A USABLE DEVS ENVIRONMENT

DESIGNING AN EASY-TO-USE DEVS GENERATION ENVIRONMENT WITH SUPPORT FOR AN EXPANDABLE LIBRARY

Master Thesis nominated to obtain the degree of
Master in Computer Science : Software Engineering

DECKERS MICHAËL

PROF. DR. VANGHELUWE, HANS : PROMOTOR

University of Antwerp

2015 - 2016

Deckers Michaël: *A Usable DEVS Environment,* Designing an easy-to-use DEVS generation environment with support for an expandable library, Master Thesis, © 2016

SUPERVISORS:
Prof. Dr. Vangheluwe, Hans : promotor
Van Tendeloo, Yentl : supervisor

LOCATION:
Belgium

I dedicate this thesis to my family. In particular my mother and father who have always supported me and my sister who stood up for me in times of trouble and always pushes me to make the most of myself.

ABSTRACT

DEVS[16] is very powerful modeling and simulation formalism, but up until now, there is no one tool that does everything we expect from it perfectly. As such, I set out to at least start the implementation of such a tool.

Before work started on the thesis tool itself, I had investigated and compared, in a preliminary research[4], a range of existing DEVS generation solutions. From this result a list of "usability traits" was extracted, which would serve as a guideline for the design and construction of a new tool.

These usability traits are classified under eight different categories: availability, *installation*, documentation, general fit, *interface*, *model design*, *library* and simulation. Not all of these categories have gotten the same amount of attention. The traits in the categories listed in italic were seen as the key traits, which would be important to make the tool functional (for the generation of DEVS), and to show off the potential future capabilities of the tool, should it be developed further. The other traits were, in the scope of this thesis, considered more as nice-to-haves.

The tool is implemented in three main parts: the Model, the Compiler and the GUI. Each part is responsible for an important and secluded part of the DEVS generation and simulation process:

- **The Model** is the Java back-end that stores all DEVS components (Atomic DEVS, Coupled DEVS and DEVS Messages) and is able to save them in the "proprietary syntax". This syntax was designed as an easy to read and understand way to design DEVS components. If one wanted, he could easily use just the syntax, the Compiler and PythonPDEVS[15] (PyPDEVS) for the generation and simulation of DEVS models, and not use the Model or the GUI at all. This would of course ignore two-thirds of the application, so doing so is not recommended, as using the GUI (which in turn uses the Model) does allow the generation of DEVS models in the proprietary syntax to (theoretically) be faster and easier.
- **The Compiler** translates the proprietary syntax into a Python file that can serve as input for the PyPDEVS simulation kernel.
- **The GUI** is the main point of interaction between the user and the actual DEVS. The GUI allows the user to efficiently create DEVS models in the proprietary syntax (without necessarily needing to know the internal layout and structure), compile the entire project (with all DEVS components) with a single click of a button, and run the PyPDEVS simulation internally with a minimum of required settings. The results and output of the simulation are shown to the user from within the GUI.

The creation of the tool is only one of the two major topics of this thesis. The second is the design of a DEVS library, and to add a collection of DEVS library blocks to it

already. The selection of blocks that are interesting for use in DEVS was also composed in the preliminary research, this time by looking at both the existing DEVS tools that implement a library, but also at other modeling formalisms. A total of 45 blocks were found in this research, of which 32 were eventually implemented in DEVS for this thesis.

All implemented blocks can be categorized in seven categories: mathematical blocks (sum, inverse, ...), logic gate blocks (or, xor, and, ...), generator blocks, queueing blocks (FIFO, circular, ...), delay blocks, statistical blocks and data and model manipulation blocks (batch, combine, ...). All these blocks have been kept as generic as possible, to maximize the amount of situations that can be used in.

The thesis concludes by evaluating the tool and library by comparing it to its existing counterparts. This shows that in the future there is definitely a place for both the tool and the library, but further work is necessary as they are both still in the prototype stage.

DUTCH ABSTRACT - NEDERLANDSE SAMENVATTING

DEVS[16] is een krachtig modelleer-, en simulatie-formalisme, maar op het moment van schrijven is er geen enkele tool perfect doet wat we ervan verwachten, laat staan dat ze ook nog eens een bibliotheek en een krachtige simulatie-omgeving zouden hebben. Om deze reden was het doel van deze thesis om op z'n minst te starten aan de implementatie van zo'n applicatie.

Voordat het werk aan de eigenlijke applicatie werd gestart heb ik eerste, in een voorafgaand onderzoek[4] een selectie van bestaande DEVS omgevingen onderzocht en vergeleken. Het resultaat hiervan leidde tot een lijst van "gebruiksvriendelijkheid eigenschappen", die zou dienen als een richtlijn voor het design en de constructie van een nieuwe applicatie.

Al deze eigenschappen zijn onderverdeeld in acht verschillende categorieën: beschikbaarheid, *installatie*, documentatie, *gebruikersinterface*, *model ontwerp*, *bibliotheek* en simulatie. Niet al deze categorieën kregen even veel aandacht, aangezien sommigen nu eenmaal belangrijker waren dan anderen voor het doel van deze thesis. De schuingedrukte categorieën bevatten de eigenschappen die als essentieel worden aanzien. Deze moeten ervoor zorgen dat de applicatie functioneel is (voor het genereren van DEVS), en om potentiële toekomstige capaciteiten van de applicatie in de verf te zetten. De andere eigenschappen waren gezien het doel van deze thesis eerder "nice-to-haves".

De implementatie van de tool bestaat uit drie grote delen: het Model, de Compiler en de GUI. Elk deel is verantwoordelijk voor een belangrijk en afgescheiden deel van het DEVS generatie en simulatie proces.

- **Het Model** bevat de Java structuren die alle DEVS componenten opslaan (Atomic DEVS, Coupled DEVS en DEVS Messages) en kan deze opslaan in de "specifieke syntaxis". Deze syntaxis is ontwerpen als een eenvoudig te lezen en te begrijpen manier om DEVS componenten voor te stellen. Moest dit gewenst zijn, dan zouden enkel de syntaxis, de Compiler en PythonPDEVS[15] (PyPDEVS) volstaan om DEVS modellen te genereren en de simuleren. Op deze manier worden wel twee grote delen van de applicatie, namelijk het Model en de GUI, genegeerd, en het is dus niet aangeraden. De GUI en het Model samen maken namelijk het genereren van DEVS (theoretisch gezien) veel sneller en makkelijker.

- **De Compiler** vertaalt de DEVS modellen van de specifieke syntaxis naar een enkel Python bestand, dat kan dienen als de invoer voor de PyPDEVS simulator.

- **De GUI** is waar de interactie tussen de gebruiker en de DEVS modellen en simulatie echt gebeurt. De GUI laat the gebruiker toe efficient DEVS modellen te creÃ«ren in de specifieke syntaxis (zonder dat deze echt kennis over de interne layout van deze syntaxis moet hebben). Het compileren van een project (met alle

DEVS componenten) kan met een enkele druk op een knop, en ook de simulatie kan intern gestart worden met een minimum aan vereiste instellingen. De uitvoer en resultaten van de simulatie worden ook van in de GUI aan de gebruiker getoond.

Het creëren van de applicatie is slecht een van de twee hoofdzakelijke delen van deze thesis. Het tweede is het ontwerpen van een DEVS bibliotheek, en het toevoegen van geïmplementeerde modellen aan deze bibliotheek. De selectie van welke blocks interessant zijn om geïmplementeerd te worden gebeurde ook reeds in het voorafgaande onderzoek. Dit maal werd er naar zowel bestaande DEVS omgevingen, als ook andere modelleer formalismes gekeken. In totaal werden in dit onderzoek 45 DEVS blokken gevonden, waarvan er 32 effectief geïmplementeerd werden in deze thesis.

Alle geïmplementeerde blokken kunnen gecategoriseerd worden in zeven types: wiskundige blokken (som, inverse, ...), logische poort blokken (or, xor, and, ...), generator blokken, queue blokken (FIFO, circulair, ...), vertraging blokken, statistische blokken en data en model manipulatie blokken (batch, combineer, ...). Al deze blokken werden zo generisch mogelijk gehouden, zodat ze zouden kunnen toegepast worden in zoveel mogelijk verschillende situaties.

De thesis concludeert door het evalueren van de applicatie en de bibliotheek door ze te vergelijken met hun reeds bestaande tegenhangers. Dit toont aan dat er in de toekomst zeker een plaats kan zijn voor zowel de applicatie als de bibliotheek, maar dat toekomstig werk nog wel nodig is, aangezien ze beide nog in de prototype fase zitten.

We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

— Donald E. Knuth [7]

ACKNOWLEDGEMENTS

First of all, many thanks go out the my supervisor Yentl. He has helped me tremendously with giving advice and tips, without which the completion of this thesis in its current form would never have been possible.

Also, thanks to my family and friends, who never made an issue of the time I spent working on my thesis and the rest of my studies. As well as allowing me to pursue my dreams and provide help wherever possible. Thank you very much for the understanding and the never-ending support.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

1 INTRODUCTION

The title of this thesis is "A Usable DEVS Environment", which can be interpreted in the following way: The goal of this thesis is to improve on the current state-of-art when it comes to environments and tools designed for the creation and simulation of DEVS models. The envisioned end result is a newly created tool that combines the best usability aspects of all tools currently available.

The first objective is to learn exactly what "usability" means in the context of a DEVS development. To do this, a selection of the DEVS creation tools that are available today was made, and these were evaluated and compared to each other. This was all done in a research project prior to the start of this thesis. This research is not technically part of it, but was performed for the sole purpose of this thesis.

One important thought to note is that usability in the context of this thesis is split up into two major parts: the usability of the tool itself, and the usability increase by having some of the work taken away from the user.

USABILITY OF THE TOOL

The usability of the tool comprises the design elements of the tool itself. High level examples of such elements are: the program's looks, the flow of operation, how easy it is to learn to work with it, how much time is lost on non-constructive actions when using the tool, and so on.

These elements together define how fast and easy it is for a user to create a DEVS model. The goal of the tool is obviously to score high on all these elements. Chapter 4 discusses how the design of the tool is formed to achieve this goal. Chapters 7 to 10 of show how the design was eventually implemented, and gives a idea of the look and feel of the tool. After completion, the tool was evaluated by comparing to those that inspired it. The result of this evaluation can be found in Chapter 12

RELIEVING THE USER OF WORK

What better way of speeding up the process of creating DEVS models, than to eliminate some of the work the user has to do? A very popular way of doing this in many other contexts is the implementation of libraries that contain very frequently used functionality. These libraries can be easily shared, so it is relatively easy for users to benefit from the prior work of others. As a result, the more users there are that use the library, the more users there are that might contribute to the library, and the more functionality the library will contain. A library with more functionality will again attract more users, and the spiral continues.

The obvious solution is thus to implement a library for the development of DEVS. A good way to start building the library is to add some generic elements to it that can be used in a broad variety of situations, as such maximizing the amount of users that can benefit from it. Chapters 5 and 6 discuss how this is proposed to be achieved and which blocks would be a good fit. The implementation of both the library itself and the elements in it are discussed in Chapter 11. The implemented library elements are evaluated in Chapter 13.

DEVS

For readers of this text that are not familiar with the DEVS formalism, a short introduction is included in the next Chapter 2.

2

THE DEVS FORMALISM

The DEVS formalism was originally created by Zeigler[17] to provide modeling and simulation of discrete-event systems. DEVS support a continuous time base, which means that the advancement of time is not subject to minimal size steps. In practice, the time could thus be any point in $\mathbb{R}$. What makes DEVS discrete-event, is the fact that between two points in time, only a *finite* number of *events* can occur. Only these events can change the state of the system. In between events, the state of the system may not change (which would only possible if the formalism had a continuous time base, which DEVS does not).

The DEVS formalism can be completely explained by discussing the two levels of the structure: Atomic DEVS and Coupled DEVS, both of which are called DEVS components.[16]

ATOMIC DEVS

The Atomic DEVS is the lowest level building block of a DEVS. A single Atomic DEVS block describes the behaviour of the part of the system that it represents. An atomic DEVS modifies the state of the system through the use of internal states and deterministic transitions between those states. An Atomic DEVS block can be represented in the following way:

$$\text{Atomic DEVS} \equiv \langle S, ta, \delta_{int}, X, \delta_{ext}, Y, \lambda \rangle$$

Where:

- S: a set of sequential *states*. Exactly one state is always current, and which state will become current is deterministically defined by the transitions (which will be introduces later).
- ta: the *time advance*. This time indicates how much simulation time has to pass before the next internal transition will fire. Which transition fires is decided in the *internal transition function*. The *time advance* should always be a number greater than or equal to zero.
- δ_{int}: the *internal transition function*. When the *time advance* has passed, this function decides, based on the internal state of the block, which state will become the next current state.
- X: the *input set*. This set contains the incoming events for a specific time.
- δ_{ext}: the *external transition function*. Whenever an external event is received, the autonomous behaviour of the block is stopped and it is up to the *external transition*

function to specify which state will become current, based on the internal state of the block and the incoming event.

Y: the *output set*. This set contains the output events that are generated whenever an internal transition occurs.

λ: the *output function*. This function generates the output event whenever the state of the block is changed through an internal transition. It does this based on the internal state of the block prior to the transition.

COUPLED DEVS

A Coupled DEVS block describes the entirety or a part of the system by forming a network of connected DEVS components. These components can be either Atomic DEVS or other Coupled DEVS. A Coupled DEVS block is represented as follows:

$$\text{Coupled DEVS} \equiv \langle X_{self}, Y_{self}, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\}, select \rangle$$

Where:

X_{self}: the *input set*. This set contains the inputs of the Coupled DEVS block.

X_{self}: the *output set*. This set contains the outputs of the Coupled DEVS block.

D: the *component references set*. This set contains references (names) for all its subcomponents.

$\{M_i\}$: the *set of subcomponents*, where $i \in D$. Each subcomponent can be an Atomic or Coupled DEVS block (since DEVS is closed under coupling[16]).

$\{I_i\}$: the *set of influencees*, where $i \in D \cup self$, $self$ being the Coupled DEVS block itself. It represents the set of subcomponents (or itself) that are influenced by component i.

$\{Z_{i,j}\}$: the *set of output-to-input translation functions*. This maps the connections between of both subcomponents with other subcomponents or subcomponents with the Coupled DEVS block itsef.

$select$: the *tie-break function*, which chooses between events that want to happen simultaneously.

In plain English this means that a Coupled DEVS is a hierarchical structure that can contain a number of subcomponents (which can be either Atomic or Coupled DEVS). It creates an internal network of these subcomponents, by linking them together using their outputs and inputs, thus allowing events to be sent between them. The Coupled DEVS block itself can also have inputs and ouputs, which allows the subcomponents to communicate with the rest of the system.

Part I

DESIGN

The design of a software product is a critical step, during which ideas get transformed into concrete plans, on which the eventual development will be based. In this part, I recapitulate on research I had previously done and discuss how the results of this research can be utilized during the creation of the new DEVS tool (Chapters 3 and 4) and library (Chapters 5 and 6).

3

RECAPPING PRIOR RESEARCH: COSMETIC AND PRACTICAL

As was discussed already in the outline, the research[4] that was done prior to this thesis consisted of two major parts. This chapter will discuss the first (and largest) of these two, namely the investigation of existing tools based on varying usability traits.

In this first part, I evaluated a selection of tools (*DEVSimPy*[2], *VLE*[10], *DEVS-Suite*[6], *MS4Me*[11], *CD++Builder*[3], *PowerDEVS*[1] and *AToMPM*[12] (more specifically DEVSDebugger [14]), all of which are designed to implement DEVS (AToMPM was not designed specifically for DEVS, but does support it). Through using all of them extensively trying to create a working model and concurrently writing down my thoughts and experiences as objectively as possible, I was able to compile an extensive list of usability traits that will be re-introduced below. Here, this list will serve as a basis for a new tool, and not so much as an in depth evaluation and comparison of the existing tools (like it is in the original research report), since only the combined best practices are of importance for the design of a new tool. Thus, some thoughts here might vary a bit from the conclusions formed in the original report. It is important to remember that the original research was, in fact, created to be a standalone whole, which could in theory be used as a basis of information for other future projects that discuss modeling tools of any formalism.

3.1 FULL LIST OF USABILITY TRAITS

In the next section I will (almost literally) repeat the list of usability traits that were introduced and discussed in the research. These traits combined contain the most important pieces of information that were gathered during the entire research. To bring some form of categorization and division in the long list of traits, they are all classified under eight different categories, each of which will get its own section.

How all the inspected tools compare on these traits can be found in Chapter 12, where they are all put together into a table per category. For each trait and tool combination it is indicated whether the tool supports the trait or not.

3.1.1 *Availability*

Traits that are concerned with the level of availability and activity. Given the academic nature of this thesis, these traits are of less importance to a new tool, but we will include them for the sake of completeness, and they can increase in importance should they be applied in the future for further development or a functional release of the new tool.

- **Website**: indicates whether the tool has an active website that hosts the download, information about the tool, the authors, etc. For multiple tools, this website is nothing more than a GitHub or SourceForge page.
- **Website up to date (< 1 year)**: indicates whether the website has undergone recent changes (within the last year) or updates. This does not have to mean that the tool is still under development, but it might indicate that it is at least still being used.
- **Simple download**: indicates whether it is straightforward to download the version of the tool, but also any dependencies required for the tool.
  - DEVSimPy required extra dependencies that had to be matched to specific versions of already installed software, making the download and installation more difficult than it should.
  - MS4Me requires you to request the download link for a trial version and is as such not a simple download.
- **Open Source**: indicates whether the tool comes with the sources, such that these can be modified freely.
- **Free**: indicates whether the tool is free to use (paid with free trial is indicated as not free).
- **Active project**: indicates whether the tool has undergone any changes in the last year.

3.1.2 *Installation*

Traits concerning the ease of installation, and possible existence of versions (or platform independence) across different operating systems.

- **No installation required**: indicates that the tool can be ran without running any installer, either for the tool or any dependencies.
- **Platform independent**: indicates whether the tool is platform independent, meaning the same version can be ran on any platform, as is the case with most Python or Java tools/files.
- **Windows**: indicates whether the tool is available for Windows.
- **Linux**: indicates whether the tool is available for Linux.
- **Mac**: indicates whether the tool is available for Max. An "(x)" indication means it should work on Mac based on the same installation files as Linux, but there is no version specifically for Mac.
- **Limited external dependencies**: indicates that the extra dependencies required to run the tool did not hamper or slow down the installation.
- **No manual configuration**: indicates that after the installation, the tool is ready to run without having to manually configure anything.

- **Installation tutorial/manual available**: indicates whether a tutorial (written or video) that explains the process of installation is available.
- **First party installation tutorial/manual available**: indicates whether the tutorial is available on the tool's own website.

3.1.3 *Documentation*

Traits concerning the availability of documentation for the installation of use of the different tools, and, to lesser extend, the quality of this documentation.

- **User manual (English) available**: indicates whether a user manual is available, either on the tool's website or elsewhere.
- **Academic paper(s) (English) available**: indicates whether papers discussing the tool (not merely using the tool) are available on line.
- **Academic paper(s) (English) available on website**: indicates whether papers discussing the tool are available for download (or at least referenced) on the tool's website.
- **Tutorials available**: indicates whether any tools are available online (e.g. on Youtube).
- **First party tutorial available**: indicates whether the tool's website has one or more tutorials available.
- **First party example project**: indicates whether example projects or models are available for download on the website.

3.1.4 *General Fit*

Some more global (and admittedly more subjective) traits that discuss the potential use cases for the tools.

- **Programming language knowledge necessary**: indicates whether any (advanced) knowledge of a standard programming language (C++, Java or Python) is required to make anything more than the most basic elements. The "(x)" indication means that there is some way to design a working model without having to know any programming languages, be it that they use their own syntax, provide a library with basic models, or have such an easy template that any knowledge necessary is trivial.
- **Fit for academic purpose**: indicates (subjectively) whether the tool is mature and useful enough to be used in an academic environment. For a tool to qualify for this, it has to be stable (but can have some minor bugs) and it has to provide working models and simulation. It may expect prior knowledge from the user, and the installation process does not have to be perfect. Being open source is a plus as academic people may need (and have the expertise) to modify the tool to their own specific field of application.

- **Fit of professional purpose**: indicates (subjectively) whether the tool is stable and good enough to be used in a professional environment. Bugs need to be extremely rare, the tool needs to be relatively generic, such that it can be applied to a variety of fields. Installation and upkeep need to be low as professionals might not have the knowledge, time or funds to have downtime. Simulation has to be well developed, complete and produce clear results.
  - CD++Builder has an "(x)" indication because it has all the right features, but has not been supported since 2011.

3.1.5 *Interface*

Traits that discuss the actual visual user interface of tools by evaluating cosmetic elements and functional design choices. Although these cosmetic elements might sound as if they are subjective traits, there are some general GUI design guidelines [8] that a developer or designer should consider following in order to create an objectively good user interface.

- **Clearly laid-out interface (no tutorial needed)**: indicates (subjectively) that the user interface has all the right things in the right places. Criteria for a tool to satisfy this feature include but are not limited to:
  - Menus and options are in the right place (top status bar) and labeled similarly as other tools (and the OS).
  - Buttons on the toolbar have an image that corresponds to the task it does, and hovering on the button shows extra information.
  - The working environment (either text editor or modeling pane) take up the majority of space.
  - Console, error handling, etc, ... are at the bottom of the place, in accordance with most development/modeling tools.
  - The (optional) project explorer is to the left of the main editor, in accordance with most development/modeling tools.
  - In general, when playing around with the tool, most of the features and functionalities should explain themselves.

  A tool does not need to satisfy all individual requirements to satisfy globally, if a tool feels familiar and the learning curve is not too high, it is fine. An "(x)" indication means that the tool satisfies mostly subjectively, but might not do so for novice users (in the case of AToMPM) or those unfamiliar with Eclipse (MS4Me and CD++Builder).

- **Modern interface look**: indicates (subjectively) whether the tool looks modern enough for the year 2016. Standard Swing layout, a lot of gray-tones and unresponsive design limit some tools.

3.1.6 *Model Design*

Traits that discuss the design of the DEVS models within the tool, rather than the usability of the tool itself, but also how the tool's user interface aids the user in this design practice. These traits differentiate between text-oriented and visual-oriented editors and discuss just how practical and extensive the actual DEVS design is, and which extra functionality is implemented. These are especially important for the new tool, as these traits form the most important basis for usability regarding the actual design of DEVS, regardless of how cosmetically attractive or usable the tool itself is, and they are a key factor in the creation of a functional tool.

- **Visual based model design**: indicates that the main tool for modeling is graphical-based. Indications are a prominent drag-and-drop pane, options to adapt blocks visually and the fact that creating coupled DEVS is done through creating visual connections as opposed to textually add connections between blocks.
- **Logic implementation from within tool**: indicates that a block's logic code can be edited from within the tool (using either an included text editor or automatically opening the correct file in another local tool).
- **Majority of the screen taken up by design pane**: *self-explanatory*.
- **No unexpected visual editor quirks**: indicates that when designing in the tool, blocks do not suddenly resize, do not snap to places the user did not intend, links stay connected when connected blocks are moved, and other of such examples.
- **Intuitive visual editor controls**: indicates that, in general, the same sort of clicks and keyboard shortcuts are used in the tool as they would be in similar tools. For example: right clicking opens the context menu, double clicking opens properties or edit menu, resizing is done by dragging the corners, ... AToMPM fails for example because opening more information on a model requires the user to press shift + clicking on the object. Although this is usually easy to get used to, it is not intuitive.
- **High level of visual adaptability**: indicates that blocks or other objects can be visually modified in multiple ways, including but not limited to size, color and orientation.
- **Textual based model design**: indicates that the entire DEVS model can be created using only textual files. The tool then allows this to be made easier, or allows for simulation, but would not be necessary to create a syntactically correct model.
- **Visual representation of textual design**: indicates that the tool has a way to visually represent atomic and coupled models that have been designed textually.
- **Included textual editor**: indicates that the textual files can be generated from within the tool, without ever having to use any text editor.
- **Model validity checking**: indicates that the tool is capable of checking the validity of a model and (optionally) locate any errors should the model not be valid.

