
754     def preActivityCalculation(self):
755         return None
756 
757     def postActivityCalculation(self, _):
758         return 0
```

Listing A.1: The citylayout benchmark model including some domain-specific optimizations

A.2 Experiment

In order to run the model, an experiment file is necessary, as shown in Listing A.2. Though this is a very basic experiment file, all additional options are similar in usage and a list of them can be found in the documentation. Very simplistic models, which only show the core of the feature, are shown in the documentation that is included in the PythonPDEVS package.

```
1  import sys
2  # Include PyPDEVS source
3  sys.path.append("../../src/")
4  from simulator import Simulator
5  from generated_city import City
6
7  sim = Simulator(City())
8  sim.setVerbose(None)
9  sim.setTerminationTime(5000.0)
10 sim.simulate()
```

Listing A.2: A simple experiment file

A.3 Scheduler

As an example of how to write a custom scheduler, the code for the minimal list scheduler is provided in Listing A.3.

```
class Scheduler(object):
    """
    Scheduler class itself
    """
    def __init__(self, models, epsilon, totalModels):
        """
        Constructor

        :param models: all models in the simulation
        :param epsilon: the desired epsilon for time equality
        :param totalModels: a list of all models in the complete simulation
        """
        # Make a copy!
        self.models = list(models)
        self.minval = (float(’inf’), float(’inf’))
        self.epsilon = epsilon
        self.massReschedule([])
        # We have no need for the totalModels in this case

    def schedule(self, model):
        """
        Schedule a model

        :param model: the model to schedule
        """
        # Add the model to the list
        self.models.append(model)
        if model.timeNext < self.minval:
            # Update our cached value if this event occurs earlier
            self.minval = model.timeNext

    def unschedule(self, model):
        """
        Unschedule a model

        :param model: model to unschedule
        """
        # Remove the model from our list of models
        self.models.remove(model)

        if model.timeNext == self.minval:
            # It was the minimal value
            # Iterate over the list and find the new minimal value
            self.minval = (float(’inf’), float(’inf’))
            for m in self.models:
                if m.timeNext < self.minval:
                    self.minval = m.timeNext

    def massReschedule(self, reschedule_set):
        """
        Reschedule all models provided.
        Equivalent to calling unschedule(model); schedule(model) on every element in the iterable.

        :param reschedule_set: iterable containing all models to reschedule
        """
        # Independent of the reschedule_set, as we simply reiterate the list
        self.minval = (float(’inf’), float(’inf’))
        for m in self.models:
            if m.timeNext < self.minval:
                self.minval = m.timeNext

    def readFirst(self):
        """
        Returns the time of the first model that has to transition

        :returns: timestamp of the first model
        """
        # Use our cached value
```

```
69            return self.minval
70
71        def getImminent(self, time):
72            """
73            Returns a list of all models that transition at the provided time, with the specified epsilon
                  deviation allowed.
74
75            :param time: timestamp to check for models
76            """
77            immChildren = []
78            t, age = time
79            # Simply iterate and use our epsilon
80            for model in self.models:
81                if abs(model.timeNext[0] - t) < self.epsilon and model.timeNext[1] == age:
82                    immChildren.append(model)
83            return immChildren
```

Listing A.3: A custom scheduler

A.4 Static Allocator

A simple static allocator is presented in Listing A.4. This allocator will allocate direct subtrees of the root model to a single node, in an attempt to equalize the number of models. As it is a static allocator, the `getTerminationTime` method should return 0 and the algorithm should not use the `edges` and `totalActivities` parameters.

```
1  class AutoAllocator(object):
2      """
3      Allocate all models in a static manner, simply trying to divide the number of models equally.
4      Our 'heuristic' is to allocate in chunks as defined in the root coupled model.
5      """
6      def allocate(self, models, edges, nrnodes, totalActivities):
7          """
8          Calculate allocations for the nodes, using the information provided.
9
10         :param models: the models to allocte
11         :param edges: the edges between the models
12         :param nrnodes: the number of nodes to allocate over. Simply an upper bound!
13         :param totalActivities: activity tracking information from each model
14         :returns: allocation that was found
15         """
16         allocation = {}
17         allocatedTopmost = {}
18         currentNode = 0
19         totalModels = len(models)
20
21         # Iterate over every model
22         for model in models:
23             # Not yet allocated, so allocate it somewhere
24             child = model
25             searchmodel = model
26             # Find its main subtree
27             while searchmodel.parent is not None:
28                 child = searchmodel
29                 searchmodel = searchmodel.parent
30             # searchmodel is now the root model
31             # child is its 1st decendant, on which we will allocate
32             try:
33                 # Allocate the child to the same node as the main subtree
34                 node = allocatedTopmost[child]
35             except KeyError:
36                 # Subtree is unallocated, so allocate it somewhere
37                 currentNode = (currentNode + 1) % nrnodes
38                 allocatedTopmost[child] = currentNode
39                 node = currentNode
40             allocation[model.model_id] = node
41
42         return allocation
43
44     def getTerminationTime(self):
45         """
46         Returns the time it takes for the allocator to make an 'educated guess' of the advised allocation.
47         This time will not be used exactly, but as soon as the GVT passes over it. While this is not
               exactly
48         necessary, it avoids the overhead of putting such a test in frequently used code.
49
50         :returns: float -- the time at which to perform the allocations (and save them)
51         """
52         # No need for any run time information
53         return 0.0
```

Listing A.4: A custom static allocator

A.5 Dynamic Allocator

Listing A.5 presents a simple dynamic allocator as is used in the distributed synthetic activity benchmark. It will distribute the activity by exploiting the ring structure of the model. The average activity per node is computed, after which the ring is iterated and models are added until the node is near the average. As it is a dynamic allocator, its `getTerminationTime` should return a value different from 0. Additionally, it can use the `edges` and `totalActivities` parameters.

```
class MyAllocator(object):
    """
    Allocate all models at the start of the simulation. After this, model relocation is handed over to a
        relocator.
    """
    def allocate(self, models, edges, nrnodes, totalActivities):
        """
        Calculate allocations for the nodes, using the information provided.

        :param models: the models to allocte
        :param edges: the edges between the models
        :param nrnodes: the number of nodes to allocate over. Simply an upper bound!
        :param totalActivities: activity tracking information from each model
        :returns: allocation that was found
        """
        # Return something of the form: {0: 0, 1: 0, 2: 0, 3: 1}
        # To allocate model_ids 0, 1 and 2 to node 0 and model_id 3 to node 1
        avgload = sum(totalActivities.values()) / nrnodes
        alloc = {}
        runningload = 0.0
        currentnode = 0
        for node, activity in totalActivities.items():
            if runningload + (activity / 2) > avgload:
                currentnode = (currentnode + 1) % nrnodes
                runningload = 0.0
            runningload += activity
            alloc[node] = currentnode
        return alloc

    def getTerminationTime(self):
        """
        Returns the time it takes for the allocator to make an 'educated guess' of the advised allocation.
        This time will not be used exactly, but as soon as the GVT passes over it. While this is not
            exactly
        necessary, it avoids the overhead of putting such a test in frequently used code.

        :returns: float -- the time at which to perform the allocations (and save them)
        """
        # No need for any run time information means 0.0
        return 2.0
```

Listing A.5: A custom dynamic allocator

A.6 Activity Relocator

One of the relocators used in the citylayout benchmark is shown in Listing A.6. It will simply accumulate the default activity values (the time taken in internal transition functions) of the districts and try to balance the load.

```
from collections import defaultdict

class CityRelocator(object):
    def __init__(self):
        self.server = None
        self.kernels = 0
        self.model_ids = []

    def setController(self, controller):
        self.kernels = controller.kernels
        self.server = controller.server
        self.model_ids = controller.model_ids
        self.districts = defaultdict(list)
        for m in controller.total_model.componentSet:
            self.districts[m.district].append(m)
        self.districts = [self.districts[i] for i in range(len(self.districts))]

    def getRelocations(self, GVT, activities, horizon):
        # Ignore activities variable
        # Fetch all models and their activity
        if GVT < 100.0:
            return {}
        previous_district_allocation = [district[0].location for district in self.districts]
        relocate = {}
        district_activities = defaultdict(float)
        for i in range(self.kernels):
            for model_id, activity in self.server.getProxy(i).getCompleteActivity().items():
                district_activities[self.model_ids[model_id].district] += activity
        district_activities = [district_activities[i] for i in range(len(district_activities))]
        print("All␣loads:␣" + str(district_activities))
        avg_activity_per_node = float(sum(district_activities)) / self.kernels
        print("Avg:␣" + str(avg_activity_per_node))
        running_activity = 0.0
        district_allocation = []
        to_allocate = []
        for district, activity in enumerate(district_activities):
            if abs(running_activity - avg_activity_per_node) < abs(running_activity -
                avg_activity_per_node + activity):
                # Enough activity for this node, so put all these districts there
                district_allocation.append(to_allocate)
                running_activity = activity
                to_allocate = [district]
            else:
                running_activity += activity
                to_allocate.append(district)
        if len(district_allocation) < self.kernels:
            # Still have to add the last node
            district_allocation.append(to_allocate)
        else:
            district_allocation[-1].extend(to_allocate)
        print("Migrating␣to␣%s" % district_allocation)
        for node, districts in enumerate(district_allocation):
            for district in districts:
                if previous_district_allocation[district] == node or (previous_district_allocation[
                    district] < node and GVT > horizon):
                    continue
                print("Moving␣" + str(district) + "␣to␣" + str(node))
                for model in self.districts[district]:
                    relocate[model.model_id] = node
        return relocate

    def useLastStateOnly(self):
        return False
```

Listing A.6: A custom relocator

A.7 Last state Relocator

One of the relocators used in the citylayout benchmark is shown in Listing A.7. It will fetch all activity values of every model seperately, after which the activity per district is accumulated. A prediction is made about how the activity evolves, before relocating the districts over different nodes.

```
1  from collections import defaultdict
2
3  class CityRelocator(object):
4      """
5      A relocation component for the citylayout benchmark
6      """
7      def __init__(self):
8          """
9          Constructor
10         """
11         self.server = None
12         self.kernels = 0
13         self.model_ids = []
14
15     def setController(self, controller):
16         """
17         Configures the controller, to call different simulation nodes and access the model.
18
19         This method is called as soon as the controller is constructed.
20
21         :param controller: the controller used in the simulation
22         """
23         self.kernels = controller.kernels
24         self.server = controller.server
25         self.model_ids = controller.model_ids
26         self.districts = defaultdict(list)
27         for m in controller.total_model.componentSet:
28             self.districts[m.district].append(m)
29         self.districts = [self.districts[i] for i in range(len(self.districts))]
30
31     def getRelocations(self, GVT, activities, horizon):
32         """
33         Calculate the relocations to be done.
34
35         :param GVT: the current GVT
36         :param activities: the accumulated activity values for every node seperately (or None if last
               state is used)
37         :param horizon: the horizon over which activities were measured
38         """
39         if GVT < 100.0:
40             # Still in warmup
41             return {}
42         # Fetch the current allocation
43         previous_district_allocation = [district[0].location for district in self.districts]
44         relocate = {}
45         district_activities = defaultdict(float)
46         # Fetch the activities of every model seperately and fill in the districts activity
47         for i in range(self.kernels):
48             for model_id, activity in self.server.getProxy(i).getCompleteActivity().items():
49                 district_activities[self.model_ids[model_id].district] += activity
50         # Convert it to a list for easy access
51         district_activities = [district_activities[i] for i in range(len(district_activities))]
52
53         # Shift the loads a little to 'predict the future'
54         new_district_activities = list(district_activities)
55         for index in range(len(district_activities)):
56             # 0.0015 is a value found from several profiling runs to depict the time it takes for a car to
                   leave the district
57             leaving = district_activities[index] * (0.0015 * horizon)
58             new_district_activities[index] -= leaving
59             # Note that this is an oversimplification, though it is 'fairly' accurate
60             if index != len(district_activities) - 1:
61                 if index < 5:
62                     #  Assume the 5 first districts are generating and do not receive any new cars
63                     arrived = 0
64                 else:
```