- **No manual logic recompilation after changes**: indicates that it is not necessary to recompile sources or models before they can be used within the tool for further design or simulation.
- **Exclusive use of existing programming languages**: indicates that the tool only uses existing programming languages to create or add logic to models.
- **Generic template auto-generated**: indicates that the tool automatically generates some sort of template for the logic of the model. This might mean, for example, creating a Java class with all methods predefined, such that the user only has to make the necessary additions. Tools can take it one step further and combine this with providing different fields for different transition functions (internal, external, output, ...), hiding all the code the user does not need to see.
- **Use of proprietary language**: indicates that the tool has its own syntax to some the entire (or part of the) model creation.
- **Syntax highlighting for proprietary language**: *self-explanatory*.
- **Proprietary language for atomic model**: indicates that a proprietary language/syntax is used to create atomic models.
- **Proprietary language for coupled model**: indicates that a proprietary language/syntax is used to connect different atomic (or hierarchical coupled) models.

3.1.7 *Library*

These are all traits that are concerned with the library (or lack there-of) of the tools. At this point in this report anything regarding libraries is not discussed in much detail, as the entire second part of the research (and the next chapter) revolve solely around library support.

- **Library support**: indicates that there is support for some sort of library within the tool, from which blocks or models can be used whilst creating a new model.
  - MS4Me has the "(x)" indication because it supports a library of existing models, but you cannot just include the blocks in a new project without additional software.
- **Expandable library**: indicates that blocks generated manually from the tool can be added to the existing library.
- **Online model repository**: indicates that the library has an online component from which models can be downloaded. Full example projects that can be downloaded and opened in the tool do not count, as they are not originally intended to be picked apart and used in other projects.
- **Included library with basic building blocks**: indicates that some basic building blocks (such as vector blocks, mathematical blocks, etc) are included with a fresh

install of the program. Input or output blocks designed to generate or log simulation results do not count, as they do not technically add to the "construction" of the model. This of course does not mean these blocks are not useful.

3.1.8 *Simulation*

Traits concerning the simulation functionality of DEVS tools. They discuss how well simulation has been implemented and which options and different forms of simulation are supported.

- **Simulation from within tool**: indicates that the tool itself provides the (most basic) functionality to simulate the model.
- **Advanced simulation controls**: indicates that the tool can do more than just "run" the simulation.
- **Complete run simulation**: indicates that the tool can run the full simulation (until infinity).
- **Step-by-step simulation**: indicates that the tool can run the simulation one or more steps at a time.
- **Partial run simulation**: indicates that the tool can run the simulation partially, up to a certain time or step.
- **Pause and resume simulation**: indicates that the tool can pause the simulation when it is running, and continue later on without having to reset the simulation.
- **Graphical representation of simulation**: indicates that the tool will show the simulation steps being executed (often by showing messages being passed from one object to another).
- **Live data shown during simulation**: indicates that changes in the model (values or data, often used for debugging) are shown and updated continuously whilst the simulation is running.
- **Data stored after simulation**: indicates that the changes that occurred during the simulation are stored in a file for future inspection or debugging (after the simulation has finished).
  - in PowerDEVS, specific blocks exist for the storage or plotting of data during the simulation.
- **Live log shown during simulation**: indicates that a live console logs information about the transformation. This information can timestamps of events, transitions that occur, etc.
- **Log stored after simulation**: indicates that the log messages that were generated during the simulation are stored in a file for future inspection or debugging (after the simulation has finished).

4

DESIGN CHOICES

It would be unrealistic to expect the new tool to implement all the features introduced in Chapter 3, and at the same time be better structured and have a modern and consistent user interface. Still, it is important to set high expectations and then focus on a few main parts. This way, possible future endeavours to continue work on the tool know what drove the choices so far, and what the envisioned end result is.

Realizing this, I set out to start building a tool with basic functionality and decided to add a limited selection of traits, giving preference to quality over quantity. This was further encouraged by the realisation that others (AutoDesk, with work around DesignDEVS[5]) had been doing similar work around the same time. I would not be able to realistically compete with their product and after consultation with my supervisor and promoter, we decided to focus more on the design of a DEVS library.

Consequently, choices had to be made. Not only to save time, but also because it would be rather unwise to spend valuable time designing parts of the program for which alternatives already exist and are implemented better than I could ever do in the limited time I have available, or doing things that others have done before, which do not add anything of interest to the current state-of-art.
The remainder of this chapter will discuss most of the choices that have been impactful in the end result.

4.1 AVAILABILITY, INSTALLATION AND DOCUMENTATION

As realization came that choices would have to be made in the development of a complete tool, most traits under the availability and installation part took a back seat to other functionalities, which would have more of an impact on the state-of-art. Considering the tool that will result from this thesis will be a prototype of some sorts, which will for the time being only developed further within a (probably) academic setting, extensive product websites or manuals will not be necessary, as it will be quite a while before the public will be able to benefit from the existence of these.

Things that *are* important, and as such will be taken into account throughout the implementation of the tool, are issues like limited dependencies, platform independence and limited to no manual configuration. The main reason for this is that these are all problems that will grow linearly (or even exponentially) with the growth of the project if they are not kept in check from the very start. The more dependencies and settings are required, the harder it gets for changes or additions to the software to be implemented. Similarly, maintaining platform independence is easier than introducing it.

Since the tool at this point is not made for distribution, the use for an extensive user manual is extremely limited. However, a short tutorial will be included in this text to familiarize those interested in evaluating this thesis or continuing the work.

4.2 GENERAL FIT AND INTERFACE

Again, it is important to note that this thesis will remain a work in progress and the end result is not a tool ready for deployment. As such the general fit will technically be academic, but in a sense that it will serve an academic purpose in the form of academic research, and not an academic application.

The lay-out needs a mostly functional lay-out, since in a prototype software product function over form is very applicable.

4.3 DEVS MODEL DESIGN

The first major choice, which has an impact on the entire rest of the tool, was how I envisioned the design of actual DEVS should happen. Knowing that I would be using an already existing DEVS simulation kernel (see next section (4.4)), I had to find a way that combined both the *Model Design* traits found above with the choice of simulation Kernel. I settled on a text-based DEVS creation environment. In order to limit the required knowledge of either DEVS, programming or the simulation kernel, I decided that for the definition and creation of DEVS models, I would create a custom, human readable, proprietary syntax based on inspiration gathered from both the DEVS formalism in AToMPM[12] and the proprietary syntax of MS4Me[11]. A human readable proprietary syntax would reduce the time to learn for novice users of Python or DEVS, and allows for an easy adaption should the simulation kernel change or be replaced with something different, since then only the compilation functions that translate the proprietary syntax into simulation kernel format need to be changed. Details about the syntax can be found in Part ii.

4.4 DEVS SIMULATION

One very easy-to-make choice (because I would never be able to create anything similar in functionality with the available time) was to use an already existing simulation kernel, namely PythonPDEVS[15] (PyPDEVS). This is a parallel and classic DEVS simulation kernel written in Python, designed originally as a master's thesis at the University of Antwerp by Yentl Van Tendeloo, my supervisor. PyPDEVS is already being used in other tools, for example DEVSimPy[2], one of the tools I discussed in the previous research.

Using this simulation kernel meant that only a minimum amount of time would actually have to be spent on designing and implementing a way of simulation. The work was consequently reduced to compiling the DEVS models that the user creates using the proprietary syntax into Python files that are interpretable by the PyPDEVS kernel.

Originally, the idea was to implement many of the *Simulation* traits, that way adding extending, albeit externally, the functionality gained by making use of the PyPDEVS simulation kernel. This time-consuming effort was later dropped in favor of the more important *library* plan. that will be introduced in Chapter 5.

4.5 DEVS LIBRARY

As mentioned before, the implementation of a library will be one of the major concerns in this master thesis, as it is clear that there is only a very limited support over the entire range of tools. A formal introduction and in depth discussion of the library part will be shown next, in Chapter 5.

5

RECAPPING PRIOR RESEARCH: LIBRARY COMPONENTS

As was touched on before, a big part of this thesis evolves around the implementation of a library containing DEVS blocks. Such a library can hugely impact the speed with which complex DEVS projects can be created through exchanging some tedious and repetitive work for predefined blocks, thus leaving more time available for the more project-specific parts of the model.

The same research that resulted in the list of traits that was used in the previous chapters, also documented an extensive search for useful and general DEVS blocks. A selection of sources were used to find these blocks, including (but not necessarily limited to) SimEvents[9], Extend[13] and PowerDEVS[1]. In the prior research[4] all of these blocks have been discussed in much more detail (about the internal structure, input ports, output ports, ...), so if any more details about any specific blocks are required, I suggest having a look at that document.

In this thesis text the list of library blocks that resulted from the research will be repeated in a more concise way. Here, only the name, a very short description containing the input and results will be shown, along with a possible source if the functionality of the block is anything above trivial. To keep some overview in the rather extensive list of blocks, they have been organized in a number of categories. Each section below discusses one of such categories and introduces each of the blocks that belong in this category.

5.1 MATHEMATICAL BLOCKS

Mathematical blocks are blocks that perform mathematical operations on its inputs in order to generate the desired output. These blocks only work with mathematical values, not any other types.

The mathematical blocks that were selected are listed in Table 1:

Table 1: Selected mathematical blocks

Mathematical Blocks	
Name	**Description**
SUM	the SUM block waits for input on both its incoming connections and immediately outputs the sum of each of the incoming values.
MULTIPLIER	the MULTIPLIER block waits for input on both its incoming connections and immediately outputs the product of each of the incoming values.

GAIN	the GAIN block waits for input on its single incoming connection and immediately outputs the value on this connection with a predefined value.
ABSOLUTE	the ABSOLUTE block waits for input on its single incoming connection and immediately outputs the absolute of the value on this connection.
NEGATIVE	the NEGATIVE block waits for input on its single incoming connection and immediately outputs the negative of the value on this connection.
INVERSE	the INVERSE block waits for input on its single incoming connection and immediately outputs the inverse of the value on this connection.
ROUND	the ROUND block waits for input on its single incoming connection and immediately outputs the rounded value on this connection. The amount of decimals is predefined in the block.
SQUARE	the SQUARE block waits for input on its single incoming connection and immediately outputs the square of the value on this connection.
SQRT	the SQRT block waits for input on its single incoming connection and immediately outputs the square root of the value on this connection.
POWER	the POWER block waits for input on both of its incoming connections (the value v and the exponent x) and immediately outputs v^x.
INTEGRATOR	the INTEGRATOR block outputs the integral of the value of its incoming connection with respect to time, it contains an internal initial condition. (Source: PowerDEVS[1])
DERIVATIVE	the DERIVATIVE block outputs the approximate derivative of the value of its incoming connection with respect to time. (Source: Simulink [1])

5.2 LOGIC GATE BLOCKS

A second category of blocks are the *Logic Gate Blocks*, that implement different types of logic gates. At this point, only the four basic logic gates have been implemented, as all more complex gates can be created as a combination of these. All these blocks require numerical input where anything that is 0 or smaller is seen as a negative signal, and anything larger than 0 is seen as a positive signal.

The logic gate blocks that were selected are listed in Table 2:

1 http://nl.mathworks.com/help/simulink/slref/derivative.html

Table 2: Selected logic gate blocks

Logic Gate Blocks	
Name	**Description**
OR	the OR block waits for input on both of its incoming connections and outputs a positive signal if either of or both of its incoming values are positive, otherwise the output is negative.
XOR	the XOR block waits for input on both of its incoming connections and outputs a positive signal if one, but not both of its incoming values are positive, otherwise the output is negative.
AND	the AND block waits for input on both of its incoming connections and outputs a positive signal if both incoming values are positive, otherwise the output is negative.
NOT	the NOT block waits for input on a single incoming connection and outputs a positive signal if the incoming value is negative, otherwise the output is negative.

5.3 GENERATOR BLOCKS

Generators create signals and send them into the model. The type of signal can vary, as well as the time at which they are sent. Obviously, it would be possible to define dozens of differently behaving generators, so the ones I chose to include are some that seemed particularly useful.

The generator blocks that were selected are listed in Table 3:

Table 3: Selected generator blocks

Generator Blocks	
Name	**Description**
GENERATOR (repeating)	the repeating GENERATOR block generates and outputs a signal, the value of which is predefined in the block, continuously at a fixed interval, which is also predefined.
GENERATOR (single)	the single GENERATOR block generates and outputs exactly one signal, the value of which is predefined in the block, at the start of the simulation.
GENERATOR (repeating, random)	the repeating random GENERATOR block generates and outputs a signal, the value of which is a random number, continuously at a fixed predefined interval.
PROGRAM	the PROGRAM block contains a predefined list of timestamps and a predefined list of items. It outputs an item of the list when its corresponding time equals the current simulation time.

5.4 QUEUEING BLOCKS

Queues are a very popular and powerful tool to extend the possibilities of a model. Just like with generators, many different types and variations of queues are imaginable, so I have limited myself to some of the most frequently used. Queues can store any type of value.

The queue blocks that were selected are listed in Table 4:

Table 4: Selected queue blocks

Queue Blocks	
Name	**Description**
FIFO Queue	the FIFO (First In, First Out) Queue block accepts and stores items. When requested to send a signal (either on a timed basis or on request), the item that has been stored the longest will be released. (Source: Extend[13])
LIFO Queue	the LIFO (Last In, First Out) Queue block accepts and stores items. When requested to send a signal (either on a timed basis or on request), the item that arrived the most recent will be released. (Source: Extend[13])
PRIORITY Queue	the PRIORITY Queue block accepts and stores items. Each of these items contains some sort of priority. When requested to send a signal (either on a timed basis or on request), the item with the highest priority (or that has been in the list the longest on equal priorities) will be released. (Source: Extend[13])
CIRCULAR Queue	the CIRCULAR Queue block accepts and stores items in a circular buffer of a predefined size. When the buffer is full, incoming items are either dropped, or the oldest item in the queue is overwritten. When requested to send a signal (either on a timed basis or on request), the item that has been stored the longest (and not overwritten) will be released.
MATCHING Queue	the MATCHING Queue block accepts and stores items. When requested to send a signal, the request signal contains a value that is to be matched with any of the values in the queue, the item that matches or contains the requested value and has been stored the longest (of all matching values) will be released. (Source: Extend[13])

5.5 DELAY BLOCKS

Delay blocks are designed to hold one ore multiple values for a specified amount of simulation time.

The delay blocks that were selected are listed in Table 5:

Table 5: Selected delay blocks

Delay Blocks	
Name	**Description**
DELAY	the DELAY block stores an incoming item for a predefined time (simulation time). When this time expires, the item is released. The time items have to be held can also be modified. (Source: Extend[13])
DELAY (Attribute)	the DELAY by attribute block stores an incoming item for a certain time. This time is an attribute of the incoming item, or the item itself. The name of this attribute is predefined. When this time expires, the item is released. (Source: Extend[13])
DELAY (Multiple)	the DELAY multiple block stores multiple items for a certain delay and thus acts as a timed queue. When the time expires, an item is released in a FIFO manner. The block can be modified into a size-limited server by setting a limit on the amount of items that can be stored. It will then drop any items that arrive if the server is full. (Sources: Extend[13]), Simulink [2]
DELAY (Multiple, Attribute)	the DELAY multiple by attribute block stores multiple incoming items for a certain time. This time is an attribute of the incoming item, or the item itself. The name of this attribute is predefined. When the time of any item currently present in the block expires, this item is released (no matter the arrival time or order). (Source: Extend[13])

5.6 STATISTICAL BLOCKS

Statistical blocks are designed primarily to inspect the state of the model throughout the simulation, however, they could just as well be used as an integral, functional part of the model.

The statistical blocks that were selected are listed in Table 6:

Table 6: Selected statistical blocks

Statistical Blocks	
Name	**Description**
SIMULATION TIME	the SIMULATION TIME block keeps track of the current simulation time and outputs this whenever a signal is sent through or the time is requested.

2 http://nl.mathworks.com/help/simevents/ref/entityserver.html

TIMER	the TIMER block calculates the simulation time that has passed since the timer has started (start signal) and stopped (stop signal). It outputs the result when the stop signal arrives.
COUNT ITEMS	the COUNT ITEMS block keeps track of how many items have arrived at the block. Arrived items are passed through. The result is output when a request signal is received.
COLLECTOR	the COLLECTOR block is an advanced version of the COUNT ITEMS block, that not only holds the number of items collected, but also stores all the items and the simulation time point at which they were received. Each individual stat, or all at the same time, can be requested through signals on different ports.
LIST SIZE	the LIST SIZE block outputs the size of an incoming list. This list can contain any type of values.
NUMERICAL LIST STATS	the NUMERICAL LIST STATS is an advanced list stat block that calculates a number of stats on an incoming list of numerical values. These stats are: size, average, median, maximum, minimum, standard deviation and variance. The individual stats can be requested through signals on different ports.

5.7 DATA AND MODEL MANIPULATION BLOCKS

These blocks are designed to manipulate items, both on an item-level (manipulating the internal values or attributes of an item) and a model-wide-level (merging, splitting, grouping, ... of multiple individual items).

The data and model manipulation blocks that were selected are listed in Table 7:

Table 7: Selected data and model manipulation blocks

Item-Level Manipulation Blocks	
Name	**Description**
CHANGE NUMERICAL ATTRIBUTE	the CHANGE NUMERICAL ATTRIBUTE block modifies a certain numerical attribute of an incoming item and outputs the modified item. The name of the attribute is predefined. The value can either be incremented, decremented, multiplied or divided by a predefined numerical constant. (Source: Extend[13])
ATTRIBUTE GET	the ATTRIBUTE GET block finds the attribute (the name of which is predefined) of an incoming item and outputs the outgoing item and the attribute.

ATTRIBUTE SET	the ATTRIBUTE SET block modifies the attribute (the name of which is predefined) of an incoming item by setting the value of this attribute to an internally stored value, after which the item is sent to the output again. The internally stored value is predefined but can be changed throughout the simulation with incoming values on a specific port.
ITEM REPLICATOR	the ITEM REPLICATOR block creates a predefined amount of copies of an incoming item. The original item is output on one port, the copies are output on another point. (Source: Simulink [3])
Model-Wide Manipulation Blocks	
Name	**Description**
COMBINE	the COMBINE block merges the signal arriving at two different input ports on a single output port. (Source: Extend[13])
BATCH	the BATCH block has three different inputs and for each of this input it has a predefined number. It creates a single list with items incoming on all the inputs. When each input has received an amount of items that is equal to its predefined number, a single list with the correct number of incoming values from each input is sent to the output. (Source: Extend[13])
UNBATCH	the UNBATCH block receives a list of items, usually from the BATCH block, and splits it back up into individual items. It has three outputs, corresponding to the three inputs of the BATCH block and also a predefined number for each of these outputs. This number decides how many of the items from the list should be sent to each of the three outputs. Technically, a BATCH and an UNBATCH block with the same predefined numbers should negate each other's effect.
INPUT SWITCH	the INPUT SWITCH block has multiple inputs, of which only one can be active. Only items that arrive at the active input are forwarded to the single output, other items are dropped. Which input is active is stored internally and is predefined but can be modified during simulation by sending a value to a dedicated input port. (Source: Simulink [4])
OUTPUT SWITCH	the OUTPUT SWITCH block has multiple outputs, of which only one can be active. Items that arrive at the input are forwarded only to the active output. Which output is active is stored internally and is predefined but can be modified during simulation by sending a value to a dedicated input port. (Source: Simulink [5])

3 http://nl.mathworks.com/help/simevents/ref/entityreplicator.html
4 http://nl.mathworks.com/help/simevents/ref/entityinputswitch.html
5 http://nl.mathworks.com/help/simevents/ref/entityoutputswitch.html

GATE	the GATE block either forwards or drops incoming items based on whether the internally stored gate is open (positive value) or closed (0 or negative value). The internal gate value is predefined but can be modified during simulation by sending a value to a dedicated input port. (Source: Extend[13], Simulink [6])

6 http://nl.mathworks.com/help/simevents/ref/entitygate.html

6

LIBRARY DISCUSSION

The goal for a big part of the thesis is to have a user-expandable library of already implemented DEVS blocks. So far, in most tools that have libraries, the developers decided to store entire models. There is a use for this, but it is still limited in granularity, since the bigger the model in a library, the more specific a use case has to be in order to actually use this model. Furthermore, due to the ongoing project of PyPDEVS at the University of Antwerp, it would be very useful to have a library based on this kernel, since then the blocks in this library can be immediately (or at least with minimal effort) applied in many tools or projects that already utilize PyPDEVS. Thus serving a use beyond this thesis itself.

Even though at this point in this text, we are still in the design phase of the library, work on the tool itself had already been underway for quite a while. As such, we decided to continue developing the tool until such time that the minimal usability was adequate to efficiently generate DEVS models.

6.1 LIBRARY DESIGN

The main objective in the implementation of the library structure itself (not the blocks within the library), is to minimize the time required to implement it, and at the same time not let the already developed tool go to waste. These two objectives mean that the tool will be upgraded to include a way of importing DEVS blocks that are created with the tool, as if they were part of a library.

Details about both the integration of this 'library' into the tool, but also about how the library blocks themselves are structured can be found in Part ii.

6.2 LIBRARY BLOCKS

An extensive list of theoretical library blocks can be found in the previous chapter, and since the change of direction throughout this thesis, the ambition is to implement as many of these blocks as possible within the time available. The time needed to create a DEVS block can fluctuate severely depending on the difficulty and functionality of the block. To see which blocks were eventually implemented, you can look ahead to Chapter 11.3

Exactly which blocks will get preference regarding being implemented is also unclear at this time, but I do believe that a mix of blocks from different categories would be the most desirable option, albeit not the most time-efficient, in order to maximize the ability to show off the potential benefits of the envisioned library of DEVS blocks.

Part II

IMPLEMENTATION

Now that that the design goals have been discussed in much detail, it is time to start actually performing constructive work by developing and implementing a functional prototype of the DEVS environment tool, after which this prototype will immediately be put to the test by using it to create a multitude of DEVS blocks for the envisioned library.

This part is structured in the same way: the first chapters (7, 8, 9 and 10) discuss the implementation of the tool, whereas the final chapter (11) documents the implementation of the library.

7

THE GENERAL STRUCTURE OF A DEVS ENVIRONMENT

A software project of this size cannot just be started on without any form of planning or structure, as the internal overview would be lost before the first successful run. This chapter breaks down the entire tool as it stands into a number of major pieces, each of which will be discussed in more detail in its own chapter.

Before doing all this, however, it is necessary to start at the absolute beginning, which means deciding on a few very important practical questions, which will happen in the following section.

7.1 CHOICES REGARDING THE PROGRAMMING PROJECT

The most impactful choice of all is which programming language to use for the design of the tool. A number of different options were available, of which only two were ever seriously considered: Python and Java. Both languages have two very important properties in common. The first is that they are both platform independent, meaning that, with minimal effort, the tool could be platform independent as well. The second is that they require little (in most cases no) prior installation. Most computers that will realistically run software like a DEVS editor, will probably already have both Python and Java installed.

Further comparison of the two languages (considering this thesis, not general differences) is discussed in the list below:

- **Python**: Python seems like the obvious choice for the design of a tool that makes use of PyPDEVS[15], a DEVS simulation kernel written (as the name suggests) in Python. The main problem with this language is that I personally have rather limited experience with it in general, and no experience what so ever when it comes to making graphical user interfaces. Should this language therefore be selected, than a lot of time would have to be spent on learning just how to build GUIs in python, time that could otherwise be very well used for the design of the tool itself.