```
65                    # These are commercial districts
66                    commercials_left = float(10-index)
67                    # Assume uniform distribution of cars
68                    arrived = leaving * ((commercials_left - 1) / (commercials_left))
69                # Update it with the found values
70                new_district_activities[index+1] += (leaving - arrived)
71        district_activities = new_district_activities
72        # Our prediction is now in district_activities
73        avg_activity_per_node = float(sum(district_activities)) / self.kernels
74        if avg_activity_per_node < 20:
75            # Not enough load per node to actually profit from the relocation
76            return {}
77
78        # Now distribute the load according to the found prediction
79        running_activity = 0.0
80        district_allocation = []
81        to_allocate = []
82        for district, activity in enumerate(district_activities):
83            if abs(running_activity - avg_activity_per_node) < abs(running_activity -
                  avg_activity_per_node + activity):
84                # Enough activity for this node, so put all these districts there
85                district_allocation.append(to_allocate)
86                running_activity = activity
87                to_allocate = [district]
88            else:
89                running_activity += activity
90                to_allocate.append(district)
91        if len(district_allocation) < self.kernels:
92            # Still have to add the last node
93            district_allocation.append(to_allocate)
94        else:
95            district_allocation[-1].extend(to_allocate)
96
97        # Now that all districts are allocated, we have to remap to models
98        for node, districts in enumerate(district_allocation):
99            for district in districts:
100               if previous_district_allocation[district] == node or (previous_district_allocation[
                      district] < node and GVT > horizon):
101                   # Do not relocate models 'back'
102                   continue
103               for model in self.districts[district]:
104                   relocate[model.model_id] = node
105       return relocate
106
107   def useLastStateOnly(self):
108       """
109       Whether or not to use last state
110       """
111       return True
```

Listing A.7: A custom "last state" relocator

A.8 Tracer

To give an indication of how a full featured tracer is used, we present the verbose tracer in Listing A.8. Note that this is a full-fledged tracer, so it has support for distributed simulation, simulation continuation, checkpointing, ...

```
from util import runTraceAtController
import sys

class TracerVerbose(object):
    """
    A tracer for simple verbose output
    """
    def __init__(self, uid, server, filename):
        """
        Constructor

        :param uid: the UID of this tracer
        :param server: the server to make remote calls on
        :param filename: file to save the trace to, can be None for output to stdout
        """
        if server.getName() == 0:
            # Only create the file at the controller
            self.filename = filename
        else:
            # Not at the controller, so ignore
            self.filename = None
        self.server = server
        self.prevtime = (-1, -1)
        # Our assigned id
        self.uid = uid

    def startTracer(self, recover):
        """
        Starts up the tracer

        :param recover: whether or not this is a recovery call (so whether or not the file should be
            appended to)
        """
        if self.filename is None:
            self.verb_file = sys.stdout
        elif recover:
            self.verb_file = open(self.filename, 'a+')
        else:
            self.verb_file = open(self.filename, 'w')

    def stopTracer(self):
        """
        Stop the tracer
        """
        self.verb_file.flush()

    def trace(self, time, text):
        """
        Actual tracing function

        :param time: time at which this trace happened
        :param text: the text that was traced
        """
        string = ""
        if time > self.prevtime:
            # A time delimiter is required
            string = ("\n__␣␣Current␣Time:␣%10.2f␣" + "_"*42 + "␣\n\n") % (time[0])
            self.prevtime = time
        string += "%s\n" % text
        # Write out the text as required
        try:
            self.verb_file.write(string)
        except TypeError:
            # For Python3
            self.verb_file.write(string.encode())

    def traceInternal(self, aDEVS):
```

```
67          """
68          Tracing done for the internal transition function
69
70          :param aDEVS: the model that transitioned
71          """
72          text = "\n\tINTERNAL␣TRANSITION␣in␣model␣<%s>\n" % aDEVS.getModelFullName()
73          text += "\t\tNew␣State:␣%s\n" % str(aDEVS.state)
74          text += "\t\tOutput␣Port␣Configuration:\n"
75          for I in range(len(aDEVS.OPorts)):
76              text += "\t\t\tport␣<" + str(aDEVS.OPorts[I].getPortName()) + ">:\n"
77              for msg in aDEVS.myOutput.get(aDEVS.OPorts[I], []):
78                  text += "\t\t\t\t" + str(msg) + "\n"
79          text += "\t\tNext␣scheduled␣internal␣transition␣at␣time␣%.2f\n" % (aDEVS.timeNext[0])
80          runTraceAtController(self.server, self.uid, aDEVS, [aDEVS.timeLast, '"' + text + '"'])
81
82      def traceConfluent(self, aDEVS):
83          """
84          Tracing done for the confluent transition function
85
86          :param aDEVS: the model that transitioned
87          """
88          text = "\n\tCONFLUENT␣TRANSITION␣in␣model␣<%s>\n" % aDEVS.getModelFullName()
89          text += "\t\tInput␣Port␣Configuration:\n"
90          for I in range(len(aDEVS.IPorts)):
91              text += "\t\t\tport␣<" + str(aDEVS.IPorts[I].getPortName()) + ">:␣\n"
92              for msg in aDEVS.myInput.get(aDEVS.IPorts[I], []):
93                  text += "\t\t\t\t" + str(msg) + "\n"
94          text += "\t\tNew␣State:␣%s\n" % str(aDEVS.state)
95          text += "\t\tOutput␣Port␣Configuration:\n"
96          for I in range(len(aDEVS.OPorts)):
97              text += "\t\t\tport␣<" + str(aDEVS.OPorts[I].getPortName()) + ">:\n"
98              for msg in aDEVS.myOutput.get(aDEVS.OPorts[I], []):
99                  text += "\t\t\t\t" + str(msg) + "\n"
100         text += "\t\tNext␣scheduled␣internal␣transition␣at␣time␣%.2f\n" % (aDEVS.timeNext[0])
101         runTraceAtController(self.server, self.uid, aDEVS, [aDEVS.timeLast, '"' + text + '"'])
102
103     def traceExternal(self, aDEVS):
104         """
105         Tracing done for the external transition function
106
107         :param aDEVS: the model that transitioned
108         """
109         text = "\n\tEXTERNAL␣TRANSITION␣in␣model␣<%s>\n" % aDEVS.getModelFullName()
110         text += "\t\tInput␣Port␣Configuration:\n"
111         for I in range(len(aDEVS.IPorts)):
112             text += "\t\t\tport␣<" + str(aDEVS.IPorts[I].getPortName()) + ">:\n"
113             for msg in aDEVS.myInput.get(aDEVS.IPorts[I], []):
114                 text += "\t\t\t\t" + str(msg) + "\n"
115         text += "\t\tNew␣State:␣%s\n" % str(aDEVS.state)
116         text += "\t\tNext␣scheduled␣internal␣transition␣at␣time␣%.2f\n" % (aDEVS.timeNext[0])
117         runTraceAtController(self.server, self.uid, aDEVS, [aDEVS.timeLast, '"' + text + '"'])
118
119     def traceInit(self, aDEVS, t):
120         """
121         Tracing done for the initialisation
122
123         :param aDEVS: the model that was initialised
124         :param t: time at which it should be traced
125         """
126         text = "\n\tINITIAL␣CONDITIONS␣in␣model␣<%s>\n" % aDEVS.getModelFullName()
127         text += "\t\tInitial␣State:␣%s\n" % str(aDEVS.state)
128         text += "\t\tNext␣scheduled␣internal␣transition␣at␣time␣%.2f\n" % (aDEVS.timeNext[0])
129         runTraceAtController(self.server, self.uid, aDEVS, [t, '"' + text + '"'])
130
131     def traceUser(self, time, aDEVS, variable, value):
132         text = "\n\tUSER␣CHANGE␣in␣model␣<%s>\n" % aDEVS.getModelFullName()
133         text += "\t\tAltered␣attribute␣<%s>␣to␣value␣<%s>\n" % (variable, value)
134         # Is only called at the controller, outside of the GVT loop, so commit directly
135         self.trace(time, text)
```

Listing A.8: A custom tracer

B
Citylayout generator

For completeness, the utility that was used to generate the citylayout model is included here. Note that it generates a rectangular city which consists of a sequence of multiple districts. Whereas this is not real limitation of the generator, it provides a more simple mapping between activity and districts. Additionally, it reduces the amount of inter-district communication as a district only has to communicate with 2 other districts.

```
import random
import math
import sys

############################################
#  Our sample city looks like this:
# x12345
# 1 | |
# 2-+-+- -->
# 3 | |
# 4-+-+- <--
# 5 | |
#
#    ^ |
#    | V
#
#  At each intersection, there are traffic lights
#  Some cars ride over it and all of them should
#  reach their destination
############################################

north = 0
east = 1
south = 2
west = 3

district_map = {}

class City(object):
    def addRoad(self, name):
        self.exists.add(name)
        self.outfile.write("␣␣␣␣␣␣␣␣self.road_%s␣=␣self.addSubModel(Road(load=int(sys.argv[1]),␣name='road_%s',␣district=%s),␣district_map[%i])\n" % (name, name, district_map[name], district_map[name]))

    def addResidential(self, name, path):
        #print("Found path " + str(path))
        self.exists.add(name)
        self.outfile.write("␣␣␣␣␣␣␣␣self.residential_%s␣=␣self.addSubModel(Residence(%s,␣name='residential_%s',␣district=%s),␣district_map[%i])\n" % (name, path, name, district_map[name], district_map[name]))
```