- **Java**: Throughout the course of my studies, Java has been the language in which I have made by far the most projects, including projects with GUIs. This means no time would have to be spent on becoming familiar with the language itself. The main difficulty with Java is that is does not match the language of the simulation kernel that will be used. However, this turned out not to be a real problem anyway, when making use of a compiler. This compiler turns DEVS models from a proprietary format (which will be discussed later) that supports Python code into a Python script that can be input in the kernel. As you will read later on, the choice of using a compiler actually brings with it another major benefit.

Although using Java for the tool and Python for the simulation kernel means that both have to be installed on the user's system, both of them are so generally wide-spread that this did not influence the decision.

Knowing all of the above, I chose to go with the language I am most familiar with, which is Java. Having decided on a programming language, the next step is to choose an API for the creation of the user interface.
Of the commonly used Java UI APIs, I am most familiar with Swing, which is readily available in the standard Java JDK and thus requires no additional dependencies to be linked with the tool. Swing is also capable of adapting its style and detail design to the OS on which the tool is executed. These three things combined (no learning overhead, no extra dependencies and relatively modern design) mean that Swing is an excellent choice for the design of a DEVS tool prototype.

7.2 INTERACTION MODEL

Before the structure of the code itself is described, this section first examines how the highest-level conceptual building blocks interact with each other. This is demonstrated by Figure 1. The complete interaction can be explained as follows:

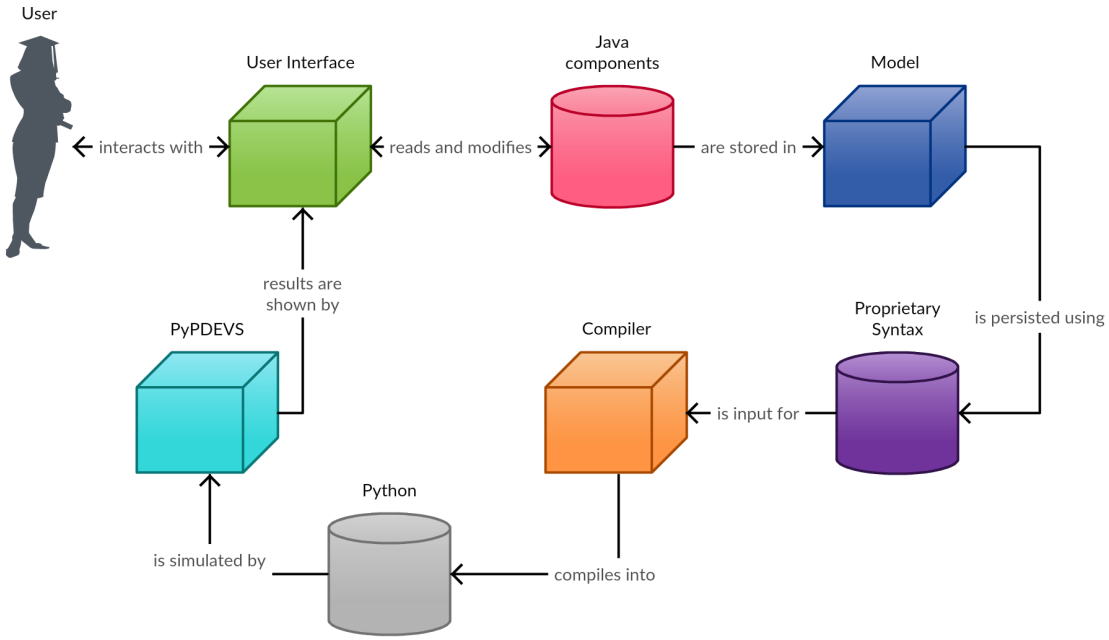

Figure 1: Interaction model of the global structure.

1. The *User* interacts with the *User Interface* (creating/changing/compiling/simulatiing the project).
2. The *User Interface* gets all of its information (and modifies it) from the Java representations of the DEVS components, which are stored as *Java components*.
3. The *Java components* are generated and internally stored by the *Model*.

4. The *Model* saves all DEVS components on the hard drive in the form of the *Proprietary Syntax*.
5. The *Proprietary Syntax* serves as input for the *Compiler*.
6. The *Compiler* turns the *Proprietary Syntax* into *Python* code.
7. The *Python* representation of the project is simulated using *PyPDEVS*.
8. The output from the *PyPDEVS* simulation is sent to the *User Interface*.
9. The *User Interface* shows the simulation output to the *User*.

7.3 INTERNAL STRUCTURE

Of course, the entire project was not built as a single block of Java classes. Doing it that way would imply that the overview would be lost before any important functionality would be successfully implemented. Instead, the tool's main functionalities have been split up in nine different groups, of which each is represented by a Java package, all of which will be introduced here. Each major package gets its own chapter further in this Part of the text.

All the different packages are discussed briefly in the list below:

- **Compiler**: The compiler was created to compile the different models that are described in the proprietary syntax (which will be defined soon) into a Python file that can be used as input for the PyPDEVS simulation kernel.
- **Model**: The model package contains Java representations of the two main DEVS model elements (*Atomic DEVS* and *Coupled DEVS*) as well as a simple *Message* type, which can be used to define somewhat more complex (compared to standard types such as integers, strings or lists, ...) messages between the outputs and inputs of DEVS. Throughout the rest of this text, the collection of these three elements (Atomic and Coupled DEVS and Messages) will be referred to as the DEVS Components.
  Furthermore, it also defines a *Project* and its *Settings*. The project is a collection of DEVS blocks combined to make a more complex fully working model.
  - **Model Persistence**: a sub-package of the Model package. The classes in this package are designed to save the different parts of the DEVS model (all Atomic DEVS, Coupled DEVS and Messages). They are converted from their Java representation (which is used internally within the tool) to their representation in the proprietary syntax, which is human readable (without having any knowledge of Java) and is used for the compilation into PyPDEVS.
  - **State**: The State package serves as the main link between the model (which contains the data on DEVS projects and its elements) and the GUI, which is where the user gets to see and modify this data. Although it technically lies between the Model and the GUI parts of the entire project, it is too small a

package to stand on its own, and, since it is independent of whichever GUI implementation is used, but strongly dependent on the model implementation, classifying it under the Model package seemed appropriate.

  - **Library**: Similarly to the State, the Library package is too small to be classified as one of the main packages in this project. The Library package takes care of importing the library blocks into the currently active DEVS project. Since the only GUI interaction with this package is the selection of which library to import from a file picker, and the actual reading of the library and including it into the project are much more significant and part of the model, this is also where this package is classified.

- **GUI**: The GUI package envelopes everything that is user-interface related. The main GUI package contains the global `main` method that runs the tool, and provides the general lay-out, including the menu bar, model tree and log box.
  - **GUI Editor**: The GUI Editor package provides the user with a visual representation of the internal structure of all three different DEVS components. Any of these components can as such be inspected or modified. It actually serves as an easier and safer method of creating models in the proprietary syntax compared to doing it all manually.
  - **GUI Graph**: The GUI Graph package contains the functionality to visually represent either a Coupled DEVS or Atomic DEVS by showing a visual graph of the different DEVS blocks and their connections, or the internal states and its transitions respectively.
  - **GUI Simulator**: Even though the simulation of DEVS models should technically not be defined in any of the GUI classes, the use of PyPDEVS has made this so trivial, that the main challenge considering simulation is to show the results to the user. As such, it has been classified under the GUI packages.

Figure 2 is a structural image that contains all the packages introduced above and shows a graphical representation of how these are all connected and how they interact with each other. In this figure, any blue elements depict Model packages, an orange element belongs to the Compiler package and green signifies the GUI package. Mixed colors (State and Library) show that a package does not really belong to one or another, but are strongly connected to both. As far as connectors go, solid connectors depict a direct connection between different packages. These direct lines depict an actual internal interaction within the Java code. Dashed lines show that there is an indirect influence of the originating package on the receiving package. This means that the results of one package impact the results of another, but there is no direct connection. For example, the Model Persistence package stores the proprietary syntax files that the Compiler requires for compilation, but never do the two packages actually interact.

Obviously, the details of the structure and Java implementation of this structure are way to complicated to be represented by a simple, high-level image such as Figure 2. In order to further specify the final internal architecture, each individual main package, together with their internal and external connections, will be examined in much more detail in the following chapters. First, the Compiler package will be

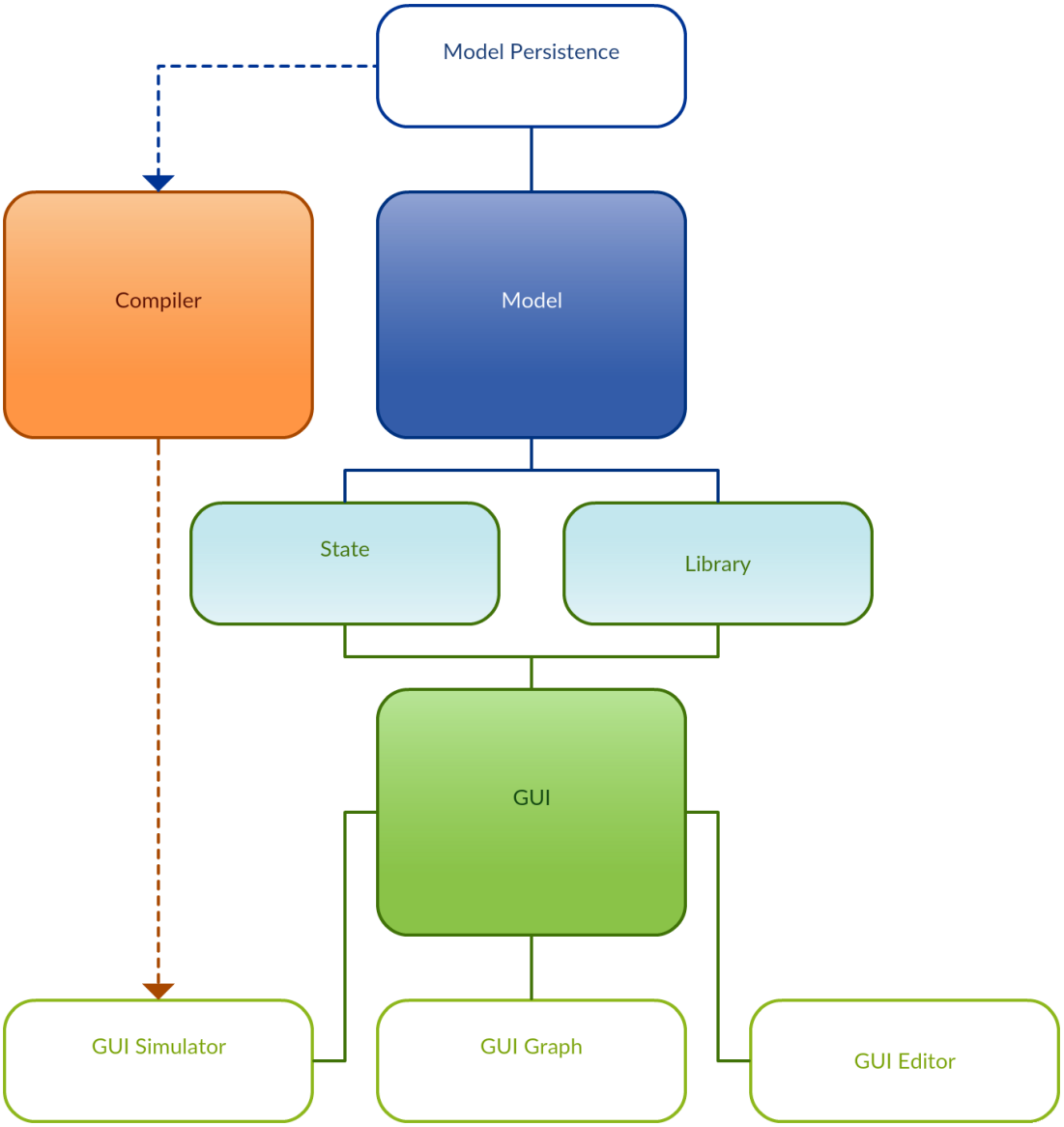

Figure 2: Graphical representation of the package (and thus software) structure.

discussed in Chapter 8, and with it the proprietary syntax. The Model package and all its sub-packages will be discussed in Chapter 9. The final structure element, the GUI package, will be discussed in Chapter 10.1.4. After those three chapters, the remainder of Part ii will go into more detail about the library, more precisely the implementation of the different blocks.

8

THE COMPILER

The first major part of the tool that was implemented was the *Compiler*, which turns DEVS components from the proprietary syntax into a Python file that can be used for the PyPDEVS simulation. It thus serves as a linking layer between PyPDEVS and the rest of the application. We have briefly touched on it before, but the tool is designed to be relatively easily adaptable to other simulation kernels. The only real thing that would have to be redesigned, is the Compiler.

Before the actual compiler can be discussed, I have to first explain what it is that the compiler turns into PyPDEVS input. Therefore, the proprietary syntax that has been mentioned a number of times before, will be examined first.

8.1 PROPRIETARY DEVS SYNTAX

The proprietary DEVS Syntax is a method of defining DEVS components (Atomic DEVS, Coupled DEVS and Messages) that is human readable, easy to understand and which requires minimum overhead to generate. The three different types of DEVS component each have their own syntax, although the general look and feel is maintained on the various types.

It is important to realize that the examples and library blocks created using this proprietary syntax assume that Python is used for the simulation. This will have effect on exactly how parameters in the syntax are defined (since it Python it is not necessary to define a type in order to declare or initialize a variable). Should, in the future, the tool be modified to use (for example) a Java-based simulation kernel, then it would be necessary to modify how this declaration is done. Furthermore, Python is also the language used for any advanced functionality that might be necessary in the model. As a consequence, it is safe to say that a model defined in the proprietary syntax assuming a Python-based simulation kernel should work on any Python-based simulation kernel, but will not work on any kernel that is not Python-based. The GUI editor in the tool itself is not Python specific, which will be explained in more detail in Chapter 10.1.4.

Technically, it is possible to generate DEVS models using just this syntax and a basic text editor, it should still be easier and more efficient than building a model using standard PyPDEVS python code, as a lot of the overhead (class and method headers, imports, links, ...) is automatically added to this syntax during the compilation. Of course, the fastest way to generate models with this syntax, is through using the complete and finished tool itself, which was designed specifically for the purpose of generating DEVS models in this syntax, further reducing the overhead by removing the need for manually typing the headers, etc.

8.1.1 *Atomic DEVS*

The most low-level building block of a DEVS model is also the most technical. Atomic DEVS files that were designed with this specific syntax can be recognized by the ".adevs" file extension

Atomic DEVS contain the following attributes:

- **Name**: The name of the block.
- **State parameters**: The internally stored parameters that can accessed throughout the block internals.
- **Ports**: Input and output ports that allow the block to interact with others.
- **States**: The internal states in which the block can be at any point in time.
- **Time Advance**: The amount of time it takes for a state to transition to another on its own. Each state has its own time advance.
- **Output Function**: Right before an internal transition happens, the block gets an opportunity to send something to its output and thus any connected block(s). The output function decides exactly what is sent to which output port.
- **Internal Transition**: The internal transition is responsible for deciding which will be the new state when the time advance is reached. It can make decisions based on the internal state of the block, and modify the state of the block should this be necessary.
- **External Transition**: The external transition is responsible for deciding which will be the new state in case any of the block's input ports received some data. It can make decisions based on the internal state of the block or depending on what the input is, and on which port this arrived. It can also modify the state of the block should this be necessary.

To keep things simple, the headers, which precede the details of attribute that will be described, have kept the same name in the proprietary syntax as they have in the list above. After this short introduction of what each of the attributes are responsible for, a similar list as the one above will show how these attributes are represented in the syntax. In all the information below, it is assumed that PyPDEVS is used as the simulation kernel. Using a different simulation kernel would mean that the Python code needs to be structured differently, or even that the code should not be Python at all.

- **Name**: The name of the block is defined as follows:

  `[name = x]` (where x is the name of the block)

- **State parameters**: The state parameters of the block are defined as follows:

  ```
  [state parameters]
  ```

x1 = x2 (where x1 is the parameter's name and x2 is its initial value)
y1 = y2
...

This general structure (the header between square brackets on the first line followed by everything belonging to the attribute denoted by the header) will be used in all the other attributes.

At any place further down in the file, these parameter names can be used just like you would use a parameter in any programming language.

Also note that, in any examples you might see, Python is the main language. As such, you would initialize an exemplary numerical parameters as "`counter = 0`". Should the simulation kernel be changed to a, for example, Java-based version, then the numerical parameter would have to be initialized as "`int counter = 0`"

- **Ports**: The ports of the block are defined as follows:

  [ports]
  IN : x (where x is the name of an input port)
  OUT : y (where y is the name of an output port)
  ...

  In the external transition, you can check whether something was received on a certain input port by checking "`'x' in inputs.keys()`", and the actual value can be retrieved as "`inputs['x']`".

  In the output function, you can send something to a particular output port by including it in the dictionary that will be returned:
  "`return {'x' : x}`".

- **States**: The states of the block are defined as follows:

  [states]
  x : starting state (where x is the name of the state)
  y
  ...

  Exactly one state has to be the designated starting state by adding
  " : starting state" as you can see above.

  These states' names will be extensively used in the following four attributes.

- **Time Advance**: The time advance values for all states are defined as follows:

  [time advance]
  x : {

```
...
} (where x is the state to which this time advance applies)
y : {
...
}
...
```

Between the curly braces any Python code can be written to calculate the time advance value. The final statement in this Python code should always be "`return z`", where z is the time advance.

It is important that a time advance has to be specified for every state in the block, even if there will never be an internal transition from that state to another. In that case, the time advance should be an infinite number. Using PyPDEVS, this can be done by writing "`return INFINITY`" between the curly braces.

- **Output Function**: The output function for all states are defined as follows:

```
[output function]
x : {
...
} (where x is the state to which this output function applies)
y : {
...
}
...
```

Similarly to the time advance, between the curly braces any Python code can be written to calculate the values that need to be sent to the outputs. The final statement in this Python code should always be the return of a dictionary "`return {'x' : x_v, 'y' : y_v, ...}`", where x is the output port to which the value x_v will be sent).

Just like with the time advance, it is important that an output function is specified for every state in the block, even if nothing needs to be passed along. In that case, the return of an empty dictionary "`return {}`" should be the final statement.

- **Internal Transition**: The internal transition(s) are defined as follows:

```
[internal transition]
from_state : x (where x is the state the block is currently in)
to_state : y (where y is the state the block will be in after the transition)
condition : {
...
}
```

```
action : {
...
}
from_state : x
to_state : z
condition : {
...
}
action : {
...
}
...
```

Between the curly braces after the `condition` statement, any Python code can be written to decide whether this transition should fire at this time. The internal state of the block can be used for this purpose. The final statement between these braces should be returning either `True` or `False` or a boolean test.

Between the curly braces after the `action` statement, any Python code can be written to alter the state of the model before the transition takes place. The final statement between these braces should be returning the updated state through a dictionary containing all the state parameters: "`return {'x' :` x_v`, 'y' :` y_v`, ...}`", where x and y are state parameters and x_v and y_v are the new values for these parameters.

In contrast to the time advance and output function, there does not have to be a transition for every combination of states, only those transitions that are necessary have to be implemented.

- **External Transition**: The external transition(s) are defined as follows:

```
[external transition]
from_state : x (where x is the state the block is currently in)
to_state : y (where y is the state the block will be in after the transition)
condition : {
...
}
action : {
...
}
from_state : x
to_state : z
condition : {
...
}
action : {
...
```

```
}
...
```

Almost everything here is the same as with internal transitions, only now also the input ports and the values arriving there can be used in the condition and action code segments. How these can be accessed has been explained earlier under the *ports* attribute.

All these attributes combined form a complete Atomic DEVS block. Code Fragment 1 below contains an example of such a complete block, namely one that implements a non-trivial Circular Queue.

```
[name = CircularQueue]


[state parameters]
override = False
rear = -1
front = -1
queue = [None] * 6


[ports]
IN : incoming
IN : get
OUT : outgoing


[states]
receiving : starting state
sending


[time advance]
receiving : {
return INFINITY
}
sending : {
return 0
}


[output function]
receiving : {
return {}
}
sending : {
size = len(queue)
if front == -1 and rear == -1:
  return {}
retval = queue[front]
front = front + 1
if front == size:
  front = 0
if front - 1 == rear:
  front = -1
```

```
  rear = -1
return {'outgoing' : retval}
}


[internal transition]
from_state : sending
to_state : receiving
condition : {
return True
}
action : {
return {'override' : override, 'rear' : rear, 'front' : front, 'queue' : queue}
}


[external transition]
from_state : receiving
to_state : sending
condition : {
if 'get' in inputs.keys():
  return True
return False
}
action : {
return {'override' : override, 'rear' : rear, 'front' : front, 'queue' : queue}
}
from_state : receiving
to_state : receiving
condition : {
if 'incoming' in inputs.keys():
  return True
return False
}
action : {
size = len(queue)
if override == False and ((rear == size-1 and front == 0) or (front == rear + 1)):
  print('queue is full')
  return {'override' : override, 'rear' : rear, 'front' : front, 'queue' : queue}
else:
  if rear == size-1 and front != 0:
    rear = -1
rear = rear + 1
queue[rear] = inputs['incoming']

if override == True and front == rear:
  front = front + 1
  if front == size:
    front = 0

if front == -1:
  front = 0

return {'override' : override, 'rear' : rear, 'front' : front, 'queue' : queue}
}
```

Code 1: Atomic DEVS block that contains a circular queue

8.1.2 *Coupled DEVS*

The higher-level (and furthermore nestable) building block of a DEVS model is the Coupled DEVS block. Coupled DEVS files that were designed with this specific syntax can be recognized by the ".cdevs" file extension

Coupled DEVS contain the following attributes:

- **Name**: The name of the block.
- **Parameters**: Pass-trough parameters of block that this Coupled DEVS block contains.
- **Components**: All the DEVS blocks that are contained in this Coupled DEVS, these could be both Atomic or Coupled DEVS blocks.
- **Priorities**: The order imposed on the blocks that are part of this Coupled DEVS block that decides which block gets to do a transition first, should multiple blocks have scheduled a transition for the same time. This is important since in Classic DEVS (which is used in this tool), transitions cannot fire "simultaneously", and a different order can have a major influence on the simulation result.
- **Ports**: Input and output ports that allow the block to interact with others.
- **Connections**: Connections between the blocks contained in this Coupled DEVS block, as well as connections between the ports of this block and the contained blocks.

To keep things straightforward, just like in the Atomic DEVS syntax, the name of the headers is still the same as the name of the attributes. The following list will again discuss in more detail how the Coupled DEVS syntax was designed.

- **Name**: The name of the block is defined as follows:

  `[name = x]` (where x is the name of the block)

- **Parameters**: The state parameters of the block are defined as follows:

  `[parameters]`
  x_c `:` x_n `=` x_v (where x_c is the component to which the parameter is passed, x_n is the parameter's name and x_v is its value)
  y_c `:` y_n `=` y_v
  `...`

  The parameters defined in a Coupled DEVS are not used within that Coupled DEVS block itself, rather, they are passed along to the specified child of the Coupled DEVS. Parameters are passed on a block-instance-level, which means each instance of a certain type of block can have different values for its parameter. Note that all Atomic DEVS have to define an initial value to their state parameters, so it is not necessary to include every state parameter of every

block here, just the ones where the initial value needs to be different from the one provided in the Atomic DEVS implementation.

- **Components**: The components of the block are defined as follows:

  `[components]`
  x_t `:` x_n (where x_t is the type and x_n is the name of this specific instance of the component.)
  y_t `:` y_n
  `...`

  Since Coupled DEVS can be nested, these components can be either Coupled DEVS or Atomic DEVS blocks.

- **Priorities**: The priorities of the block are defined as follows:

  `[priorities]`
  x_n (where x_n is the name of a component)
  y_n
  ...