```
38          self.outfile.write("␣␣␣␣␣␣␣␣self.connectPorts(self.residential_%s.q_send,␣self.road_%s.q_recv_bs)\
                n" % (name, name))
39          self.outfile.write("␣␣␣␣␣␣␣␣self.connectPorts(self.residential_%s.exit,␣self.road_%s.entries)\n" %
                 (name, name))
40          self.outfile.write("␣␣␣␣␣␣␣␣self.connectPorts(self.road_%s.q_sans_bs,␣self.residential_%s.q_rans)\
                n" % (name, name))
41
42      def addCommercial(self, name):
43          self.exists.add(name)
44          self.outfile.write("␣␣␣␣␣␣␣␣self.commercial_%s␣=␣self.addSubModel(Commercial(name='commercial_%s',
                ␣district=%s),␣district_map[%i])\n" % (name, name, district_map[name], district_map[name]))
45          self.outfile.write("␣␣␣␣␣␣␣␣self.connectPorts(self.road_%s.exits,␣self.commercial_%s.entry)\n" % (
                name, name))
46          self.outfile.write("␣␣␣␣␣␣␣␣self.connectPorts(self.commercial_%s.toCollector,␣self.collector.
                car_in)\n" % (name))
47
48      def addIntersection(self, name):
49          self.exists.add(name)
50          self.outfile.write("␣␣␣␣␣␣␣␣self.intersection_%s␣=␣self.addSubModel(Intersection(name='
                intersection_%s',␣district=%s),␣district_map[%i])\n" % (name, name, district_map[name],
                district_map[name]))
51          # Fetch the direction
52          splitName = name.split("_")
53          horizontalName = splitName[1]
54          verticalName = splitName[0]
55          horizontal = self.getDirection(splitName[0], True)
56          vertical = self.getDirection(splitName[1], False)
57
58          if horizontal == east:
59              nameHT = str(int(horizontalName) + 1)
60              nameHF = str(int(horizontalName) - 1)
61          elif horizontal == west:
62              nameHT = str(int(horizontalName) - 1)
63              nameHF = str(int(horizontalName) + 1)
64          if vertical == north:
65              nameVT = str(int(verticalName) - 1)
66              nameVF = str(int(verticalName) + 1)
67          elif vertical == south:
68              nameVT = str(int(verticalName) + 1)
69              nameVF = str(int(verticalName) - 1)
70
71          intersection = name
72          roadHorizontalTo = splitName[0] + "_" + nameHT
73          roadHorizontalFrom = splitName[0] + "_" + nameHF
74          roadVerticalTo = nameVT + "_" + splitName[1]
75          roadVerticalFrom = nameVF + "_" + splitName[1]
76
77          self.connectIntersectionToRoad(intersection, roadHorizontalTo, horizontal)
78          self.connectIntersectionToRoad(intersection, roadVerticalTo, vertical)
79          self.connectRoadToIntersection(roadHorizontalFrom, intersection, horizontal)
80          self.connectRoadToIntersection(roadVerticalFrom, intersection, vertical)
81
82      def connectIntersectionToRoad(self, intersectionname, roadname, direction):
83          self.outfile.write("␣␣␣␣␣␣␣␣self.connectPorts(self.intersection_%s.q_send[%s],␣self.road_%s.q_recv
                )\n" % (intersectionname, direction, roadname))
84          self.outfile.write("␣␣␣␣␣␣␣␣self.connectPorts(self.road_%s.q_sans,␣self.intersection_%s.q_rans[%s
                ])\n" % (roadname, intersectionname, direction))
85          self.outfile.write("␣␣␣␣␣␣␣␣self.connectPorts(self.intersection_%s.car_out[%s],␣self.road_%s.
                car_in)\n" % (intersectionname, direction, roadname))
86
87      def connectRoadToIntersection(self, roadname, intersectionname, direction):
88          # Invert the direction first
89          direction = {north: south, east: west, south: north, west: east}[direction]
90          self.outfile.write("␣␣␣␣␣␣␣␣self.connectPorts(self.road_%s.q_send,␣self.intersection_%s.q_recv[%s
                ])\n" % (roadname, intersectionname, direction))
91          self.outfile.write("␣␣␣␣␣␣␣␣self.connectPorts(self.intersection_%s.q_sans[%s],␣self.road_%s.q_rans
                )\n" % (intersectionname, direction, roadname))
92          self.outfile.write("␣␣␣␣␣␣␣␣self.connectPorts(self.road_%s.car_out,␣self.intersection_%s.car_in[%s
                ])\n" % (roadname, intersectionname, direction))
93
94      def getDirection(self, name, horizontal):
95          name = int(name)
```

```
96          while 1:
97              try:
98                  elem = self.directions[name]
99                  break
100             except KeyError:
101                 name -= 4
102                 if name < 0:
103                     return None
104                 if name % 2 != 0:
105                     return None
106
107         if horizontal:
108             return elem[1] if elem[1] in [east, west] else elem[0]
109         else:
110             return elem[1] if elem[1] in [north, south] else elem[0]
111
112     def setDirection(self, name, direction1, direction2):
113         self.directions[int(name)] = (direction1, direction2)
114
115     def planRoute(self, source, destination):
116         sh, sv = source.split("_")
117         dh, dv = destination.split("_")
118         sh, sv, dh, dv = int(sh), int(sv), int(dh), int(dv)
119         #print("Planning route from %s to %s" % (source, destination))
120
121         if sv % 2 == 0:
122             # We are in a vertical street
123             direction = self.getDirection(sv, False)
124             if direction == north:
125                 # The actual intersection to start from is the one north from it
126                 sourceIntersection = "%i_%i" % (sh-1, sv)
127             elif direction == south:
128                 sourceIntersection = "%i_%i" % (sh+1, sv)
129         else:
130             # We are in a horizontal street
131             direction = self.getDirection(sh, True)
132             if direction == east:
133                 # The actual intersection to start from is the one north from it
134                 sourceIntersection = "%i_%i" % (sh, sv+1)
135             elif direction == west:
136                 sourceIntersection = "%i_%i" % (sh, sv-1)
137
138         if dv % 2 == 0:
139             # We are in a vertical street
140             direction = self.getDirection(dv, False)
141             if direction == north:
142                 # The actual intersection is SOUTH from us
143                 destinationIntersection = "%i_%i" % (dh+1, dv)
144                 laststep = 'n'
145             elif direction == south:
146                 destinationIntersection = "%i_%i" % (dh-1, dv)
147                 laststep = 's'
148         else:
149             # We are in a horizontal street
150             direction = self.getDirection(dh, True)
151             if direction == east:
152                 # The actual intersection is WEST from us
153                 destinationIntersection = "%i_%i" % (dh, dv-1)
154                 laststep = 'e'
155             elif direction == west:
156                 destinationIntersection = "%i_%i" % (dh, dv+1)
157                 laststep = 'w'
158
159         # Now plan a route from intersection 'sourceIntersection' to 'destinationIntersection'
160         #  furthermore add the laststep, to go to the actual location
161         return self.bfSearch(sourceIntersection, destinationIntersection) + [laststep]
162
163     def bfSearch(self, source, destination):
164         #print("Search path from %s to %s" % (source, destination))
165         seen = set()
166         if source == destination:
167             return []
```