  Note that not every component has to be included in these priorities, however, it is advisable to do so, since otherwise it is impossible to know how exactly the simulation will run, since which block gets to do its transition first can have a major impact on the end result of the simulation, and without setting priorities, one cannot know which has fired in which order without checking the entire simulation log.

- **Ports**: The ports of the block are defined as follows:

  `[ports]`
  `IN :` x (where x is the name of an input port)
  `OUT :` y (where y is the name of an output port)
  `...`

  These ports can be used to pass along values that arrive at the current Coupled DEVS block to blocks within this Coupled DEVS block, and pass output from blocks within the current block to any block connected with the current Coupled DEVS block.

- **Connections**: The connections between the blocks within the current DEVS block and potentially its own ports, are defined as follows:

  `[connections]`
  $x_n.x_p$ `->` $y_n.y_p$ (where x_n is the originating block's name (or nothing if the x_p is an input port of the current block) and x_p is the originating block's output port (or one of the current block's input ports) and y_n is the target block's name (or nothing if the y_p is an output port of the current block) and y_p is the target block's input port (or one of the current block's output ports))
  $y_n.y_p$ `->` $z_n.z_p$

```
transfer function {
...
}
```

How these connections can be used is pretty self-explanatory. The only special thing worth discussing is the presence of the transfer function, which are supported by PyPDEVS, and which can be seen above. Information about these can be found in the PyPDEVS documentation[1], but in short, these are functions that modify the value that is passed from the outport to the inport whilst it is underway. The transfer function (max. of one per connection) has to be written immediately below the connection it belongs to. Between the curly brackets, any Python code can be written to modify the value that is being passed (which can be accessed as `event`). The final statement should return the modified value.

Two complete Coupled DEVS blocks written in this syntax have been included below. The first is a Coupled DEVS that does not serve as the root of a model, it has to be included as a component in another block in order to work. It shows how the connections of between the block itself and its components work. This block can be found in Code Fragment 2.

A second Coupled DEVS model does serve as the root for a model. In fact, in was created to test the Atomic DEVS block from Code Fragment 1. This block, which can be found in Code Fragment 3, demonstrates the passing of parameters and the use of a transfer function, which turns a random float number between 0 and 1 to a random int number between 0 and 100.

```
1  [name = Collector]
2
3
4  [parameters]
5
6  [components]
7  SimulationTime : sim
8  CollectorAtomic : col
9
10 [priorities]
11 sim
12 col
13
14
15 [ports]
16 IN : input
17 OUT : passthrough
18 IN : getAll
19 IN : getTimes
20 IN : getItems
21 IN : getCount
22 OUT : count
23 OUT : times
24 OUT : items
25
```

1 http://msdl.cs.mcgill.ca/projects/DEVS/PythonPDEVS/documentation/html/transferfunction.html

```
26
27 [connections]
28 input -> sim.incoming
29 getAll -> col.getAll
30 getTimes -> col.getTimes
31 getItems -> col.getItems
32 getCount -> col.getCount
33 sim.time -> col.time
34 sim.passthrough -> col.incoming
35 col.passthrough -> passthrough
36 col.count -> count
37 col.times -> times
38 col.items -> items
```

Code 2: Coupled DEVS block that contains a Collector

```
1  [name = circularqueuetest]
2
3
4  [parameters]
5  gen : generator_period = 3
6  queue : override = True
7
8  [components]
9  Generator : gen
10 Generate_Random : add
11 CircularQueue : queue
12
13 [priorities]
14 queue
15 add
16 gen
17
18
19 [ports]
20
21
22 [connections]
23 add.generated -> queue.incoming
24 transfer function {
25 return int(event*100)
26 }
27 gen.generated -> queue.get
```

Code 3: Coupled DEVS block that serves as a test for the Circular Queue Atomic DEVS model.

8.1.3 *DEVS Message*

Without having the DEVS Message, only singular values of a given Python type (assuming Python-based simulation) can be sent between inputs and outputs. In order to bring some more of the perks of the object-oriented nature of Python into the tool, the DEVS Message was created. A DEVS Message defines a container that can store any number of values of any type. This way, sending a number of different values

between blocks is made much easier. DEVS Message files can be recognized by the ".devsmessage" extension.

In programming terms, these DEVS Messages allow you to send custom (Python) classes between blocks. These classes are very limited, however, as they only serve as a container or struct, and cannot host any methods other than a constructor.

Atomic DEVS contain the following attributes:

- **Name**: The name of the message.
- **Parameters**: All individual values that can be stored in the message.

Since the syntax is so limited, the entire DEVS Message will be discussed at the same time. The syntax can be described as follows:

`[name = n]` (where n is the name of the message)

`[parameters]`
x_n (where x_n is the name of one of the parameters)
y_n
...

An extensive and useful example of a DEVS Message would be to represent an IPv4 packet. Code Fragment 4 shows exactly that.

```
1  [name = IPv4 Packet]
2
3  [parameters]
4  Version
5  IHL
6  Type_Of_Service
7  Total_Length
8  Identification
9  Flags
10 Fragment_Offset
11 Time_To_Live
12 Protocol
13 Header_Checksum
14 Source_Address
15 Destination_Address
16 Options
17 Padding
```

Code 4: DEVS Message that represents an IPv4 packet.

8.2 FROM PROPRIETARY SYNTAX TO PYPDEVS

Of course, the proprietary syntax that was elaborately discussed above can not be directly input into PyPDEVS, as this simulation kernel expects working Python code that runs without issue and links correctly into PyPDEVS simulation. The compiler will

read all of the components in the proprietary syntax format, and build from that a single Python file with fully working Python representations of each of these components.

The easiest way to apply compilation is by using the complete tool, in which case all the files in the current project will be compiled together. However, the compiler can be executed individually as a command line tool, which takes a location string as input and compiles links together all the ".adevs", ".cdevs" and ".devsmessage" files it can find at that location in a single file.

8.2.1 *Classes of the Compiler package*

The entire part of the tool that provides compilation is comprised in a total of six Java classes. All of these classes can be found in the source code that should be available with this thesis text.

- `PyPDEVS_Compiler`: This class contains the `main` method that allows the compiler to be ran on its own through command line.
- `FileGetter`: This class retrieves all the files ending in either a ".adevs", ".cdevs" or ".devsmessage" extension from a certain location. It also searches any subfolders of the provided location for any of these files. Its `generateFileList` method gives a list of Java File pointers back to the calling instance.
- `ADEVStoPyPDEVSClasses`: This class is the one actually responsible for the conversion of Atomic DEVS models into its Python equivalent. It receives a `File` pointer of a file using the proprietary syntax and reads this file line by line, converting it into Python as it goes over the file. The methods that do these converting actions are very repetitious and can get really complex due to the enormous amount of exceptions and variations. This thesis text will not discuss the algorithms in more detail, as the added value of this is extremely limited in the global scope of this project.
- `CDEVStoPyPDEVSClasses`: This class treats its incoming `File` pointer in a similar way as the `ADEVStoPyPDEVSClasses` class, converting Coupled DEVS from the proprietary syntax into Python.
- `MessageToPyPDEVSClasses`: This class treats its incoming `File` pointer in a similar way as the `ADEVStoPyPDEVSClasses` class, converting DEVS Messages from the proprietary syntax into Python.
- `PyPDEVSCreator`: This class binds all the other classes in this package together, ans is also the only connection between the compilation package and any external requests, coming from either the `PyPDEVS_Compiler` class or from within the model. This class is responsible for receiving (by calling the `create` method) the list of files that need to be compiled into the Python file. It then retrieves the data from those files (by requesting this information from the `FileGetter` class), and passes this along to the three individual compilers (by calling the `treat` method),

depending on the type of the DEVS component. Once all files have been compiled, this class also merges them together into one complete, working Python file with its own `write` method.

A visual representation of how these classes interact with each other is shown by the (very much simplified) UML diagram in Figure 3.

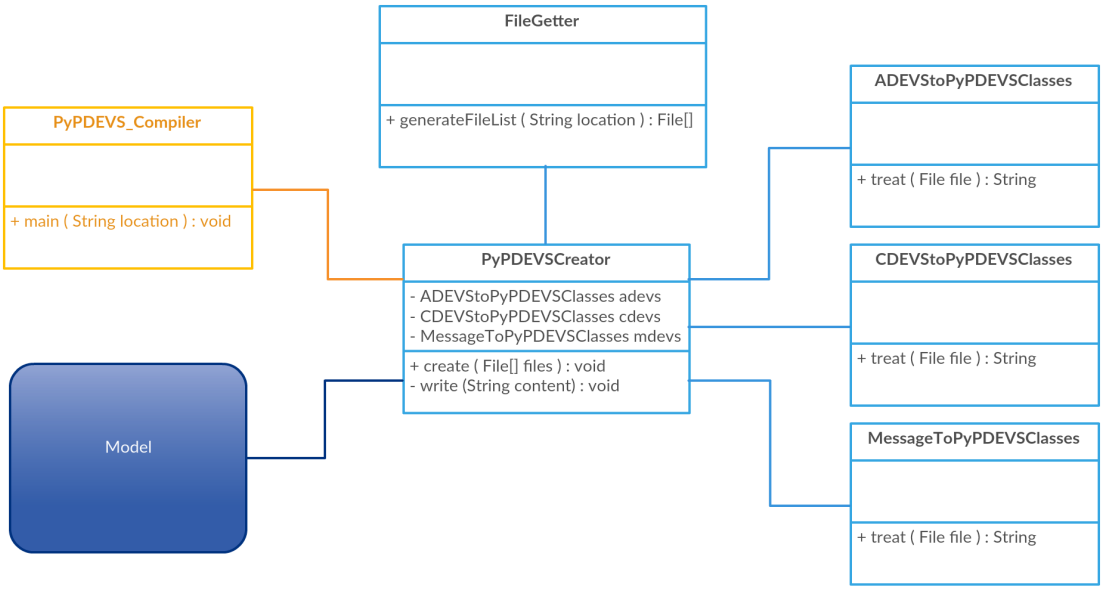

Figure 3: Simplified UML representation of the Compile package classes.

8.2.2 *Compilation result*

For the sake of demonstration, the converted versions of the Atomic DEVS from Code Fragment 1 and the Coupled DEVS from Code Fragment 3, are shown in Code Fragment 5 and 6 respectively. As you can undoubtedly see, the version written in the proprietary syntax is (somewhat) shorter, but, more importantly, provides a much more readable format that should be more convenient to figure out and faster to write.

```
1  class CircularQueueState:
2    def __init__(self, name="", queue=None, front=None, rear=None, override=None):
3      self.name = name
4      self.queue = queue
5      self.front = front
6      self.rear = rear
7      self.override = override
8
9
10   def __str__(self):
11     s = ""
12     s += "state.name = " + str(self.name) + "\n"
13     s += "state.queue = " + str(self.queue) + "\n"
14     s += "state.front = " + str(self.front) + "\n"
15     s += "state.rear = " + str(self.rear) + "\n"
16     s += "state.override = " + str(self.override) + "\n"
17     return s
```

```
18
19 class CircularQueue(AtomicDEVS):
20   def __init__(self, name = "CircularQueue", queue=[None] * 6, front=-1, rear=-1, override=
         False):
21     AtomicDEVS.__init__(self, name)
22
23     self.my_ports = {}
24
25     self.state = CircularQueueState(name="receiving", queue=queue, front=front, rear=rear,
         override=override)
26
27     self.my_ports = {"incoming": self.addInPort("incoming"), "get": self.addInPort("get"),
         "outgoing": self.addOutPort("outgoing")}
28
29   def timeAdvance(self):
30     if self.state.name == "receiving":
31       return INFINITY
32
33     if self.state.name == "sending":
34       return 0
35
36   def outputFnc(self):
37     def subfunc(self):
38       if self.state.name == "receiving":
39         return {}
40
41       if self.state.name == "sending":
42         size = len(self.state.queue)
43         if self.state.front == -1 and self.state.rear == -1:
44           return {}
45         retval = self.state.queue[self.state.front]
46         self.state.front = self.state.front + 1
47         if self.state.front == size:
48           self.state.front = 0
49         if self.state.front - 1 == self.state.rear:
50           self.state.front = -1
51           self.state.rear = -1
52         return {'outgoing' : retval}
53
54     return {self.my_ports[k]: v for k, v in subfunc(self).iteritems()}
55   def intTransition(self):
56     def cond_int_sending_to_receiving():
57       return True
58
59     def action_int_sending_to_receiving():
60       return {'override' : self.state.override, 'rear' : self.state.rear, 'front' : self.
           state.front, 'queue' : self.state.queue}
61
62     if self.state.name == "sending" and cond_int_sending_to_receiving():
63       return CircularQueueState(name="receiving", **action_int_sending_to_receiving())
64
65     else:
66       return AtomicDEVS.intTransition(self)
67   def extTransition(self, my_inputs):
68     inputs = {k.getPortName(): v for k, v in my_inputs.iteritems()}
69     def cond_ext_receiving_to_sending():
70       if 'get' in inputs.keys():
```

```
71          return True
72        return False
73
74      def action_ext_receiving_to_sending():
75        return {'override' : self.state.override, 'rear' : self.state.rear, 'front' : self.
              state.front, 'queue' : self.state.queue}
76
77      def cond_ext_receiving_to_receiving():
78        if 'incoming' in inputs.keys():
79          return True
80        return False
81
82      def action_ext_receiving_to_receiving():
83        size = len(self.state.queue)
84        if self.state.override == False and ((self.state.rear == size-1 and self.state.front
              == 0) or (self.state.front == self.state.rear + 1)):
85          print('queue is full')
86          return {'override' : self.state.override, 'rear' : self.state.rear, 'front' : self.
                state.front, 'queue' : self.state.queue}
87        else:
88          if self.state.rear == size-1 and self.state.front != 0:
89            self.state.rear = -1
90        self.state.rear = self.state.rear + 1
91        self.state.queue[self.state.rear] = inputs['incoming']
92        if self.state.override == True and self.state.front == self.state.rear:
93          self.state.front = self.state.front + 1
94          if self.state.front == size:
95            self.state.front = 0
96        if self.state.front == -1:
97          self.state.front = 0
98        return {'override' : self.state.override, 'rear' : self.state.rear, 'front' : self.
              state.front, 'queue' : self.state.queue}
99
100     if self.state.name == "receiving" and cond_ext_receiving_to_sending():
101       return CircularQueueState(name="sending", **action_ext_receiving_to_sending())
102
103
104     if self.state.name == "receiving" and cond_ext_receiving_to_receiving():
105       return CircularQueueState(name="receiving", **action_ext_receiving_to_receiving())
106
107
108     else:
109       return AtomicDEVS.extTransition(self, my_inputs)
```

Code 5: Circular Queue Atomic DEVS model converted into Python.

The attributes of the proprietary syntax of the Atomic DEVS model can be recognized in the Python version (Code Fragment 5):

- **Name**: The name can be found on line 20, in the constructor.
- **State Parameters**: The state parameters are defined in the State class (lines 1 to 17) and initialized in the constructor (line 20 and 25).
- **Ports**: The ports are defined in the constructor, on line 27. In the external transition, they are referenced in the same way as in the proprietary syntax (for example on lines 78 and 91).

- **States**: The states are never formally defined in the Python code. The initial state is set in the State class constructor (line 25), and only the current state is stored here. In the rest of the code the states are referenced to as being a string value (for example on lines 30 and 62, and in methods like those on lines 56 and 59).
- **Time Advance**: The time advance for all states are implemented in the `timeAdvance` method. The state for a certain time advance is decided on through a test (such as on line 30) and the Python codes that sits between the curly brackets in the proprietary syntax is the code beneath this test.
- **Output Function**: The output function is very similar to the time advance, but can be found in the `outputFnc` method.
- **Internal Transition**: All internal transitions can be found in the `intTransition` method. Every internal transition is represented by three main parts: the trivial two lines that call the other two parts (lines 62 to 63), the condition (lines 56 to 57) and the action (lines 59 to 60). Both the from and to states are embedded in the method names and the to state is passed on the final statement (line 63).
- **External Transition**: The external transitions are represented very similar to the internal transitions, and can be found in the `extTransition` method.

A source of inspiration for the general look and layout of the Python result (for both the Atomic and the Coupled DEVS), easily recognizable in elements such as the `def cond_ext_receiving_to_receiving()` or the `CircularQueueState` class, was the result of the DEVS compiler used in the AToMPM DEVS formalism[12]. I had previous experience with this compiler since I had to use it for a different project last year, and found that the result is a really well structured and readable Python conversion. I thus cannot take credit for this layout. Having a clear layout for even the compiled Python files is important since model debugging will have to happen from within these compiled files, as long as a debugger is not implemented in the tool (which it will not be in the scope of this thesis).

```
1  class circularqueuetest(CoupledDEVS):
2    def __init__(self, name = "circularqueuetest", gen_generator_period = 3, queue_override =
         True):
3      CoupledDEVS.__init__(self, name)
4
5      self.my_ports = {}
6      self.submodels = {}
7
8      self.my_ports = {}
9
10     self.submodels["gen"] = self.addSubModel(Generator(name = "gen", generator_period =
           gen_generator_period))
11     self.submodels["add"] = self.addSubModel(Generate_Random(name = "add"))
12     self.submodels["queue"] = self.addSubModel(CircularQueue(name = "queue", override =
           queue_override))
13
14     def transfer_add_generated_to_queue_incoming(event):
15       return int(event*100)
16
```

```
17      self.connectPorts(self.submodels["add"].my_ports["generated"], self.submodels["queue"].
            my_ports["incoming"], transfer_add_generated_to_queue_incoming)
18      self.connectPorts(self.submodels["gen"].my_ports["generated"], self.submodels["queue"].
            my_ports["get"])
19
20    def select(self, imm):
21      for i, val in enumerate(imm):
22        if isinstance(val, AtomicDEVS) or isinstance(val, CoupledDEVS):
23          if val.getModelName() == "queue":
24            return val
25      for i, val in enumerate(imm):
26        if isinstance(val, AtomicDEVS) or isinstance(val, CoupledDEVS):
27          if val.getModelName() == "add":
28            return val
29      for i, val in enumerate(imm):
30        if isinstance(val, AtomicDEVS) or isinstance(val, CoupledDEVS):
31          if val.getModelName() == "gen":
32            return val
33      return imm[0]
```

Code 6: Circular Queue test Coupled DEVS model converted into Python.

The attributes of the proprietary syntax of the Coupled DEVS model can be recognized in the Python version (Code Fragment 6).

- **Name**: The name can be found on line 2, in the constructor.
- **Parameters**: The parameters can also be found in the constructor on line 2, a prefix has been added to the names of the parameters such that a distinction can be made between parameters for the multiple instances of the same type of block.
- **Components**: The components are added in the constructor (lines 10 to 12).
- **Priorities**: The priorities are used to implement the `select` method. It tries to find the block with the highest priority first, if that is not in the list of blocks that request a transition at this time, the next highest priority is tested, and so on.
- **Ports**: Ports are added in the constructor (line 8, however, this Coupled DEVS model in particular does not have any ports.
- **Connections**: Connections are also defined in the constructor, which you can see on lines 17 and 18. Potential transfer function are defined before the connections, and they are referenced within the connection itself (a transfer function can be found on lines 14 and 15, and on line 17, you can see that the method is passed as a parameter to the `connectPorts` method.

The compilation result of any DEVS Message file is just a simple Python class containing the parameters that were defined in the proprietary syntax and a `__str__` that generates a string representation of the class. Code Fragment 7 shows the Python version of the IPv4 message introduced in Code Fragment 4, and will not be discussed further regarding its simplicity.

```
1  class IPv4_Packet:
2    def __init__(self, Version=None, IHL = None, Type_Of_Service = None, Total_Length = None,
         ...):
```

```
3       self.Version = Version
4       self.IHL = IHL
5       self.Type_Of_Service = Type_Of_Service
6       self.Total_Length = Total_Length
7       ...
8
9
10    def __str__(self):
11      s = ""
12      s += "state.Version = " + str(self.Version) + "\n"
13      s += "state.IHL = " + str(self.IHL) + "\n"
14      s += "state.Type_Of_Service = " + str(self.Type_Of_Service) + "\n"
15      s += "state.Total_Length = " + str(self.Total_Length) + "\n"
16      ...
17      return s
```

Code 7: IPv4 DEVS Message (partially) converted into Python.

8.2.3 *End result*

The end result of the entire compilation process is the combination of all these individual compiled DEVS components (Atomic and Coupled blocks and Messages) into a single Python file, that can be used for DEVS simulation using PyPDEVS. This simulation can be done by manually creating a simulation file and running this file using the PyPDEVS command line interface, or by using the build-in "simulate" button from the complete tool, which automatically generates the simulation file and runs the simulation internally, showing the simulation output on screen.

9

THE MODEL

The main purpose of the Model package is to internally maintain the DEVS project that is currently open in the tool. Next to loading and keeping the state of the project and all of its DEVS components, which is done by the main Model package, it also provides functionality for saving the model into the proprietary syntax (in the Model Persistence package).

Two other ("hybrid") packages were also classified to fall under the bigger Model package, even though they are not technically in the package. Their functionality sits between the Model and GUI packages. The first is the State package, which is an interface between the Model and the GUI, through which all interaction (function calls and return values) passes. This was implemented to minimize the coupling between both of these separate packages. The second of these hybrid packages is the Library package, which opens the library file (which is in its current form a glorified zip folder) and extracts all the DEVS components in this file into the model. This package is classified as hybrid since its most important method is called exclusively from the GUI, but it does modify the model. Another reason why this package (that at this point only contains one class) is on its own, is to allow and encourage further development on the library aspect of the tool, since this is undoubtedly one of, if not, the most important aspect.

9.1 CLASSES OF THE MODEL AND MODEL PERSISTENCE PACKAGES

In this section all classes belonging to either the Model or the Model Persistence packages will be discussed. This discussion will be rather brief since most classes serve mainly to represent a specific type of DEVS component, or were created to save the DEVS components into the proprietary syntax.

The following classes belong to the Model package:

- `Component`: This class is a parent class to both the `Message` class and the `DEVS` class, created such that all instances of all three different DEVS components can stored in a single data structure, such as a list or array. It also extracts values that all three types of components need to have, which are the name and the unique ID of the component.
- `DEVS`: This class inherits from the `Component` class and is parent to the `AtomicDEVS` and `CoupledDEVS` classes. It was created such that all instances of both Atomic and Coupled DEVS (but not Messages) could be stored in a single data structure.
- `AtomicDEVS`: An instance of this class represents a single Atomic DEVS block and stores all the attributes in Java types, such that they can be used, shown and

modified throughout the Java tool, before being saved in their proprietary syntax form again (by the `AtomicDEVSPersistence` class).

- `CoupledDEVS`: An instance of this class represents a single Coupled DEVS block and stores all the attributes in Java types, such that they can be used, shown and modified throughout the Java tool, before being saved in their proprietary syntax form again (by the `CoupledDEVSPersistence` class).
- `Message`: An instance of this class represents a single DEVS Message and stores all the attributes in Java types, such that they can be used, shown and modified throughout the Java tool, before being saved in their proprietary syntax form again (by the `MessagePersistence` class).
- `Settings`: The settings class contains all the information about the currently active DEVS project. It stores all the components and contains the necessary methods to create new ones. It is also responsible for the persistence of the DEVS project's settings file (which can be recognized by the ".devssettings" extension), by providing the functionality to save and load these types of files.
- `Project`: This class initializes the current DEVS project and provides the `Settings` class with all the necessary information to load the project.

The following classes belong to the Model Persistence package:

- `AtomicDEVSPersistence`: This class translates the Atomic DEVS from its Java representation (that is used within the tool) into the proprietary syntax and saves it.
- `CoupledDEVSPersistence`: This class acts similarly to the `AtomicDEVSPersistence` class, but does this for the Coupled DEVS.
- `MessagePersistence`: This class acts similarly to the `AtomicDEVSPersistence` class, but does this for the DEVS Message.