```
168         old_source, old_destination = source, destination
169         steps = {north: 'n', east: 'e', south: 's', west: 'w'}
170         newWorklist = [(source, [])]
171         while len(newWorklist) > 0:
172             worklist = newWorklist
173             newWorklist = []
174             for source, path in worklist:
175                 s_sh, s_sv = source.split("_")
176                 sh, sv = int(s_sh), int(s_sv)
177                 step1 = self.getDirection(sh, True)
178                 step2 = self.getDirection(sv, False)
179                 newsource1 = "%i_%i" % (sh, {east: sv+2, west: sv-2}[step1])
180                 newsource2 = "%i_%i" % ({north: sh-2, south: sh+2}[step2], sv)
181                 if newsource1 in self.exists and newsource1 not in seen:
182                     newWorklist.append((newsource1, list(path + [steps[step1]])))
183                     seen.add(newsource1)
184                     if newsource1 == destination:
185                         return newWorklist[-1][1]
186                 if newsource2 in self.exists and newsource2 not in seen:
187                     newWorklist.append((newsource2, list(path + [steps[step2]])))
188                     seen.add(newsource2)
189                     if newsource2 == destination:
190                         return newWorklist[-1][1]
191 
192         raise KeyError("Couldn't␣find␣a␣route␣from␣%s␣to␣%s" % (old_source, old_destination))
193 
194     def __init__(self, districts):
195         self.outfile = open('generated_city.py', 'w')
196         self.outfile.write("import␣sys\n")
197         self.outfile.write("import␣random\n")
198         self.outfile.write("random.seed(1)\n")
199         self.outfile.write("sys.path.append('../../src/')\n")
200         self.outfile.write("from␣trafficModels␣import␣*\n")
201         self.directions = {}
202 
203         self.paths = 0
204         self.setDirection("2", north, east)
205         self.setDirection("4", south, west)
206 
207         x, y = int(sys.argv[1]), int(sys.argv[2])
208 
209         # NOTE for legacy reasons, x and y are reversed...
210         x, y = (y if y % 2 else y + 1), (x if x % 2 else x + 1)
211         nodes = int(sys.argv[4])
212         district_to_node = [min(int(float(district*nodes)/districts), nodes-1) for district in range(
                districts)]
213         print("Got␣map:␣" + str(district_to_node))
214         residentialDistricts = districts/2
215 
216         self.outfile.write("district_map␣=␣%s\n" % district_to_node)
217         self.outfile.write("class␣City(CoupledDEVS):\n")
218         self.outfile.write("␣␣␣␣def␣__init__(self):\n")
219         self.outfile.write("␣␣␣␣␣␣␣␣CoupledDEVS.__init__(self,␣'City')\n")
220         self.outfile.write("␣␣␣␣␣␣␣␣self.collector␣=␣self.addSubModel(Collector(),␣0)\n")
221         self.outfile.write("␣␣␣␣␣␣␣␣\n")
222 
223         self.districts = [[] for _ in range(districts)]
224 
225         self.exists = set()
226         # Draw horizontal streets
227         for i_x in range(2, x+1, 2):
228             for i_y in range(1, y+1, 2):
229                 district = min(int(i_x * (districts / float(x))), districts-1)
230                 name = "%i_%i" % (i_x, i_y)
231                 district_map[name] = district
232                 self.districts[district].append(name)
233                 if name not in self.exists:
234                     self.addRoad(name)
235                     self.exists.add(name)
236 
237         # Draw the vertical streets
238         for i_y in range(2, y+1, 2):
```