9.1.1 *The ".devssettings" file structure*

A DEVS project's settings are stored in a file with the ".devssettings" extension. Code Fragment 8 shows an example of one of these files, namely the project that contains the "Circular Queue test" Coupled DEVS model that has been introduced earlier.

```
1  [name = CircularQueueTest]
2
3  [DEVS components]
4  ...\files\Queues\Circular Queue Test/circularqueuetest.cdevs
5
6  [compilation output]
7  ...\files\Queues\Circular Queue Test\output/output.py
8
9  [simulation settings]
10 ...\files\Queues\Circular Queue Test\output/CircularQueueTest_experiment.py
11 [library location]
12 ...\files\Queues\Circular Queue\CircularQueueLib.devslib
```

```
...\files\generator_random_files\Generator_RandomLibrary.devslib
...\files\generator_files\GeneratorLib.devslib
```

Code 8: IPv4 DEVS Message (partially) converted into Python.

The URLs in the fragment above have been shortened to improve readability. Both absolute and relative paths are supported. The *devssettings* file uses a similar general look as the proprietary syntax, by first defining the attribute keyword in square brackets and the attributes data (except for the name) below it. The different attributes are explained in the list below:

- **[name = ...]**: The name of the project.
- **[DEVS components]**: Links to all the components that are part of this project (excluding library components).
- **[compilation output]**: Link to the location where the compiled (from the proprietary syntax in to Python) file will be stored.
- **[simulation settings]**: Link to the location where the simulation settings file (Python) will be stored, this file references the compilation output file and is the Python file that starts the PyPDEVS simulation.
- **[library location]**: Link to all library files that have to be included in the project.

9.2 CLASSES OF THE LIBRARY AND STATE PACKAGE

Both of these packages only have one class each, in the Library package, this is the `LibraryGetter` class, in the State package, it is the `StateKeeper` class. Their functionality is the following:

- `LibraryGetter`: This class read the library file (".libinfo" extension) and creates Java components for each of the blocks in the library, so that they can be added to the project and compiled together with the project's own components.
- `StateKeeper`: This class is the link between the Model and the GUI packages, through which all the communications go. It has no real use itself, except for simplifying (and controlling the coupling) the connection between the GUI and Model packages, such that changes are made as easy as possible, and potential other GUIs can be linked with the existing Model easily.

9.3 COMPLETE SIMPLIFIED UML

All the classes of the four packages discussed in this chapter are visually shown with their most important variables and methods in Figure 4. It shows how all the classes in these packages interact and inherit from on another, as well as how they are connected with one other main package: the GUI.

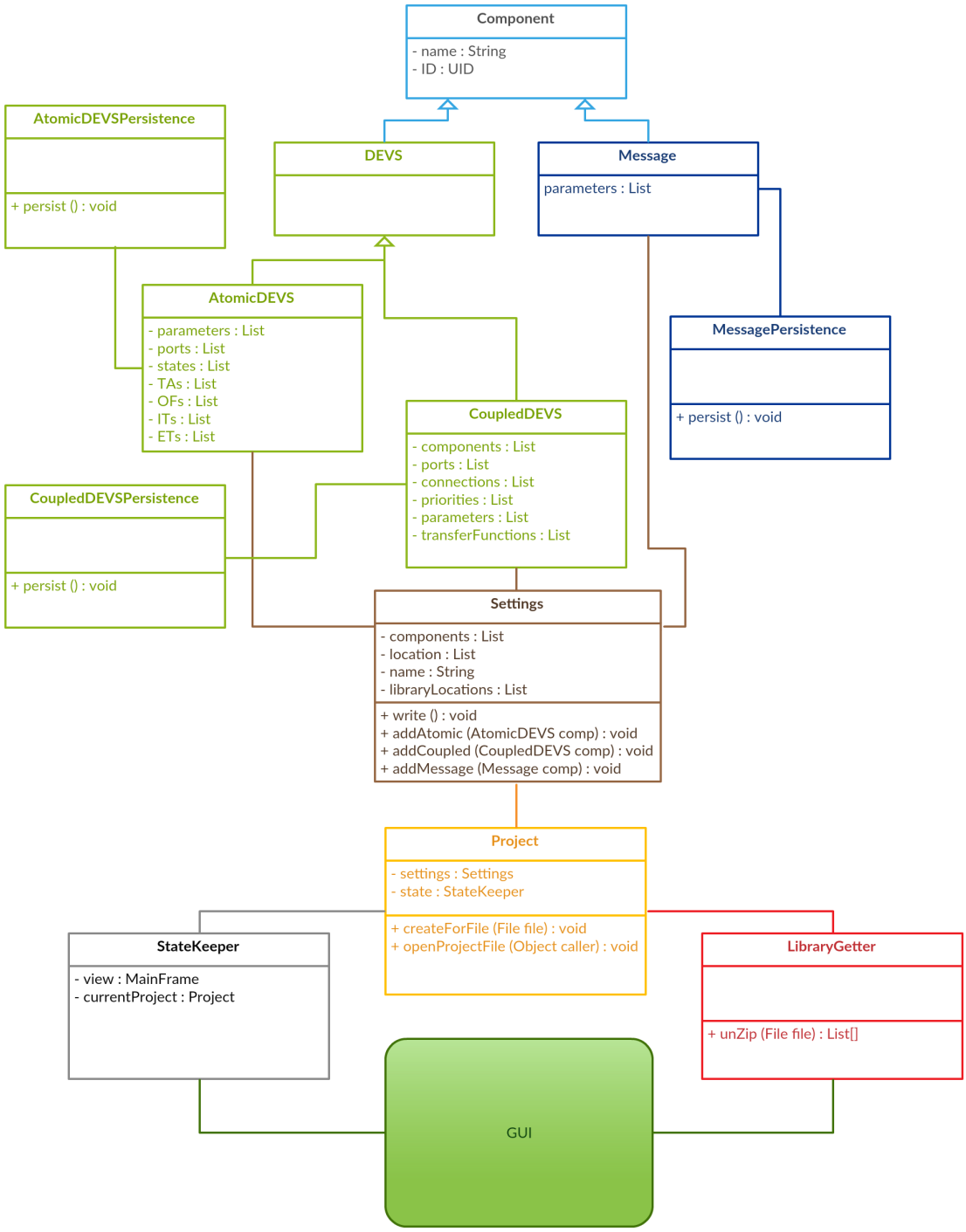

Figure 4: Simplified UML representation of the Model, Model Persistence, Library and State package classes.

10 THE GUI

The final major package of the software that describes the complete tool, is the GUI package. This package contains another three daughter packages: the GUI Editor package, the GUI Graph package and the GUI Simulator package. These four packages combined are responsible for all the visual aspects of the tool, and were designed to increase the efficiency with which the DEVS components can be designed. The GUI Editor package does this by reducing the overhead (that exists when manually writing files in the proprietary syntax or PyPDEVS code) and limit the chances of structural errors in the proprietary syntax or compiled files, by allowing the user to only modify the internal specifics of the components. The GUI Graph package allows users to inspect a visual representation of both Atomic and Coupled DEVS models, which can, in many cases, create a better overview of the internal structure than plain text. The GUI Simulator package allows the output of the simulation kernel (in this case PyPDEVS) to be shown to the user from within the tool.

This chapter will discuss all the classes in these packages similar to all the previous packages, but if the class is responsible for a visual aspect of the tool, this will be demonstrated and reviewed briefly.

10.1 CLASSES OF ALL GUI PACKAGES

The general structure of the discussion of all the classes in the GUI packages, is to first provide a short textual description (like was done in the Compiler and Model packages), but also show the visuals of the tool they are responsible for. The packages will be discusses in order of importance to the tool, and as such the main GUI package will be discussed first.

10.1.1 *Main GUI package*

The main GUI package is responsible for running the program and providing a visual framework in which all other visual aspects will be shown. It contains a total of three classes, for each of the three visual elements of the tool that are permanently shown.

- `MainFrame`: This class is responsible for starting the tool and showing the general frame of the tool (depicted by the *red* box, labeled #1 in Figure 5). It thus contains the tool's general `main` method (next to the command line `main` method for the compiler), and initializes the main frame in which all other visual aspects (apart from pop-up windows) get their place. It also provides the navigation most other functionalities through the menu bar:
  - **File**: allows the user to open an existing project or to create a new one, and to create new components in the currently open project, or load existing

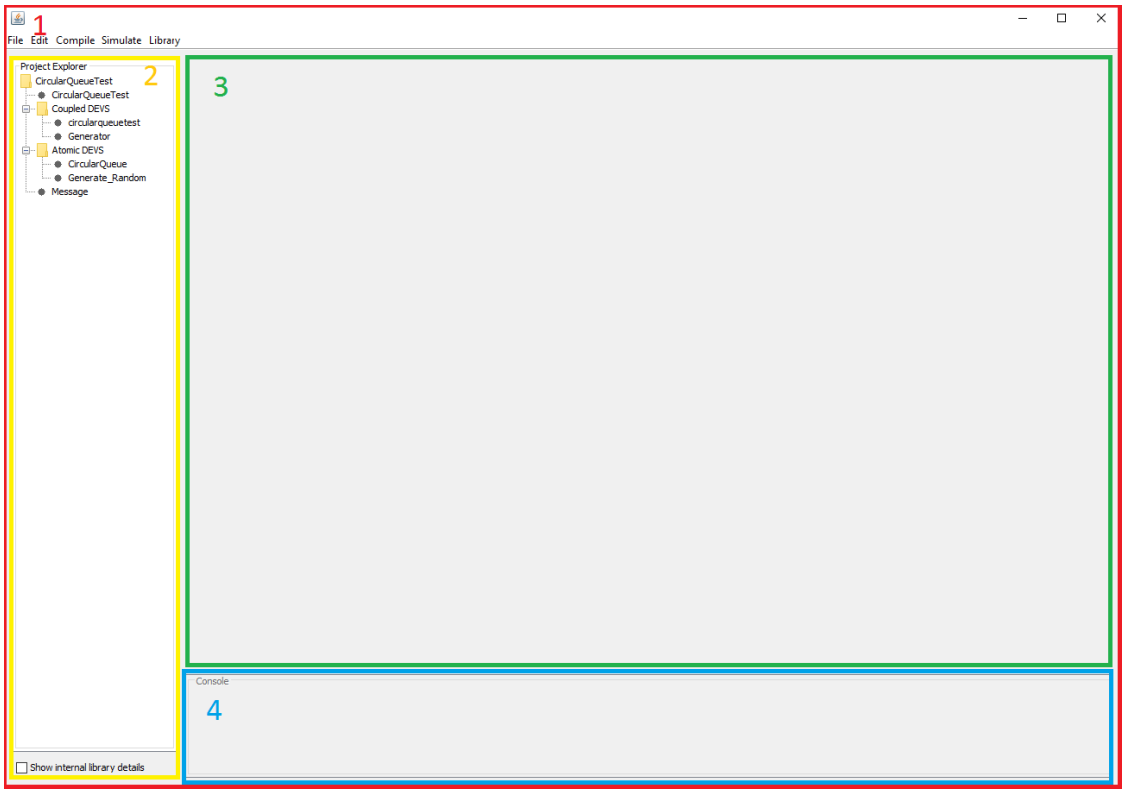

Figure 5: The look of the complete tool right after loading a project.

components into it. It opens a pop-up windows to do either one of those things (an example is shown in Figure 6).

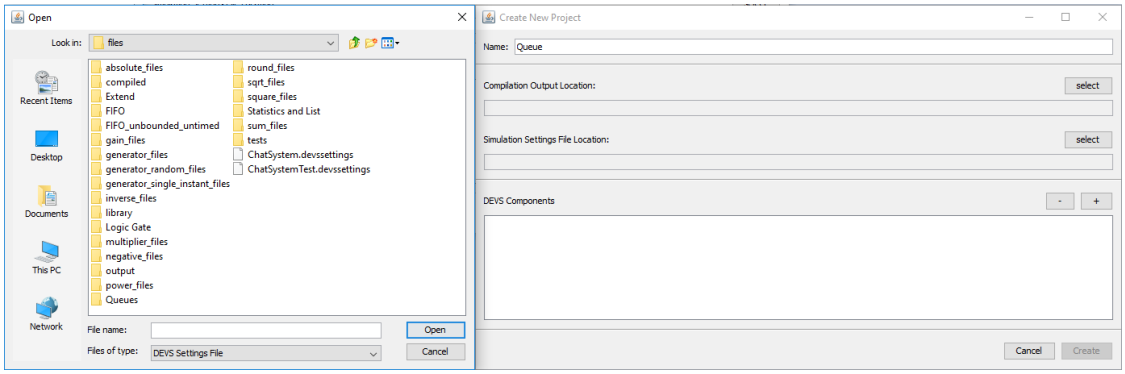

Figure 6: New project dialog.

– **Edit**: allows the user to clear the console.
– **Compile**: allows the user to compile the currently open project into Python, making it ready for simulation.
– **Simulate**: allows the user to run the simulation. The user is asked to select which Coupled DEVS model to simulate and how long (until which simulation time) the simulation should run. After this the parameters of the Coupled DEVS model can be modified, if default values (which are shown

by default) have to be changed. Figure 7 shows the dialogs associated with these actions.

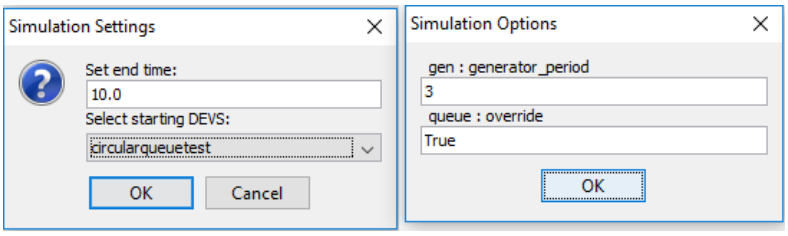

Figure 7: Simulation setting dialogs.

  - **Library**: allows the user to import DEVS library blocks into the model.

- `FileSelector`: This class generates the project tree (in the *yellow* box, labeled #2 in Figure 5). This tree is used for navigating the current project. By double-clicking a component the detail on this will be opened (where the *green* box, labeled #3 is located in Figure 5).

- `Log`: This class takes care of the logging console on the bottom of the UI (depicted in a *blue* box, labeled #4 in Figure 5). The console updates the user on successful or failed saves, compilations, or system errors.

10.1.2 *GUI Editor package*

The GUI Editor package contains all the visual parts of the tool that are made to show information and internals of DEVS projects and components to the user, and to let the user modify them.

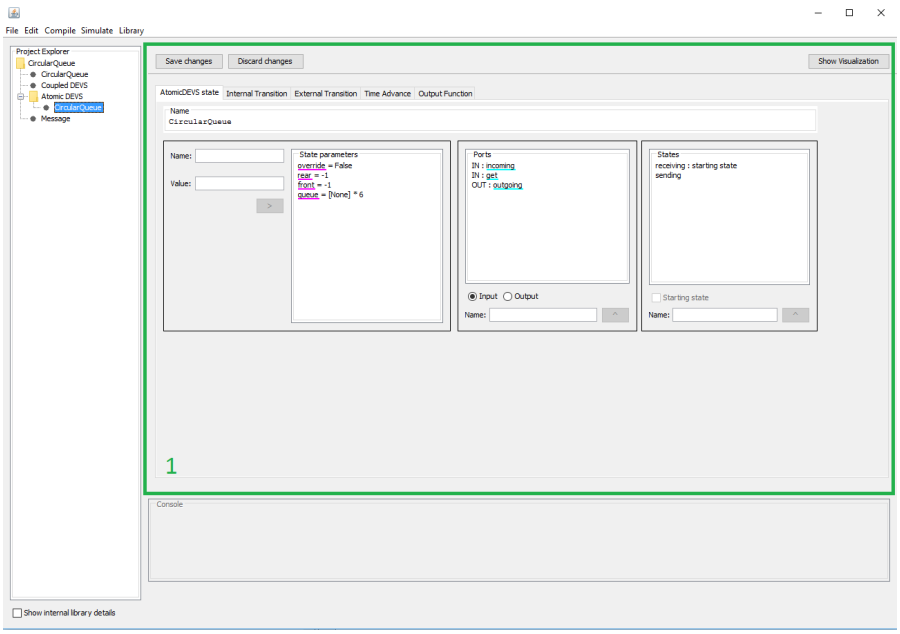

Figure 8: Atomic DEVS view.

- `AtomicDEVSPanel`: Modifying an Atomic DEVS block is done by the `AtomicDEVSPanel` class. The class retrieves the information on the selected Atomic DEVS from the model (which in turn loads in from the proprietary syntax) and shows the information to the user, who can modify it and save it.

  This view is shown by Figure 8, which shows the view for modifying the Atomic DEVS in the *green* box, labeled #1, situated within the whole tool. All attributes not currently visible can be found in the logically labeled tabs.

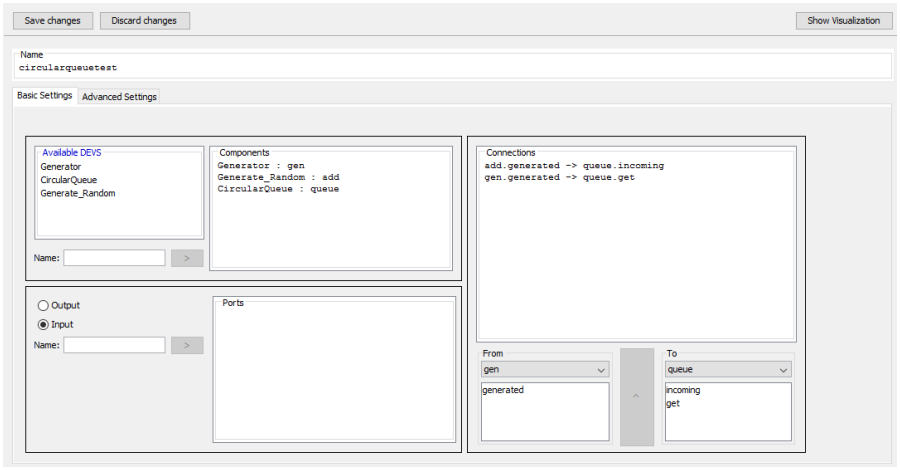

Figure 9: Coupled DEVS view.

- `CoupledDEVSPanel`: This class provides the ability to view and modify a Coupled DEVS block. Like the `AtomicDEVSPanel`, it retrieves information from the model and visually shows it to the user, who can modify and save it. In Figure 9, this view is shown. This time only the specific view (located in the *green* box, labeled #3 in Figure 5 is shown, not the entire tool. Priorities, parameters and transfer functions can be found in the "Advanced Settings" tab.

Figure 10: DEVS Message view.

- `MessagePanel`: Similar to the `AtomicDEVSPanel` and `CoupledDEVSPanel`, this class represents the DEVS Message. An example is shown in Figure 10.

- `AbstractComponentView`: A simple abstract class created such that the `save()` method of all components in the project can be called with one simple operation.

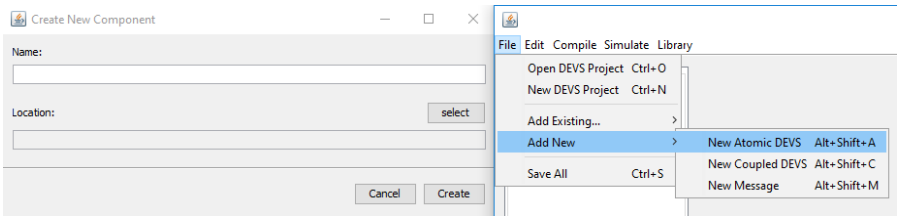

Figure 11: Creating a new DEVS component.

- `NewComponent`: A simple window (inspired by similar tools) to add a new component to the currently open DEVS project. As seen in Figure 11, it requests a name and the location where it will be created. After clicking "create", a new DEVS component (which type is decided by which option was clicked in the menu) is created and added to the model.

- `NewProject`: This class allows user to create a new DEVS project. The resulting window(s) are shown in Figure 6. First the user is asked to select the location of the new project and give it a name (left side of the picture), after which it can further specify where the compiled and simulation settings files have to go, and to possibly link existing DEVS components to it.

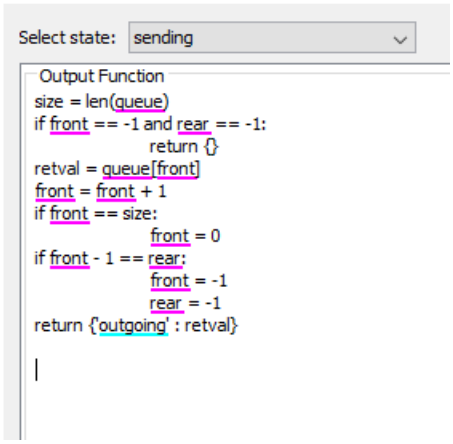

Figure 12: Syntax underlining.

- `HighlightSyntax`: This was an existing class[1], that was modified to support underlining the parameters and port names of an Atomic DEVS model whenever they are used in the Python code (in time advance, output function (Figure 12) and internal and external transitions).

1 source: http://www.java2s.com/Code/Java/Swing-JFC/JTextPaneHighlightExample.htm

10.1.3 *GUI Simulator package*

The GUI simulator package allows the Python simulation (which is ran internally by from the `MainFrame` class) to display its output in the tool. The package contains two classes, both of which were sourced online. The `LimitLinesDocumentListener`[2] and

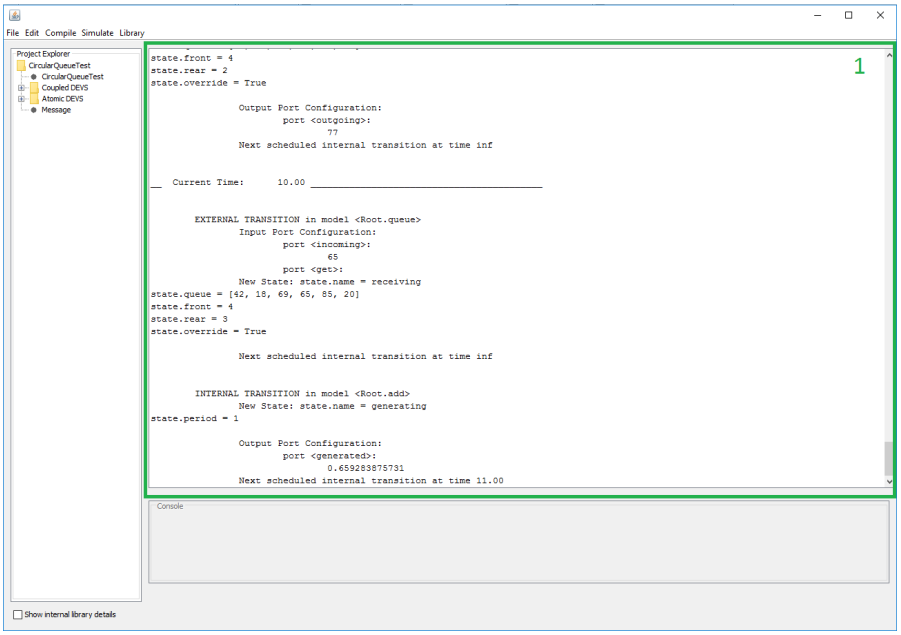

Figure 13: Simulation output.

`MessageConsole`[3] are used to efficiently display fast-moving text on a Swing panel, as shown in Figure 13 in the *green* box, labeled #1. This is shows whenever the user requests to simulate the project.

10.1.4 *GUI Graph package*

The final package that will be discussed as part of the GUI is the GUI graph package. This package utilizes the *JGraphX* library[4] to visually represent both the internal components with their connection of a Coupled DEVS component and the states with their transitions of an Atomic DEVS component. The two classes in the package, `CoupledDEVSGraph` and `AtomicDEVSGraph` pass along the information on the Coupled DEVS or Atomic DEVS to the JGraphX library respectively. Examples of both of these representations can be seen in Figures 14 and 15.

In order to create and show these graphs, the button at the top right corner of both the `AtomicDEVSPane` and `CoupledDEVSPane` (as shown in Figures 8 and 9) can be pressed.

2 source: https://tips4java.wordpress.com/2008/10/15/limit-lines-in-document/
3 source: https://tips4java.wordpress.com/2008/11/08/message-console/
4 source: https://github.com/jgraph/jgraphx

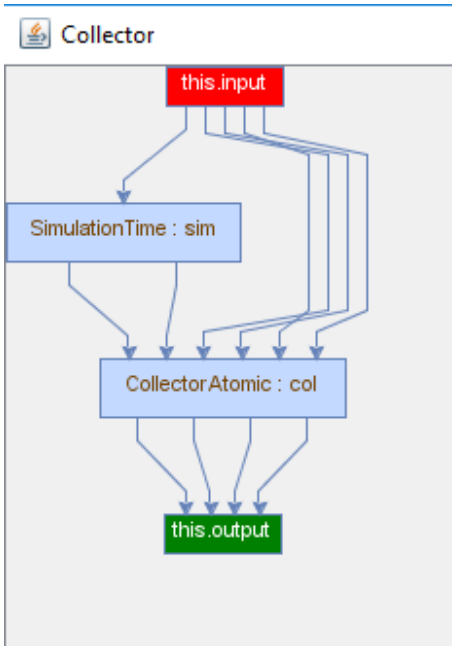

Figure 14: Components and their connections within a Coupled DEVS.

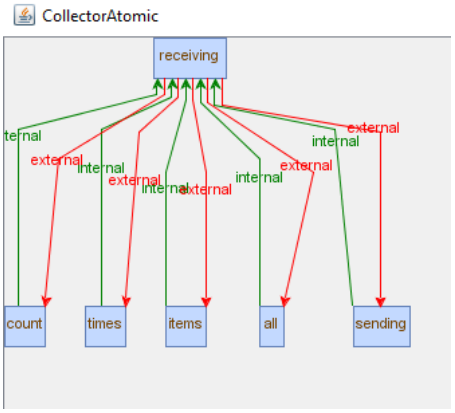

Figure 15: States and their transitions within an Atomic DEVS.

11

LIBRARY IMPLEMENTATION

The use of a library for the generation of DEVS models has already been discussed in detail in Chhapters 5 and 6, as well as in the earlier research[4]. This chapter thus focuses on the practical side of the library, namely defining the structure of a DEVS library model, making sure the complete tool supports libraries, and implementing the actual library models.

The technical implementation of the library was, compared to the implementation of other parts of the software, not too complicated. The first step in creating library support was to specify how exactly a DEVS library would be represented. This is discussed in Section 11.1. Section 11.2 talks briefly about how this has been implemented in the complete tool. The last, and biggest, section in this chapter is 11.3. Here it is described how the tool has been used to create library blocks. It includes a list of which blocks of those originally introduced in Part i have been implemented.

11.1 REPRESENTING A DEVS LIBRARY MODEL

In order to make DEVS libraries easy to use and to share (an important aspect of libraries is that being able to benefit from work that others have done before), there are a number of factors that need to be taken into consideration:

- A single DEVS library model should be kept in a single file. This way, there is no chance of parts of the library going missing when being distributed or moved.
- If a user wants to use a DEVS library model, it should not be required of him to inspect the internals of the DEVS components to know exactly what the block does, which values go to which inputs and which outputs return what result. There should thus be a way for the creator of the library to provide information on these things.
- A library will in many cases contain more than one DEVS component. It should thus be made clear which component is the Root. This is the component that receives input from, and sends output to, the model in which the library is included.

All this information lead to the representation shown in Figure 16: The outermost structure represents the library file itself. It is a file with the ".devslib" extension. Technically, this is simply a zip file, of which the extension has been changed to avoid confusion when retrieving libraries or importing them into the tool. By using a zip file, all other files that are necessary for the library model can be stored inside it, thus fulfilling the first of the three important factors discussed above.

The yellow page-like object at the top of the library DEVS model is the file that

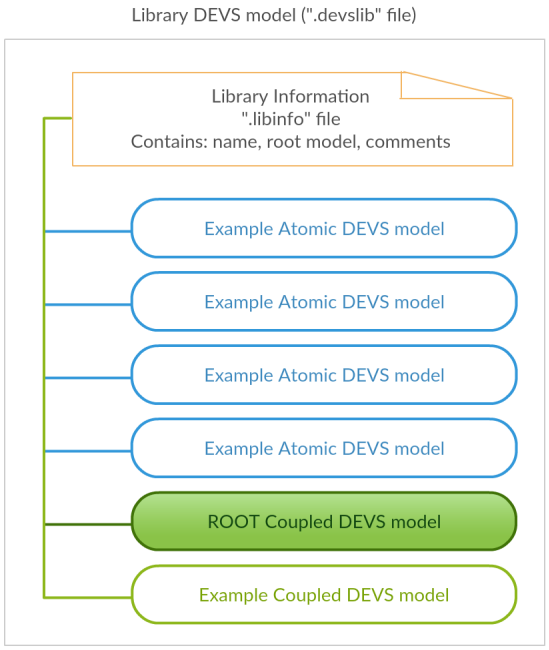

Figure 16: Visual representation of a DEVS library model.

contains all the information about the library. It is a text file with the ".libinfo" extension. This file contains data such as the name of the library, but also allows the creator to document any comments or information about the library and spread it to its users that way. This satisfies the second of the three important factors. Furthermore, this file also specifies which of the DEVS components in the zip (".devslib") structure is the Root, this information can be read by the tool (or user). This fulfills the thir and final factor.

11.2 ADDING LIBRARY SUPPORT TO THE TOOL

We have already gotten a glimpse at the presence of library support in the previous Chapter. Here, I will discuss in more detail the changes and additions to two of the major parts of the software: the Model and the GUI. The third major part, the Compiler, will not be discussed further as this was not influenced at all by the addition of library support.

11.2.1 *Changes to the Model*

The only changes to the Model package were in the `Settings` and `Component` classes. `Settings` is the class that contains the Java implementation of all the Atomic DEVS blocks, Coupled DEVS blocks and DEVS Messages that are used in the currently open project. The settings file (".devssettings" extension) has been extended with the `[library location]` header so that it can store which libraries have been imported into the projects. This is necessary to allow the tool to automatically load the components from this library when the project is loaded. Within the class itself, this change meant the introduction of a new list that stores these locations internally, and a new

method (`readFromZipInputStream`), which reads individual files that originate from the library zip file (".devslib") and prepares them to be made into components.

The `Component` class underwent a very minor change, namely the addition of the `library` field: an integer value which stores, for each DEVS component, whether it is either not part of a library, part of a library, but not root, or the root of a library. The importance for this field will become clear when the GUI is discussed.

The Library Package with its `LibraryGetter` class, is not technically part of the Model package, but was previously classified under it, which is why it will be discussed here as well. This package was (as the name suggests) created solely for the purpose of supporting libraries. The `LibraryGetter`'s methods `getComponents` and `unZip` were designed to read the contents of the library zip file. The result of this is a list of files, each individual file is sent to the `readFromZipInputStream` method (described above) where either a DEVS component is extracted from the file, or information about the library itself (e.g. what the Root component is). The `LibraryGetter` class is accessed by both the GUI (when a new library is imported), or the Model (when a project containing libraries is loaded), which is why it was previously listed as a hybrid class.

11.2.2 *Changes to the GUI*

Most functional changes to the GUI have been implemented in the main GUI package, although the GUI Editor package has also had some very minor modifications.

The classes of the main GUI package that underwent changes are `MainFrame` and `FileSelector`. The `MainFrame` view has gotten an additional menu item ("library"), from which users can select and import a new library into the project. The button on the menu is linked to the underlying method called `addExistingFile`, which previously only worked with single DEVS components, but can now also select and load DEVS library models (by employing the `LibraryGetter` class from the Library package).

In order to minimize confusion and maintain a relatively clean project tree, not all components of all included libraries should be shown in the project tree. The `FileSelector` class has thus been modified to show only the Root component of each library that is imported into the project, unless the user explicitly want to see all components. This is why the `Component` class from the Model package was extended with the `library` field. This field is used by the `FileSelector` class to decide whether a component should be shown or not. Only when the user enables the "Show internal library details" checkbox (show in the bottom left of Figure 5), will all library files, including those that are not root, be shown.

In the GUI Editor package, three classes have gotten a very minor addition. These are `AtomicDEVSPanel`, `CoupledDEVSPanel` and `MessagePanel`. In each of these classes a

method named `disableAll` has been added, that disables all user input elements in the GUI if the component that is being shown is a library component.

11.3 CREATION OF A BASIC DEVS LIBRARY

The two major parts of the thesis are: the implementation of a usable DEVS environment and the creation of a DEVS library. It is safe to say that this second part took at least as much time as the first, if not more. The DEVS library structure that was envisioned (and designed) is one that allows continued support and growth. This means that, as the use of the library become more widespread, the amount of people contributing to the library would grow continuously, as would the library itself.

However, there has to be at least a basic list of already developed library blocks such that the use and efficiency of this library (and DEVS libraries in general) can be evaluated, which could in turn motivate users start using the library. A lot of time was thus spent on generating a collection of library blocks that is large enough such that it can already be used for the generation of a big variety of DEVS projects. In the remainder of this section, this standard collection of blocks will be referenced to as the *basic library*

11.3.1 *Implemented library blocks*

Earlier, in Chapter 5, I have included a list of DEVS library blocks, each of which could potentially be interesting as a block in the basic library. However, not all blocks have been implemented. Table 8 shows exactly which blocks have been implemented and which have not.

Table 8: Envisioned library blocks. Crossed out blocks have not been implemented.

Mathematical Blocks	
SUM	ROUND
MULTIPLIER	SQUARE
GAIN	SQRT
ABSOLUTE	POWER
NEGATIVE	~~INTEGRATOR~~
INVERSE	~~DERIVATIVE~~
Logic Gate Blocks	
OR	AND
XOR	NOT
Generator Blocks	
GENERATOR (repeating)	GENERATOR (repeating, random)
GENERATOR (single)	~~PROGRAM~~

Queueing Blocks	
FIFO QUEUE	CIRCULAR QUEUE
LIFO QUEUE	~~MATCHING QUEUE~~
PRIORITY QUEUE	
Delay Blocks	
DELAY	~~DELAY (multiple)~~
DELAY (attribute)	~~DELAY (multiple, attribute)~~
Statistical Blocks	
SIMULATION TIME	COLLECTOR
TIMER	LIST SIZE
COUNT ITEMS	~~NUMERICAL LIST STATS~~
Data and Model Manipulation Blocks	
CHANGE NUMERICAL ATTRIBUTE	BATCH
~~ATTRIBUTE GET~~	UNBATCH
~~ATTRIBUTE SET~~	~~INPUT SWITCH~~
~~ITEM REPLICATOR~~	~~OUTPUT SWITCH~~
COMBINE	~~GATE~~

As you can see, most blocks have been implemented. Deciding which blocks to implement was based on a number of criteria:

- A block that is expected to get used a lot gets priority over blocks that are not as common. For example, a FIFO QUEUE is a lot more general than a MATCHING QUEUE, so it gets priority over it.
- The time it would take to implement the block. This does not mean the difficulty of the block. What it does mean is that, for example, after creating the SUM block from scratch, making the MULTIPLIER, GAIN, ABSOLUTE, ... blocks could be done very quickly by starting from the SUM block and changing what needs to be changed. Since implementing two similar blocks takes a lot less time than implementing two completely different blocks, priority was sometimes given to similar blocks.
- The functionality of some blocks can more easily be performed in an alternative way than others. Getting or setting an attribute can for example be very easily done by a transfer function, just like inverting a numerical value.

A system folder with all the implemented library blocks (not organized) can be seen in Figure 17. All library blocks that were implemented were also tested to work, by building a testing model that verifies a large amount of situations in which I imagined

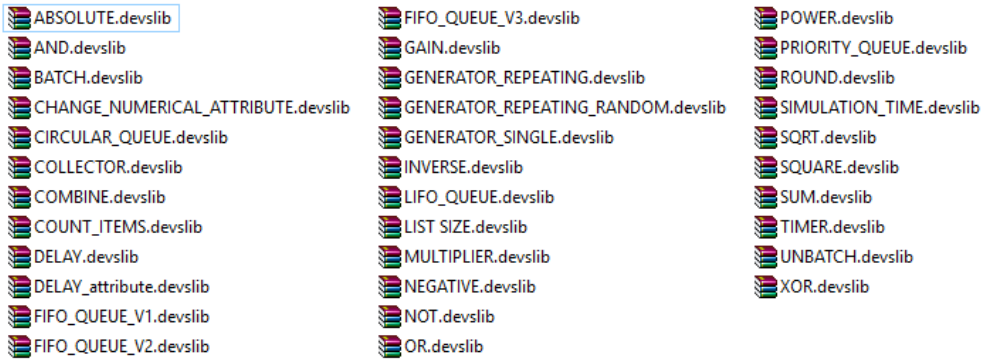

Figure 17: System folder containing all implemented ".devslib" files

the library block could be used, including in situations for which the block was not designed.

The criteria above lead to the selection of blocks that were eventually going to be implemented. The following list discusses for each block that was not implemented why others had gotten priority over it:

- INTEGRATOR and DERIVATIVE: Both blocks are anything but trivial to implement and would require prior investigation into how exactly to implement them. This, combined with the relatively low expected use in a DEVS environment, caused them to not be implemented.
- PROGRAM: The implementation of generator blocks was dictated by whether I needed a certain type of generator throughout the testing of other blocks. Since I had never had a use for a PROGRAM generator, it was not implemented.
- MATCHING QUEUE: I could not think of a lot of different situations in which this block would be necessary, compared to all other queues, which is why those got priority over it.
- DELAY (multiple) and (multiple, attribute): The functionality of these blocks can also be implemented through the use of a DELAY block and either a FIFO or a PRIORITY QUEUE. Even though they would thus not be extremely time consuming to implement, there were other blocks that got priority over them, since they could in theory be used already.
- NUMERICAL LIST STATS: This block was largely a choice of usefulness. It is hard to find a use for stats about a specifically numerical list. The functions that this block implement can furthermore be relatively easily be done within other blocks or even in transfer functions.
- ATTRIBUTE GET and SET: Even though these are very general operations, I still did not encounter any situation in which I would have benefited from having these blocks. Changing attributes of messages most frequently happens (in my experience) within individual blocks, not between them.

- ITEM REPLICATOR: Duplicating an item can also be done by simply adding another connection to the output port of the block that would be connected with the input port of the ITEM REPLICATOR. Although this is might mean a somewhat diminished structure, creating this block was not really necessary and thus got very low priority.
- INPUT and OUTPUT SWITCH: Although useful, these blocks are already quite specific and will not be used very frequently.
- GATE: This block's functionality could rather easily be replaced with an additional test in the condition pane of the external transition of an Atomic DEVS. Furthermore, I do not believe this block would be as frequently used as some others.

As a general remark, it is important to note that the library is, and should in theory always remain, a work in progress. This is also why not all blocks that were deemed interesting in the prior research have made it into the list of implemented blocks. A thesis has to be finished at a certain time, yet it is theoretically possible to indefinitely keep thinking of and implementing new blocks.

Part III

EVALUATION AND CONCLUSION

And thus the project reaches completion.

In the final part of this thesis we first look back on the newly created tool and evaluate it based on the traits originally introduced in Part i. This is all done in Chapter 12.

Chapter 13 continues the evaluation of this thesis by taking a last look at the implemented library blocks.

Given the size of this project and the imaginable capabilities of the tool and the library, there are still a lot of small and large pieces that can be developed further. Some ideas for, and views on, continued efforts are discussed in Chapter 14.

Concluding thoughts about the original idea, the process of creation and the end result of this thesis are brought together in Chapter 15, the final chapter of this text.

12

EVALUATION OF THE TOOL BASED ON USABILITY TRAITS

In Chapter 3, a long list of usability traits was shown. This list was originally created in the research that happened prior to this thesis[4] as a way of evaluating different DEVS environments by comparing them to each other. This chapter will extend the comparison by including the complete tool that was designed as part of this thesis.

You will be able to see that the new tool does not tick as many boxes as many other tools. This is because in this thesis we have focused very much on the implementation of the library, which was the main functionality for which the state-of-art was improved upon. Furthermore, this thesis is the work of a single student that has worked on it for the better part of a year, which is a lot less than any of the other tools.

For more information about the eight different categories or the traits themselves, I refer back to Chapter 3, in which they were explained. In the comparison tables below, the new tool will be referred to as *Thesis Tool*

12.1 AVAILABILITY

The comparison on Availability traits can be found in Table 9.

The thesis tool does not score well on availability traits, but that is because it is very much still a work in progress, that is not ready to be released to the public. For this reason, having a website or simple download available is not really applicable yet. At the time of writing the project is obviously still active.

Table 9: Evaluation of *availability* traits.

	Thesis Tool	DEVSimPy	VLE	DEVS-Suite	MS4Me	CD++Builder	PowerDEVS	AToMPM
Website		x	x	x	x	x	x	x
Website up to date (< 1 year)		x	x	x	x		x	x
Simple download			x	x		x	x	x
Open Source		x	x					x
Free		x	x	x		x	x	x
Active project (< 1 year)	x	x	x	x	x		x	

12.2 INSTALLATION

The comparison on Installation traits can be found in Table 10.

On installation traits, the thesis tool scores a lot better. Even though the tool, in its current form, does not really need an installation, an installation manual is therefor also not included. The tool does still require (or is best used with) NetBeans to run it, which does require installation, which is why the trait box has not been ticked.

Efforts have been made to keep the tool platform independent. External dependencies are limited since the tool only needs PyPDEVS[15], in combination with standard Python and Java. All other dependencies are included in the project.

Configuration is not necessary, the tool should work on any system out of the box.

Table 10: Evaluation of *installation* traits.

	Thesis Tool	DEVSimPy	VLE	DEVS-Suite	MS4Me	CD++Builder	PowerDEVS	AToMPM
No installation required		x		x				
Platform independent	x	x		x				x
Windows			x		x	x	x	
Linux			x				x	
Mac			(x)				(x)	
Limited external dependencies	x		x	x	x		x	
No manual configuration	x	x	x		x		x	x
Installation tutorial/manual available	x	x	x		x	x		x
First party installation tutorial/manual available	x	x	x		x			x

12.3 DOCUMENTATION

The comparison on Documentation traits can be found in Table 11.

A formal user manual is not available for the thesis tool. Because the tool is not ready for public deployment, the time required for the creation of a manual could be better spent somewhere else.

For the sake of this trait, I consider this thesis text being an academic paper, and as such it sais that there is one available.

A short tutorial will be included with this thesis text and can be found in Appendix 16. A number of example projects are included with the project's source files.

Table 11: Evaluation of *documentation* traits.

	Thesis Tool	DEVSimPy	VLE	DEVS-Suite	MS4Me	CD++Builder	PowerDEVS	AToMPM
User manual (English) available					x	x		
Academic paper(s) (English) available	x	x	x	x		x	x	x
Academic paper(s) (English) available on website	na			x				x
Tutorials available (video or textual)	x	x	x		x	x		x
First party tutorials available (video or textual)	x		x		x			x
First party example project	x		x		x	x	x	x

12.4 GENERAL FIT

The comparison on General Fit traits can be found in Table 12.

In order to being able to define complex DEVS models, some knowledge of a programming or scripting language will always be necessary, even if it means creating an entire new language from scratch. To use the thesis tool with some expectations, knowledge of Python is required.

The thesis tool is still an academic project, and should, at this time, probably not be used for the generation of DEVS models other than to further develop the tool itself. Therefor it is not fit for any academic (meaning to aid in other academic projects) or professional purpose.

Table 12: Evaluation of *general fit* traits.

	Thesis Tool	DEVSimPy	VLE	DEVS-Suite	MS4Me	CD++Builder	PowerDEVS	AToMPM
Programming language knowledge necessary	x	(x)	x	x	(x)	x	x	(x)
Fit for academic purpose		x		x	x	x	x	x
Fit for professional purpose					x	(x)		

12.5 INTERFACE

The comparison on Interface traits can be found in Table 13.

The general layout of the interface was based on some very popular programming and modelling environments. It fulfills the trait's requirements and as such the box is ticked.

Even though Swing allows the buttons and individual UI items to look system-

specific, it would still require a lot more work from someone with more graphical design expertise to create a truly modern-looking tool. What the tool looks like is much less important than the functionality of the tool at this point in time.

Table 13: Evaluation of *interface* traits.

	Thesis Tool	DEVSimPy	VLE	DEVS-Suite	MS4Me	CD++Builder	PowerDEVS	AToMPM
Clearly laid-out interface (no tutorial needed)	x	x	x	x	(x)	(x)	x	(x)
Modern interface look			x		x	x	x	x

12.6 MODEL DESIGN

The comparison on Model Design traits can be found in Table 14. In this table not all rows have been color-coded, since these traits depict a *choice* in design, and no choice is inherently better or worse.

The thesis tool was created as a textual based design. It was originally designed to work with the PyPDEVS simulation kernel specifically, for which a textual design lends itself more. DEVS naturally require a lot of textual internal design to specify its functionality, especially if blocks have to be designed from scratch. Should the library ever contains such a big collection that manually designing DEVS blocks is hardly ever necessary, than a more visually oriented tool might become more user-friendly.

There is functionality to create a graphical representation of both Atomic DEVS or Coupled DEVS models, so this box can be ticked.

Just like the included textual editor. In the thesis tool, each attribute of either Atomic or Coupled DEVS has its own small editor, which supports syntax highlighting for ports and state parameters.

The tool supports no advanced model validity checking. The only way to currently verify whether a block works is to inspect the simulation, or debug the compiled Python code should something go wrong.

The entire model can be compiled quickly with a single click of a button, so that trait is also implemented.

The tool does not exclusively use existing programming languages. It is built to modify a proprietary syntax, which describes all DEVS components. However, the tool has been designed in such a way that knowledge of the proprietary syntax is not ultimately necessary.

When a new DEVS component is made, an empty file in the proprietary syntax

is automatically generated by the tool. This would count as a generic template, as it contains everything except the internal logic of the block.

The proprietary syntax does not have syntax highlighting, but this is not a huge absence as the user does not necessarily need to come in contact with the proprietary syntax, unless he manually wants to modify it.

Table 14: Evaluation of *model design* traits.

	Thesis Tool	DEVSimPy	VLE	DEVS-Suite	MS4Me	CD++Builder	PowerDEVS	AToMPM
Visual based model design		x					x	x
Logic implementation from within tool		x					x	x
Majority of screen taken up by design pane		x					x	x
No unexpected visual editor quirks		x						
Intuitive visual editor controls		x					x	
High level of visual adaptability		x					x	
Textual based model design	x		x	x	x	x		
Visual representation of textual design	x			x	x	x		
Included textual editor	x				x	x		
Model validity checking		x			x	x	x	
No manual logic recompilation after changes	x	x			x	x	x	x
Exclusive use of existing programming languages		x	x	x			x	x
Generic template auto-generated	x	x					x	x
Use of proprietary language	x				x	x		
Syntax highlighting for proprietary language					x	x		
Proprietary language for atomic model	x				x			
Proprietary language for coupled model	x				x	x		

12.7 LIBRARY

The comparison on Library traits can be found in Table 15.

Library support was one of the most important functionalities of the tool. The only thing not currently available is an online model repository, but this is mainly because the tool itself is not even ready to be made available to the public yet, so there would be no use for an online library.

Table 15: Evaluation of *library* traits.

	Thesis Tool	DEVSimPy	VLE	DEVS-Suite	MS4Me	CD++Builder	PowerDEVS	AToMPM
Library support	x	x			(x)		x	
Expandable library	x	x					x	
Included library with basic building blocks	x						x	
Online model repository					x	x		

12.8 SIMULATION

The comparison on Simulation traits can be found in Table 16.