```
239             for i_x in range(1, x+1, 2):
240                 district = min(int(i_x * (districts / float(x))), districts-1)
241                 name = "%i_%i" % (i_x, i_y)
242                 district_map[name] = district
243                 self.districts[district].append(name)
244                 if name not in self.exists:
245                     self.addRoad(name)
246                     self.exists.add(name)
247 
248         # And the intersections
249         for i_x in range(2, x+1, 2):
250             for i_y in range(2, y+1, 2):
251                 district = min(int(i_x * (districts / float(x))), districts-1)
252                 name = "%i_%i" % (i_x, i_y)
253                 district_map[name] = district
254                 self.addIntersection(name)
255                 self.exists.add(name)
256 
257         sourceDistricts = [name for district in self.districts[:residentialDistricts] for name in district
                ]
258         destinationDistricts = [name for district in self.districts[residentialDistricts:] for name in
                district]
259         random.shuffle(sourceDistricts)
260         random.shuffle(destinationDistricts)
261 
262         paths = min(len(sourceDistricts), len(destinationDistricts))
263         for source, destination in zip(sourceDistricts, destinationDistricts):
264             self.addPath(source, destination)
265             paths -= 1
266             if paths % 100 == 0:
267                 print("To␣find:␣" + str(paths))
268 
269         print("Total␣paths:␣" + str(self.paths))
270 
271     def addPath(self, source, destination):
272         try:
273             self.addResidential(source, self.planRoute(source, destination))
274             self.addCommercial(destination)
275             self.paths += 1
276         except KeyError:
277             pass
278 
279 if __name__ == "__main__":
280     random.seed(5)
281     City(int(sys.argv[3]))
```

Listing B.1: The citylayout benchmark model generator

Bibliography

[1] IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, 2006.

[2] Adedoyin Adegoke, Hamidou Togo, and Mamadou K. Traoré. A unifying framework for specifying DEVS parallel and distributed simulation architectures. *Simulation*, 89:1293–1309, 2013.

[3] Khaldoon Al-Zoubi and Gabriel Wainer. Interfacing and coordination for a DEVS simulation protocol standard. In *Proceedings of the 2008 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*, pages 300–307, 2008.

[4] Osman Balci. The implementation of four conceptual frameworks for simulation modeling in high-level languages. In *Proceedings of the 20th Conference on Winter Simulation*, pages 287–295, 1988.

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

[6] Fernando J. Barros. Modeling formalisms for dynamic structure systems. *ACM Transactions on Modeling and Computer Simulation*, 7:501–515, 1997.

[7] Fernando J. Barros. Abstract simulators for the DSDE formalism. In *Proceedings of the 30th conference on Winter simulation*, pages 407–412, 1998.

[8] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13:31–39, 2011.

[9] Jean-Sébastien Bolduc and Hans Vangheluwe. The modelling and simulation package PythonDEVS for classical hierarchical DEVS. Technical report, McGill University, 2001.

[10] Matías Bonaventura, Gabriel Wainer, and Rodrigo Castro. Advanced IDE for modeling and simulation of discrete event systems. In *Proceedings of the 2010 Spring Simulation Multiconference*, pages 125:1–125:8, 2010.

[11] Laurent Capocchi, Jean François Santucci, Bastien Poggi, and Celine Nicolai. DEVSimPy: A collaborative python software for modeling and simulation of DEVS systems. In *Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 170–175, 2011.

[12] Bin Chen and Hans Vangheluwe. Symbolic flattening of DEVS models. In *Summer Simulation Multiconference*, pages 209–218, 2010.

[13] Gilbert Chen and Boleslaw K. Szymanski. Lookback: a new way of exploiting parallelism in discrete event simulation. In *Proceedings of the 16th workshop on Parallel and distributed simulation*, pages 153–162, 2002.

[14] Gilbert Chen and Boleslaw K. Szymanski. Four types of lookback. In *Proceedings of the 17th workshop on Parallel and distributed simulation*, pages 3–10, 2003.

[15] Gilbert Chen and Boleslaw K. Szymanski. DSIM: scaling time warp to 1,033 processors. In *Proceedings of the 37th conference on Winter simulation*, pages 346–355, 2005.

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

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

[18] President's Information Technology Advisory Committee. Computational science: ensuring america's competitiveness. *Report to the President*, 2005.

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

[20] Richard M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, pages 30–53, 1990.

[21] Richard M. Fujimoto. Performance of time warp under synthetic workkloads. In *Proceedings of the SCS Multiconference on Distributed Simulation*, 1990.

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

[23] Ezequiel Glinsky and Gabriel Wainer. DEVStone: a benchmarking technique for studying performance of DEVS modeling and simulation environments. In *Proceedings of the 2005 9th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*, pages 265–272, 2005.

[24] Ezequiel Glinsky and Gabriel Wainer. New parallel simulation techniques of DEVS and Cell-DEVS in CD++. In *Proceedings of the 39th annual Symposium on Simulation*, pages 244–251, 2006.

[25] Jan Himmelspach, Roland Ewald, Stefan Leye, and Adelinde M. Uhrmacher. Parallel and distributed simulation of parallel DEVS models. In *Proceedings of the 2007 spring simulation multiconference - Volume 2*, pages 249–256, 2007.

[26] Jan Himmelspach and Adelinde M. Uhrmacher. Sequential processing of PDEVS models. In *Proceedings of the 3rd European Modeling % Simulation Symposium*, pages 239–244, 2006.

[27] Jan Himmelspach and Adelinde M. Uhrmacher. Plug'n simulate. In *Simulation Symposium*, pages 137–143, 2007.

[28] Xiaolin Hu and Bernard P. Zeigler. Linking information and energy - activity-based energy-aware information processing. *Simulation*, 89(4):435–450, 2013.

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

[30] Shafagh Jafer, Qi Liu, and Gabriel Wainer. Synchronization methods in parallel and distributed discrete-event simulation. *Simulation Modelling Practice and Theory*, 30:54–73, 2013.

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

[32] Ki Hyung Kim, Yeong Rak Seong, Tag Gon Kim, and Kyu Ho Park. Distributed optimistic simulation of hierarchical DEVS models. In *Summer Computer Simulation Conference*, pages 32–37, 1995.

[33] Ki Hyung Kim, Yeong Rak Seong, Tag Gon Kim, and Kyu Ho Park. Distributed simulation of hierarchical DEVS models: Hierarchical scheduling locally and time warp globally. *Transactions of the Society for Computer Simulation International*, pages 135–154, 1996.

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

[35] Jasna Kuljis and Ray J. Paul. A review of web based simulation: whither we wander? In *Proceedings of the 32nd conference on Winter simulation*, pages 1872–1881, 2000.

[36] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565, 1978.