The simulation within the tool is done by the PyPDEVS simulation kernel. The only way to run the simulation is to specify an end time, and the complete simulation will be ran at once. The internal state of the model at any time can be seen in the simulation log.

Table 16: Evaluation of *simulation* traits.

	Thesis Tool	DEVSimPy	VLE	DEVS-Suite	MS4Me	CD++Builder	PowerDEVS	AToMPM
Simulation from within tool	x	x	x	x	x	x	x	x
Advanced simulation controls		x		x	x	x	x	x
Complete run simulation		x		x	x	x	x	x
Step-by-step simulation				x	x		x	x
Partial run simulation				x	x	x	x	x
Pause and resume simulation		x		x				x
Graphical representation of simulation				x	x			x
Live data shown during simulation	x			x	x		x	x
Data stored after simulation			x	x	x	x	(x)	x
Live log shown during simulation	x			x	x			x
Log stored after simulation		x		x	x	x	x	x

13

EVALUATION OF THE LIBRARY

The main sources that were used for creating the list of library blocks were PowerDEVS[1], SimEvents[9] and the Extend user manual[13]. We will compare the blocks of the thesis library to the libraries of those tools, as well as one other tool that was referenced in this text that contains some sort of library or model repository, which is DEVSimPy[2].

MS4Me[11] and CD++Builder[3] do deserve to be mentioned. They do not actually support block libraries, but do have online model repositories (CD++ has a very extensive one). This does not replace a good block library though, since these cannot be easily used for the creation of new models. However, they at least provide example models that users can use for inspiration.

Table 17 below shows for each block which tools have some form of implementation for it. An *x* in the table means that the tool contains an element that matches (or at least somewhat resembles) the representation of the library block as it was defined in Chapter 5.

Below this table all sources and DEVSimPy are very briefly discussed in terms of their libraries.

Table 17: Comparison of libraries of different tools with regards to the blocks implemented for this thesis.

	Thesis Blocks	PowerDEVS	SimEvents	Extend	DEVSimPy
Mathematical Blocks					
SUM	x	x		x	
MULTIPLIER	x	x		x	
GAIN	x	x			
ABSOLUTE	x				
NEGATIVE	x				
INVERSE	x	x			
ROUND	x				
SQUARE	x	x			
SQRT	x				
POWER	x	x		x	
Logic Gate Blocks					
OR	x			x	

AND	x			x	
XOR	x				
NOT	x			x	
Generator Blocks					
GENERATOR (repeating)	x	x	x	x	
GENERATOR (single)	x	x	x	x	
GENERATOR (repeating, random)	x	x			x
Queueing Blocks					
FIFO QUEUE	x		x	x	
LIFO QUEUE	x		x	x	
PRIORITY QUEUE	x		x	x	
CIRCULAR QUEUE	x				
Delay Blocks					
DELAY	x	x		x	
DELAY (attribute)	x			x	
Statistical Blocks					
SIMULATION TIME	x	x			
TIMER	x				
COUNT ITEMS	x			x	
COLLECTOR	x				x
LIST SIZE	x				
Data and Model Manipulation Blocks					
CHANGE NUMERICAL ATTRIBUTE	x			x	
COMBINE	x			x	
BATCH	x		x	x	
UNBATCH	x		x	x	

- **PowerDEVS**: Library-wise, PowerDEVS is very much focused on mathematical blocks. It has versions for almost all mathematical blocks that were designed here, but extend this with vector operations and much more. As for generators, this tool has many, single instant, repeating random, wave generators, poisson generators, and much more. In other categories is scores less favorably. Even though DEVS technically supports any type of event or message to be sent between blocks, PowerDEVS seems to only support numerical values.

  One interesting thing to note is that PowerDEVS also allows the user to simulate PetriNets, which demonstrates the capabilities of DEVS as a modeling formalism.

- **SimEvents**: SimEvents was a source for some of the blocks created for the thesis and this can be verified by the fact that seven blocks were shared. SimEvents relies on "entities" as its messages, so it is no surprise that core mathematical and logic gate blocks are not implemented.

- **Extend**: Extend was another major source of inspiration. A total of 18 blocks are shared between Extend and the thesis library. The only place Extend really lags behind in in the mathematical blocks. This is clearly not where its focus lies.
- **DEVSimPY**: DEVSimPy comes packaged with three library categories: Collectors, Generators and Phidgets. There are only a total of 10 (working) blocks in those three categories, of which only two correspond (approximately) to blocks in the thesis library. It is clear that, even though DEVSimPy supports libraries, they have not kept continued support going (or at least not in a way such that it can be easily accessed).

As you can see, most blocks have a comparative element in at least on of the sources. Those that do not, are blocks that automatically come to mind when thinking of others (for example the mathematical blocks: when thinking about a SQUARE block, a SQRT block comes to mind automatically), or blocks that just seemed useful. The reason for selection for each block is discussed in detail in the the prior research[4].

A lot of blocks in the thesis library assume a numerical input or output (all mathematical and logic gate blocks, as well as the random GENERATOR and CHANGE NUMERICAL ATTRIBUTE blocks). Although those do not really take full advantage of the DEVS formalism and the fact that a DEVS message could in theory be anything, they do allow for the easy creation of some relatively simple models. This is a good way for novice DEVS users to get familiar with the formalism, and encourage them to educate themselves further.

All things considered, this library lets the user at least skip some of the most basic tasks, which are very dull to perform. Examples are the implementation of generators, delay blocks or queues, blocks for which the chance of being used in anything but the most trivial models could be considered expected. Now they can just import the block and they are ready to go.

14

FUTURE WORK

Given the rather large scale of this project and the time frame of a "mere" Master's thesis to develop it, there is obviously still some work left to be done before the tool is ready for public deployment. However, both the complete tool and the library have capabilities that, with some further efforts, allow them to grow into rather successful products, either together or individually.

CONTINUED DEVELOPMENT ON THE TOOL

Of course, one could simply look at the list of traits and starting implementing them one by one. However, some possible expansions are undoubtedly more useful than others.

One very big improvement to the tool would be having some more simulation options (such as step-by-step execution or pausing and resuming the simulation). Built-in DEVS debugging is related to this. For inspiration one could look at the DEVSDebugger architecture[14].

Library handling could be made all internal to the tool, providing functionality to create libraries and possibly browsing an online repository of library blocks and importing straight from there. This has not been implemented in any of the other tools that were investigated the prior research[4].

PROSPECTS FOR THE LIBRARY

The library could be adapted to support PyPDEVS files (as well as files in the proprietary syntax). This would allow for the distribution of black-box libraries and also makes is such that the library blocks could be more easily used by any DEVS tool or project using PyPDEVS as simulation kernel.

A logical progression of the library is having support for many users to access it and add to it. This way, the size of the library could potentially grow linearly with the amount of users, with it increasing the efficiency with which any models can be created. Possible implementations would be a central database of blocks generated by verified users, or a more open-sourced free repository that everyone can access and add to. This of course depends on what future contributors envision for the library.

Undoubtedly, the most important future task for those working on the library is to have additional blocks added to it, whichever direction it might go.

15

CONCLUSION

This thesis was created in an attempt to improve the efficiency with which DEVS models could be generated and simulated. The current state-of-art, which exists of a whole range of individual tools, has many interesting ideas, but lacks a single product in which all of these ideas are combined. Which is exactly what this thesis set out to do. The problem would be tackled in two distinct, yet complementary, ways: creating a usable DEVS creation environment and implementing a library of DEVS blocks.

PRELIMINARY QUESTIONS

The two first questions that had to be asked then, were: "What defines a usable DEVS creation environment?" and "Which blocks are useful for the DEVS formalism, and where do we find these?". Both of these questions were answered in preliminary research, the results of which were key in the further working-out of both the DEVS environment and library elements.

One important thing to take away from the research on existing environments is that it is possible to define and evaluate usability in this context by a constructing a list of *traits*: elements of the design of a tool that improve functionality and usability. These traits therefore served as a basic guideline for the design of a new DEVS tool.

Considering the library blocks, it is essential to realize that DEVS can be used to represent a large range of other modeling formalisms. The implication of this is that there is almost no limit to what could be supported by DEVS blocks. Given the scope of this thesis, a selection of library blocks was made to represent this fact, by having many different categories of blocks.

CONSTRUCTED RESULTS

The practical part of this thesis led to the implementation of both the tool and the library.

THE TOOL During the creation of the DEVS tool, the focus lay on the efficient creation of DEVS models, and thus on the traits that describe this functionality. Other traits, such as: limited external dependencies, platform independence and clear lay-out, are more concerned with the tool as a whole, and not with specific functionality, so those were also kept into account for the entire implementation process.

In the final evaluation, it is clear that this strategy was applied, as the tool usually scored either very well or very poor on entire categories of traits. Some other tools score more variably, thus showing that possibly more functionalities were

implemented, but none are implemented completely (based on the list of traits). I am of the opinion that adding functionality is usually less challenging than changing existing functionality, which is why this strategy was chosen.

THE LIBRARY Selecting which blocks would be implemented was a matter of evaluating the usefulness of each block and comparing this to that of others, prioritising some over others based on this evaluation. The creation of DEVS library blocks also served as a way of testing and evaluation the tool, since all blocks were generated using it.

At this time, it would be false to say that the contents of the library are better or more extensive compared to existing tools. However, this is beside the point of the original goal regarding the library, which was to design a structure that that would benefit from multiple users adding to it over time. The most important feature right now is having the sample blocks implemented and working.

FINAL REMARKS

In spite of the limitations that exist within the tool and the library, a solid foundation for future work has been laid by this thesis. It shows that it is possible to create a DEVS environment that is both functional and usable, in fact, we have shown that these two words go hand in hand. No solid research has been done on evaluating the speed and efficiency of the tool and library that result from this thesis. However, it is clear from my own experience that together they are at least capable of building any DEVS model, with a solid belief that they improve on the state-of-art in some specific areas.

Part IV

APPENDIX

16

TUTORIAL

This tutorial was created to demonstrate how tools can be developed, compiled and simulated with the thesis tool.

The model that will be created is a relatively simple "trafficlight" model, in which there are two main elements: A *traffic light*, which switches between green, yellow and red automatically, and a *police officer*, who can manually override the traffic light, such that it starts blinking yellow, and keeps doing this until the officer tells it to continue with its normal operation.

The creation of this model will be documented step-by-step, and illustrated with screen captures of the tool. The first major step is to create the DEVS project:

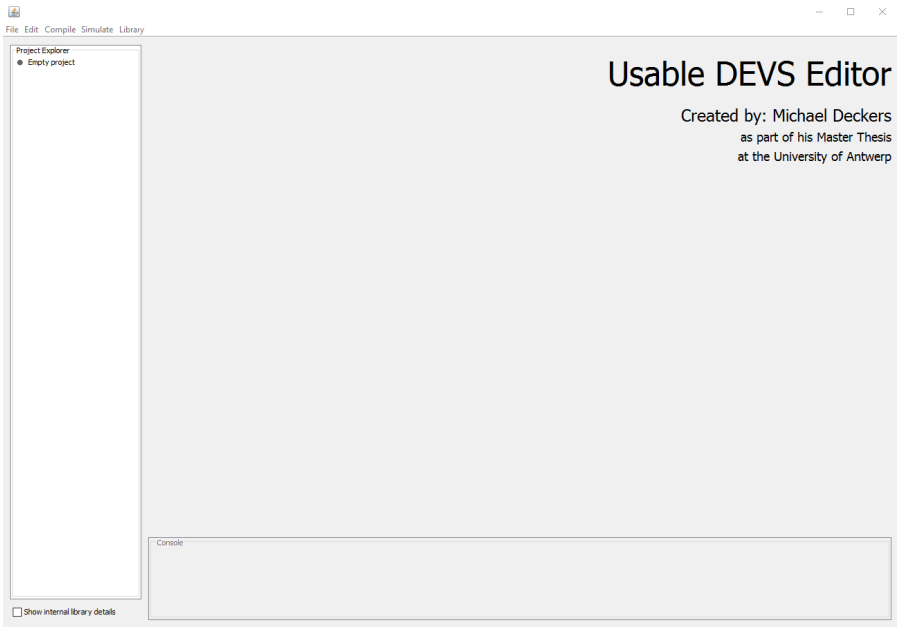

Figure 18: Tutorial step 1 result

1. Start the thesis tool. (Figure 18)
2. Create a new project: *File -> New DEVS Project* or Ctrl-N
3. Browse to the location where you want the project to be stored, and give the project a name. (Figure 19)
4. Select the locations for the compilation output and simulation settings. I usually save this in a folder named *output*, located at the same location as the project. Of course this is your own choice. Here, existing (non-library) DEVS Components

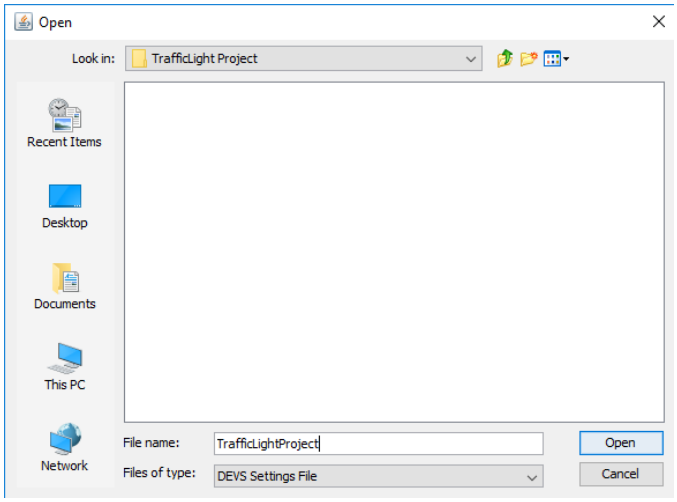

Figure 19: Tutorial step 3 result

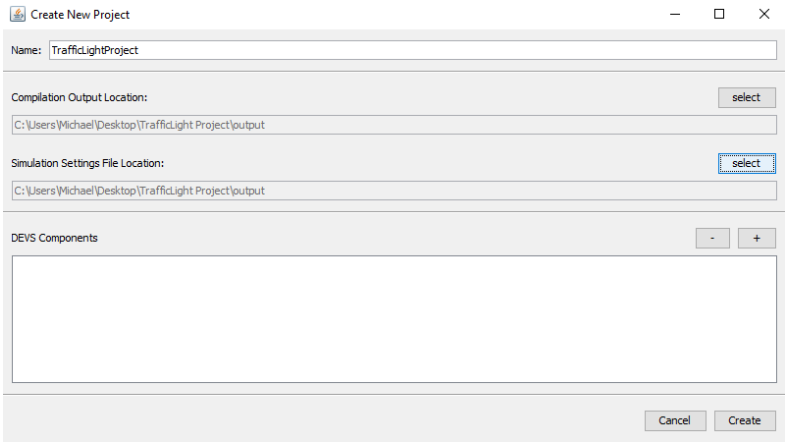

Figure 20: Tutorial step 4 result

can be added to the project, but since we will be building everything from scratch, we will not be using this. (Figure 20)

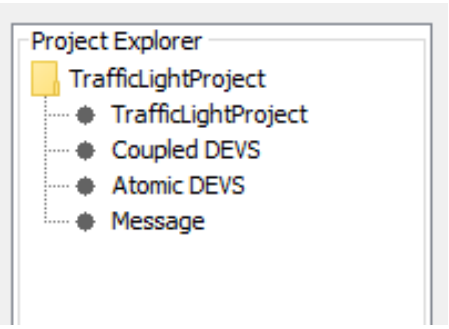

Figure 21: Tutorial step 5 result

5. The project is successfully created, but no DEVS Components have been added yet. (Figure 21)

Now that the project is created, it is time to add our first element: the traffic light. For this, we will create a new Atomic DEVS:

6. Create a new Atomic DEVS: *File -> Add New... -> New Atomic DEVS* or Alt-Shift-A

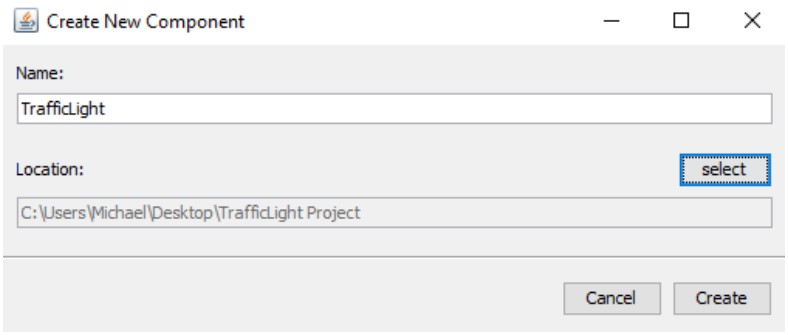

Figure 22: Tutorial step 7 result

7. Give the new component a name (TrafficLight) and select the location, the current project location is selected as standard when clicking the select button. (Figure 22)

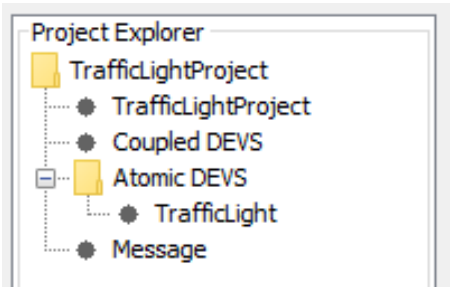

Figure 23: Tutorial step 8 result

8. A new Atomic DEVS block is now created. (Figure 23)
9. Double-Click the TrafficLight block in the Project Explorer to open the Atomic DEVS editor. (Figure 24)
10. Although we could do this in a less difficult way (specifying the values directly in the time advance) we will add parameters to the model that contain for each color how long the light should stay on. (Figure 25)
11. We also require an input port (to get commands from the police officer), so we will add one input port. (Figure 26)
12. We need four states in this Atomic DEVS: Green, Yellow, Red and Blinking. One state for each type of light the traffic light can illuminate. When the simulation starts we want the green light to be on, so this is our starting state. (Figure 27)
13. The next step is to implement our internal transitions. Click on the "Internal Transition" tab. (Figure 28)

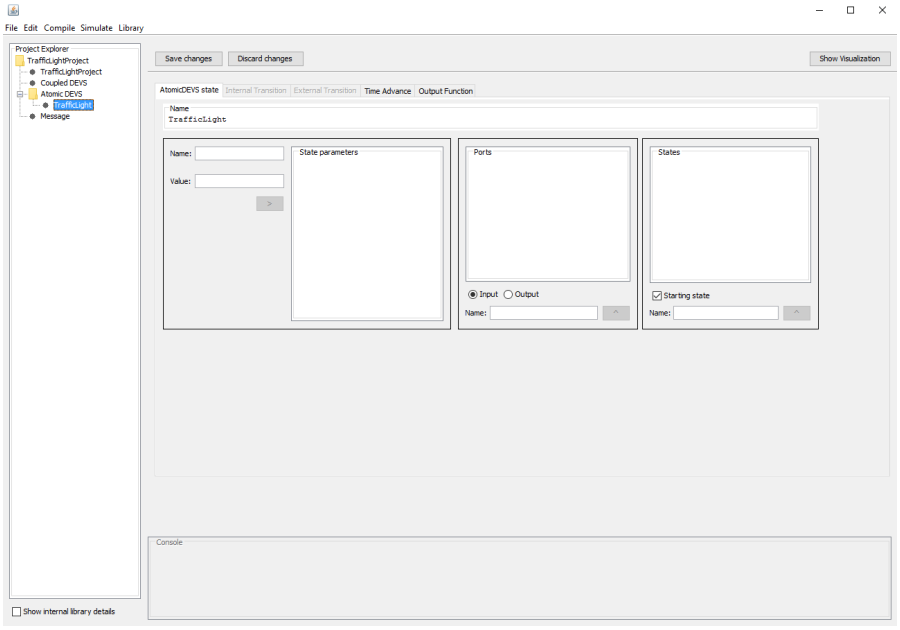

Figure 24: Tutorial step 9 result

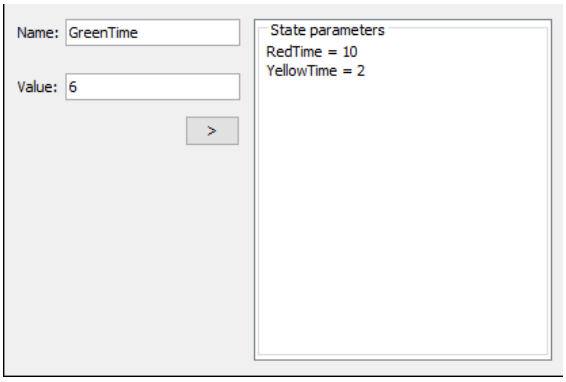

Figure 25: Tutorial step 10 result

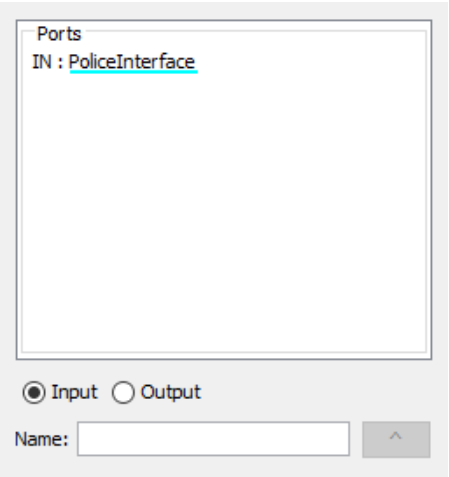

Figure 26: Tutorial step 11 result

Figure 27: Tutorial step 12 result

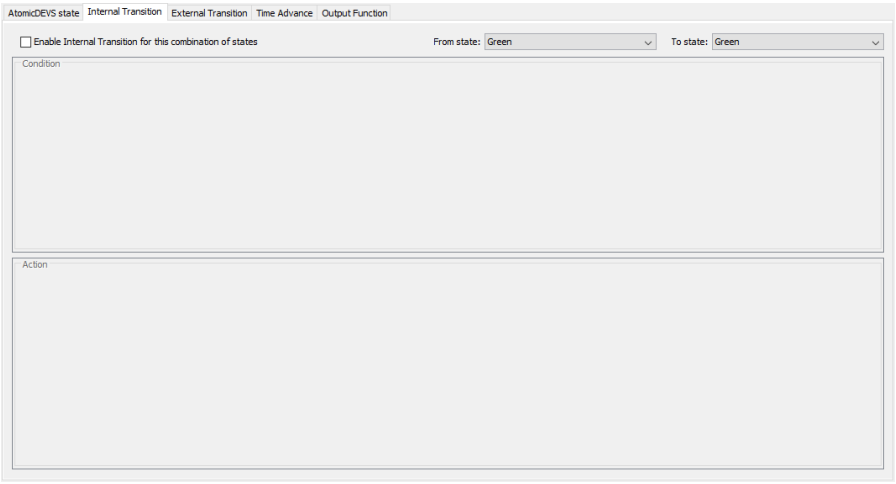

Figure 28: Tutorial step 13 result

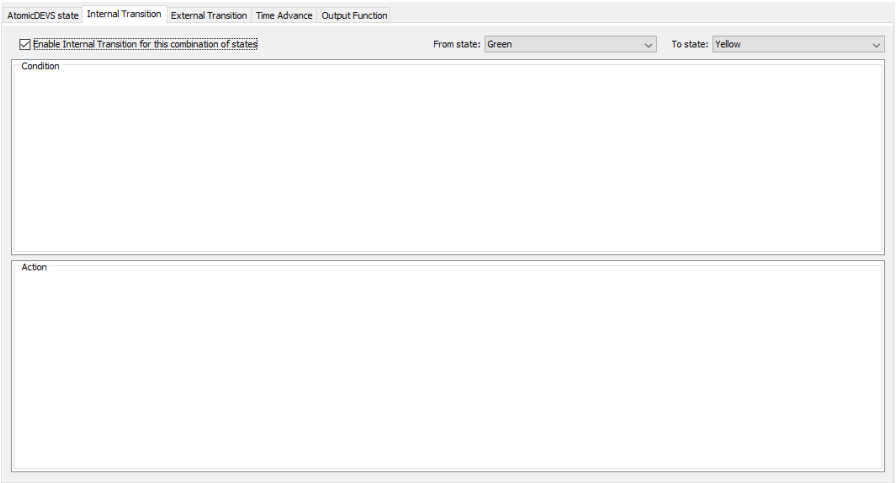

Figure 29: Tutorial step 14 result

14. Internal transitions are triggered by the expiration of the time advance, and as such allow for autonomous operation of the block. Since we want the traffic light to change its color from green to yellow to red automatically, we will have to implement this with internal transitions. We will show the example for going from green to yellow, the other ones have to be implemented in exactly the same way. Changing the drop down boxes on the top right of the editor, select "Green" as from state and "Yellow" as to state. Then tick "Enable Internal transition for this combination of states". The Condition and Action text boxes will now be activated. (Figure 29)