[37] Wan Bok Lee and Tag Gon Kim. Simulation speedup for DEVS models by composition-based compilation. In *Summer Computer Simulation 2003*, pages 395–400, 2003.

[38] Will Leland and Teunis J. Ott. Load-balancing heuristics and process behavior. *SIGMETRICS Performance Evaluation*, 14(1):54–69, May 1986.

[39] 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*, pages 183–188, 2011.

[40] Qi Liu and Gabriel Wainer. Lightweight time warp - a novel protocol for parallel optimistic simulation of large-scale DEVS and Cell-DEVS models. In *Proceedings of the 2008 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*, pages 131–138, 2008.

[41] Qi Liu and Gabriel Wainer. A performance evaluation of the Lightweight Time Warp protocol in optimistic parallel simulation of DEVS-based environmental models. In *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, pages 27–34, 2009.

[42] Friedemann Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18:423–434, 1993.

[43] Alexandre Muzy, Laurent Capocchi, and Jean François Santucci. Using activity metrics for DEVS simulation profiling, 2014. ACTIMS'14.

[44] Alexandre Muzy, Eric Innocenti, Antoine Aiello, Jean-François Santucci, and Gabriel Wainer. Specification of discrete event models for fire spreading. *Simulation*, 81(2):103–117, 2005.

[45] Alexandre Muzy and James J. Nutaro. Algorithms for efficient implementations of the DEVS & DSDEVS abstract simulators. In *1st Open International Conference on Modeling and Simulation (OICMS)*, pages 273–279, 2005.

[46] Alexandre Muzy, James J. Nutaro, Bernard P. Zeigler, and Patrick Coquillard. Modeling and simulation of fire spreading through the activity tracking paradigm. *Ecological Modelling*, 219:212–225, 2008.

[47] Alexandre Muzy, Luc Touraille, Hans Vangheluwe, Olivier Michel, Mamadou Kaba Traoré, and David R. C. Hill. Activity regions for the specification of discrete event systems. In *SpringSim/TMS-DEVS*, pages 176–182, 2010.

[48] Alexandre Muzy, Franck Varenne, Bernard P. Zeigler, Jonathan Caux, Patrick Coquillard, Luc Touraille, Dominique Prunetti, Philippe Caillou, Olivier Michel, and David R. C. Hill. Refounding of the activity concept? towards a federative paradigm for modeling and simulation. *Simulation*, 89(2):156–177, 2013.

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

[50] Alexandre Muzy and Bernard P. Zeigler. Introduction to the activity tracking paradigm in component-based simulation. *The Open Cybernetics and Systemics Journal*, pages 48–56, 2012.

[51] Alexandre Muzy Muzy, Rajanikanth Jammalamadaka, Bernard P. Zeigler, and James J. Nutaro. The activity-tracking paradigm in discrete-event modeling and simulation: The case of spatially continuous distributed systems. *Simulation*, 87(5):449–464, 2011.

[52] Patrick Nicolas. Meijin++, reference manual, 1991.

[53] James J. Nutaro. On constructing optimistic simulation algorithms for the discrete event system specification. *ACM Transactions on Modeling and Computer Simulation*, 19:1:1–1:21, 2009.

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

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

[56] Martin Potier, Antoine Spicher, and Olivier Michel. Topological computation of activity regions. In *Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '13, pages 337–342. ACM, 2013.

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

[58] José L. Risco-Martín, Alejandro Moreno, J. M. Cruz, and Joaquín Aranda. Interoperability between DEVS and non-DEVS models using DEVS/SOA. In *Proceedings of the 2009 Spring Simulation Multiconference*, pages 147:1–147:9, 2009.

[59] Behrokh Samadi. *Distributed simulation, algorithms and performance analysis*. PhD thesis, University of California, 1985.

[60] Jean François Santucci and Laurent Capocchi. Implementation and analysis of devs activity-tracking with devsimpy. *ITM Web of Conferences*, 1, 2013.

[61] Hessam S. Sarjoughian and Yu Chen. Standardizing devs models: an endogenous standpoint. In *Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium*, pages 266–273, 2011.

[62] Chungman Seo, Bernard P. Zeigler, Robert Coop, and Doohwan Kim. DEVS modeling and simulation methodology with ms4me software. In *Symposium on Theory of Modeling and Simulation - DEVS (TMS/DEVS)*, 2013.

[63] Moon Gi Seok and Tag Gon Kim. Parallel discrete event simulation for DEVS cellular models using a GPU. In *Proceedings of the 2012 Symposium on High Performance Computing*, pages 11:1–11:7, 2012.

[64] Hongyan Song. Infrastructure for DEVS modelling and experimentation. Master's thesis, School of Computer Science, McGill University, 2006.

[65] Yi Sun and Xiaolin Hu. Partial-modular DEVS for improving performance of cellular space wildfire spread simulation. In *Proceedings of the 40th Conference on Winter Simulation*, pages 1038–1046, 2008.

[66] Yi Sun and James J. Nutaro. Performance improvement using parallel simulation protocol and time warp for DEVS based applications. In *Proceedings of the 2008 12th IEEE/ACM International Symposium on Distributed Simulation and Real-Time Applications*, pages 277–284, 2008.

[67] Eugene Syriani, Hans Vangheluwe, and Amr Al Mallah. Modelling and simulation-based design of a distributed DEVS simulator. In *Proceedings of the Winter Simulation Conference*, pages 3007–3021, 2011.

[68] Luc Touraille, Mamadou K. Traoré, and David R. C. Hill. Enhancing DEVS simulation through template metaprogramming: DEVS-MetaSimulator. In *2010 Summer Simulation Multiconference*, pages 394–402, 2010.

[69] Mamadou K. Traoré. SimStudio: A next generation modeling and simulation framework. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, pages 67:1–67:6, 2008.

[70] Alejandro Troccoli and Gabriel Wainer. Implementing Parallel Cell-DEVS. In *Annual Simulation Symposium*, pages 273–280, 2003.

[71] Adelinde M. Uhrmacher. Dynamic structures in modeling and simulation: a reflective approach. *ACM Transactions on Modeling and Computer Simulation*, 11:206–232, 2001.

[72] Yentl Van Tendeloo. Research internship 1: Optimizing the PythonDEVS simulator. Technical report, University of Antwerp, 2013.

[73] Yentl Van Tendeloo. Research internship 2: Distributed and parallel DEVS simulation. Technical report, University of Antwerp, 2013.

[74] Yentl Van Tendeloo and Hans Vangheluwe. Logisim to devs translation. In *Proceedings of the 2013 IEEE/ACM 17th International Symposium on Distributed Simulation and Real Time Applications*, pages 13–20, 2013.

[75] Yentl Van Tendeloo and Hans Vangheluwe. Activity in PythonPDEVS. In *Proceedings of ACTIMS 2014*, 2014. *(to appear)*.

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

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

[78] Voon-Yee Vee and Wen-Jing Hsu. Pal: a new fossil collector for time warp. In *Proceedings of the 16th workshop on Parallel and distributed simulation*, pages 35–42, 2002.

[79] Gabriel Wainer. CD++: a toolkit to develop DEVS models. *Software: Practice and Experience*, 32(13):1261–1306, 2002.

[80] Gabriel Wainer and Norbert Giambiasi. Discrete event modeling and simulation technologies. chapter Timed cell-DEVS: modeling and simulation of cell spaces, pages 187–214. Springer-Verlag New York, Inc., 2001.

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