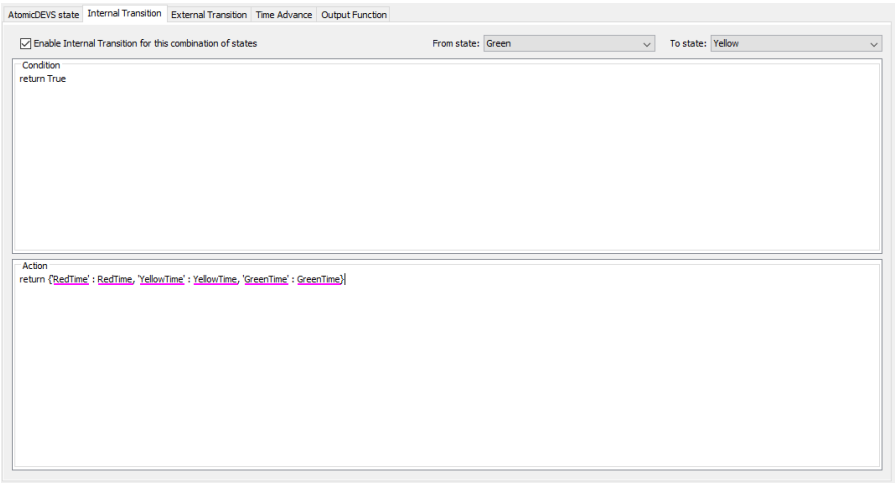

Figure 30: Tutorial step 15 result

15. In the condition field, we can choose whether we want this transition to fire or not. This could be helpful if for example a green light could go to either yellow and then red, or to red immediately, depending on the state of the block. In this case you would have two internal transitions going out from the Green state, and you would need to be able to specify in some way which one to use.
However, this is not necessary for us, since the Green state will have only one internal transition. The value of the Condition field will therefore just be `return True`
In the action field we could modify the state of the system by modifying the parameters if we wanted to, however, this is not necessary for us (since we do not have parameters), so we can just return the parameters unmodified by writing `return {'RedTime' : RedTime, 'YellowTime' : YellowTime, 'GreenTime' : GreenTime}`. (Figure 30)

16. Repeat step 15 for the following internal transitions: Yellow to Red and Red to Green.

17. We will need a total of four external transitions. The first three are allocated for the following reason: when the police officer tells the traffic light to go blinking, it could be in any of the Green, Yellow or Red states. We therefore need to enable an external transition from all of these states to the Blinking state.
In the condition field, we could again just write `return True`, since nothing

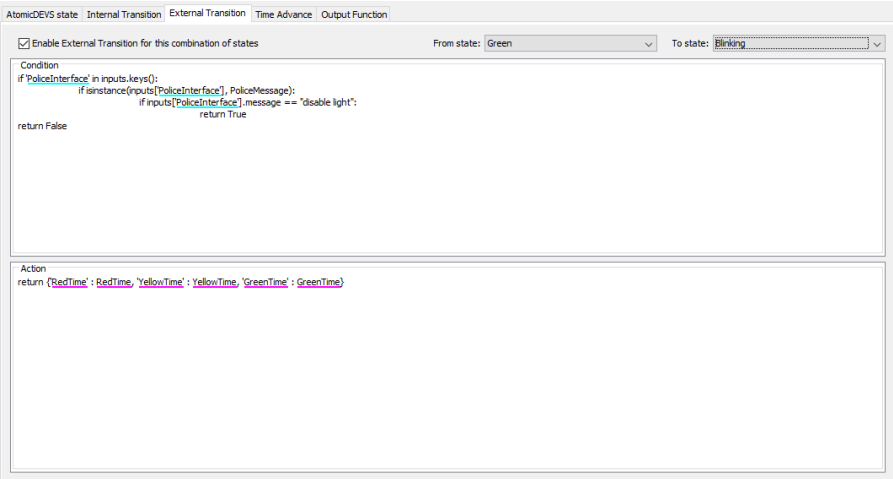

Figure 31: Tutorial step 17 result

could in theory go wrong, but we will write in some safety to be sure. The following code will be added:

```
if 'PoliceInterface' in inputs.keys():
    if isinstance(inputs['PoliceInterface'], PoliceMessage):
        if inputs['PoliceInterface'].message == "disable light":
            return True
return False
```

The first test checks whether the a message had arrived at the "PoliceInterface" port specifically (there are not other ports, but if there were this had been necessary). The second text checks whether the message that had arrived was a "PoliceMessage" (this will be created later). The third test checks whether the police told the light to disable its autonomous operation and go to the Blinking state. If all tests succeed, return `True` (fire the transition), if any of them fails, do not fire (return `False`).
We do not need to change the state of the system during these external transitions, so we can again just write `return {'RedTime' : RedTime, 'YellowTime' : YellowTime, 'GreenTime' : GreenTime}` in the action pane. (Figure 31)

18. Repeat step 17 for the following external transitions: Yellow to Blinking and Red to Blinking.

19. The final external transition fires when the police officer tells the light to resume its autonomous operation. For this, we create an external transition from the Blinking state to the Red state. The contents of the Condition pane are very similar to the other external transition. The only thing that needs to be changed is the following line: `if inputs['PoliceInterface'].message == "disable light":`, which will

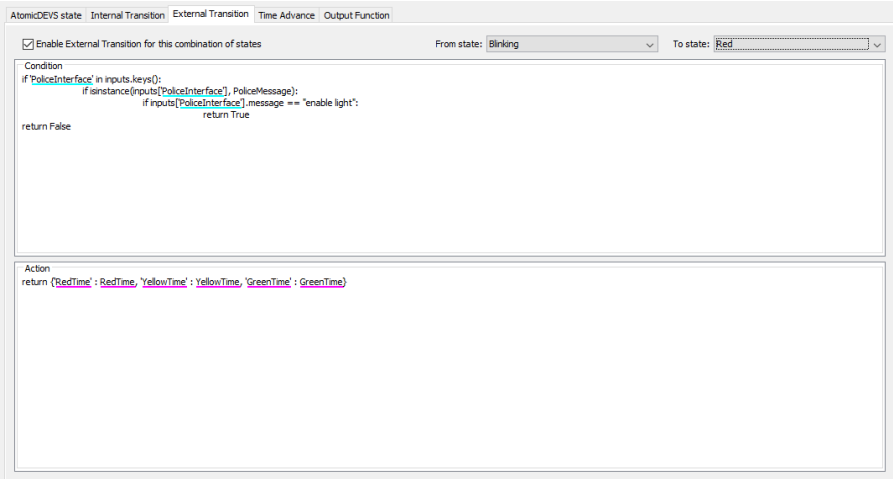

Figure 32: Tutorial step 19 result

have to become `if inputs['PoliceInterface'].message == "enable light":`. The action field value is exactly the same as all other transitions. (Figure 32)

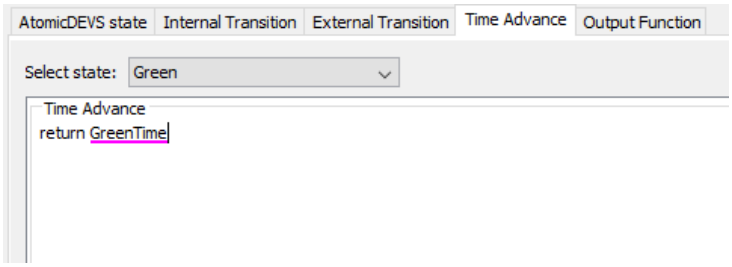

Figure 33: Tutorial step 20 result

20. The next step is to set the time advance for each of the four states. For the Green, Yellow and Red states, we had previously created the parameters. For each of these three states, return the corresponding parameter. (Figure 33).

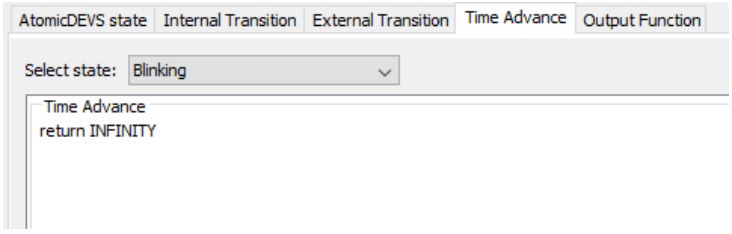

Figure 34: Tutorial step 21 result

21. We do not want the system to ever change autonomously when it is under control of the police officer (in the Blinking state), which is why we will write `return INFINITY`. (Figure 34)

22. The final step is to set the output function for each state. Since the traffic light will never output anything, we can write `return {}` at all four states.

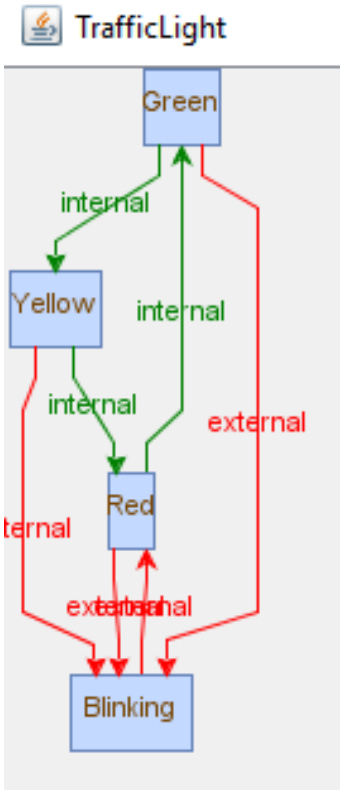

Figure 35: Tutorial: graphical representation of TrafficLight Atomic DEVS

This concludes the creation of the Atomic DEVS for the traffic light. The internals of the state can be shown by clicking the button labeled "Show Visualization". Make sure to save the model before clicking this. It should look like Figure 35. During the creation of this Atomic DEVS, we used and briefly talked about a "PoliceMessage". This is a DEVS Message, which we will create next:

23. Create a new Message: *File -> Add New... -> New Message* or Alt-Shift-M
24. Repeat step 7 for the new message, and name it "PoliceMessage" (without the quotation marks).
25. Double-click the item in the Project Explorer. The DEVS Message editor is opened. (Figure 36)
26. Add a new parameter, named "message". (Figure 37)

These are all the steps needed to create the PoliceMessage. The next step is to create the PoliceOfficer Atomic DEVS. This will be discussed in less detail as the creation of the traffic light, since much of the process is comparable:

27. Create a new Atomic DEVS named "PoliceOfficer".
28. Add one output port named "Command".
29. Create two states: "OutControl" and "InControl". Make "OutControl" the starting state.

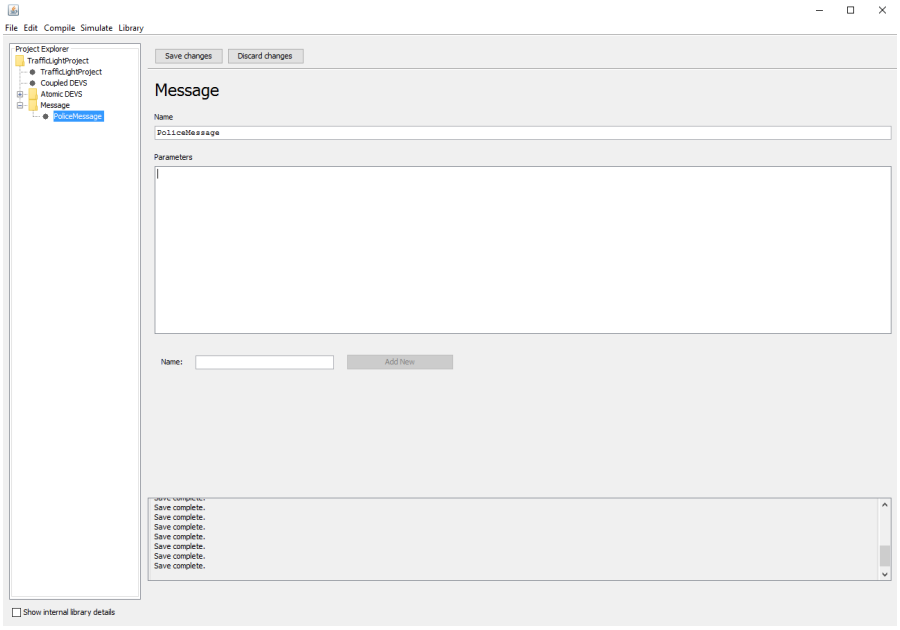

Figure 36: Tutorial step 25 result

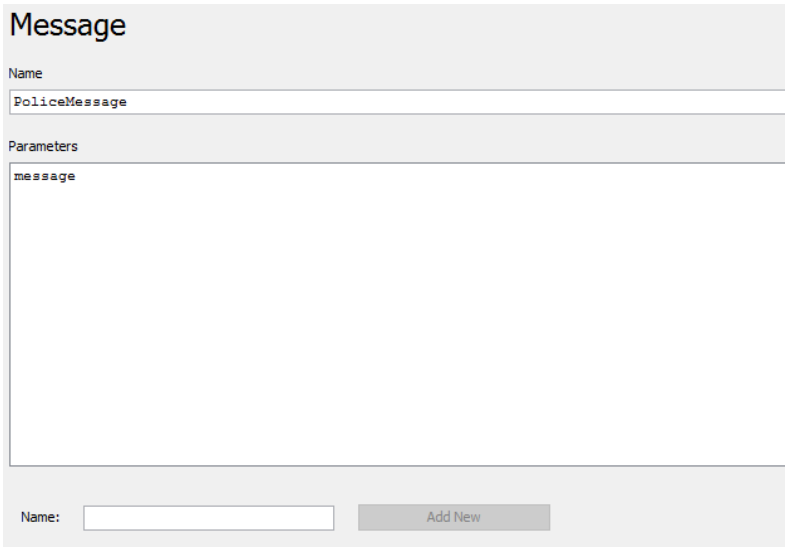

Figure 37: Tutorial step 26 result

30. Create internal transitions going from OutControl to InControl and vice versa. For both the transitions the condition can be `return True` and the action can be `return {}`.

31. We do not need any external transitions.

32. We want to leave the traffic light running autonomously for 45 seconds, and then have the police officer take over for 15 seconds. In the time advance, write `return 45` for the OutControl state and `return 15` for the InControl state.

33. In the output function we do have to do something a little bit more complex. We want to create a PoliceMessage that has "disable light" as the value of its

`message` parameter when the Police Officer want to disable the traffic light, and a PoliceMessage that says "enable light" when he wants to give back control to the light.
Thus, in the output function of the OutControl state, write `return \{'Command' : PoliceMessage(message=''disable light'')\}` and in the output function of the InControl state, write `return \{'Command' :PoliceMessage(message=''enable light'')\}`. This creates the PoliceMessage and sends it to the Command port, which will later be connected to the PoliceInterface port of the TrafficLight

This completes the PoliceOfficer Atomic DEVS. Now all that is left to do is create the Coupled DEVS that links the two together:

34. Create a new Coupled DEVS: *File -> Add New... -> New Coupled DEVS* or Alt-Shift-C

35. Repeat step 7 for the new Coupled DEVS, and name it "TrafficLightCoupled" (without the quotation marks).

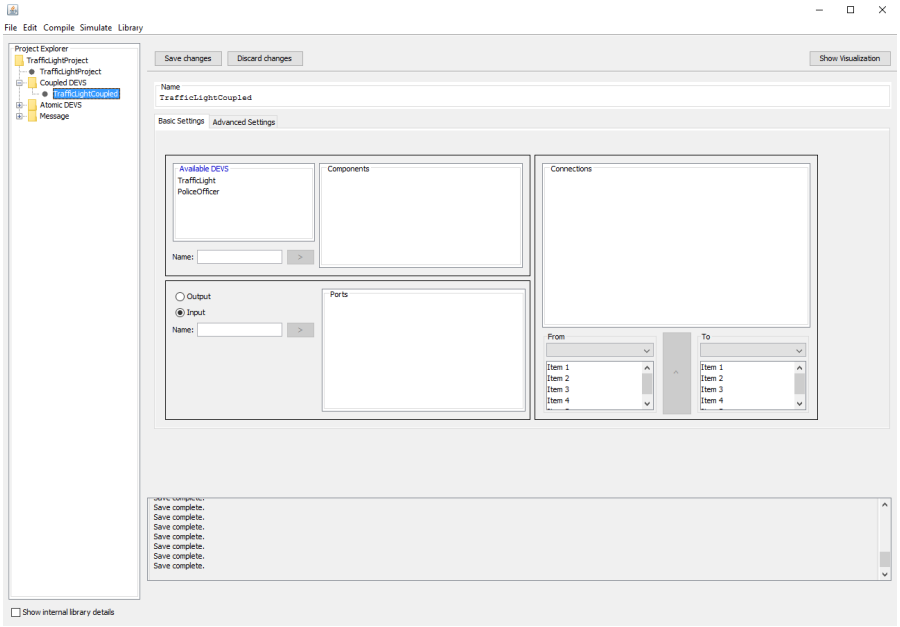

Figure 38: Tutorial step 36 result

36. Double-Click the newly generated Coupled DEVS in the Project Explorer to open the Coupled DEVS editor. (Figure 38)

37. You can see that the previously created Atomic DEVS components are already in the "Available DEVS" list. All components (Coupled and Atomic DEVS) in the project will be shown here. Select the TrafficLight block and name it "light" before clicking the add ">" button. Next, select the PoliceOfficer block and name it "cop" before adding it. (Figure 39)

38. Our Coupled DEVS block is the root block and will not need any ports. We can ignore these.

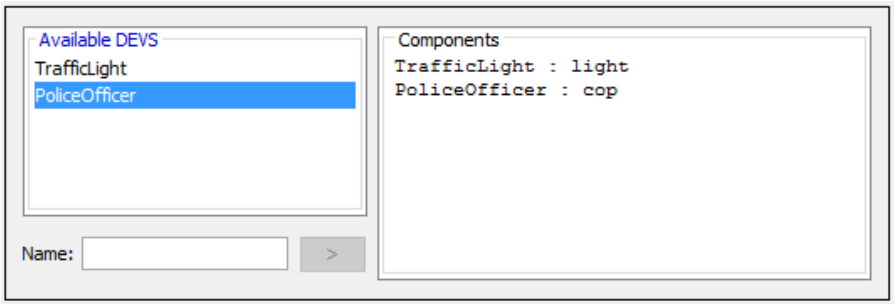

Figure 39: Tutorial step 37 result

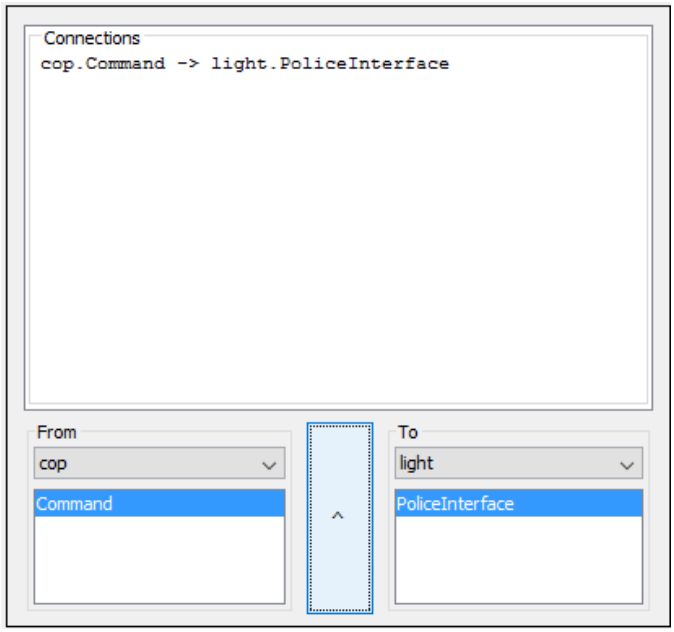

Figure 40: Tutorial step 39 result

39. Create a connection between the PoliceOfficer (cop) block's outport (Command) to the TrafficLight (light) block's inport (PoliceInterface). (Figure 40)

40. Click the "Advanced Settings" tab.

41. We want the *cop* to have priority over the *light*, so we change this by selecting *light* and clicking "down" (or selecting *cop* and clicking "up"). We do not need to pass any parameters or implement transfer functions, so we can ignore these fields. (Figure 41)

The model is now complete. The next steps explain how to compile and simulate the model we have just created:

42. Save all components

43. Compile the model by clicking *Compile -> Compile to PyPDEVS*. The Console should say that the compilation is complete.

44. Starting the simulation can be done by clicking *Simulate -> Run Simulation*.

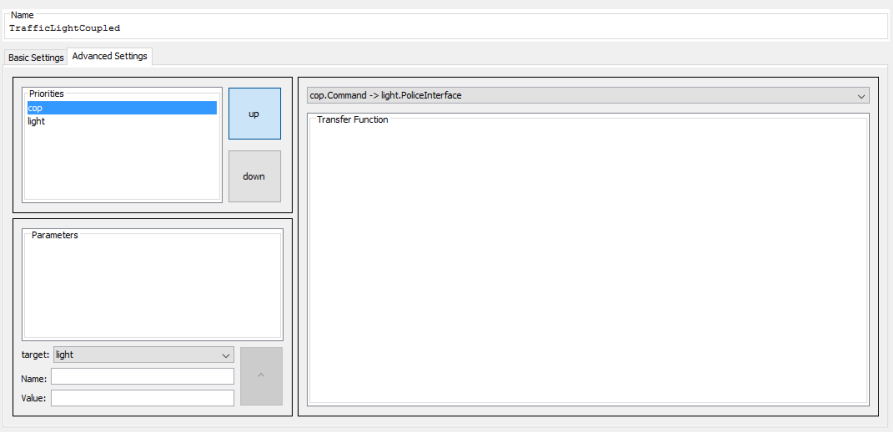

Figure 41: Tutorial step 41 result

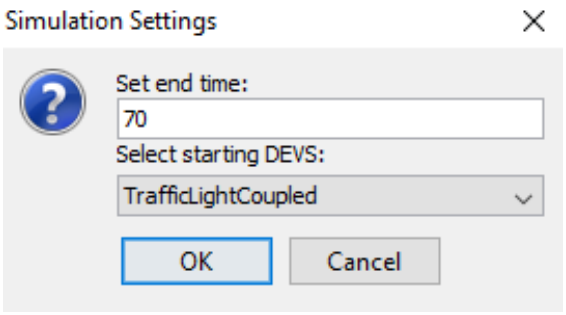

Figure 42: Tutorial step 45 result

45. In the "Simulation Settings" window specify that we want the simulation to run for 70 (simulated) seconds, by setting the end time to 70. The correct Coupled DEVS block has already been selected, but if this were not the case (should there be more than one Coupled DEVS blocks in the project), then select the correct root block. Click OK. (Figure 42)

46. The next window is currently empty, but this would allow us to change the values of the parameters specified in the root Coupled DEVS block, had we added any.

If all steps have been followed correctly, the tool should now run the simulation and show the output in the UI of the tool. In the end, the tool should look like Figure 43. The complete simulation is copied into Code Sample 9.

The creation and simulation of the Traffic Light model is now finished.

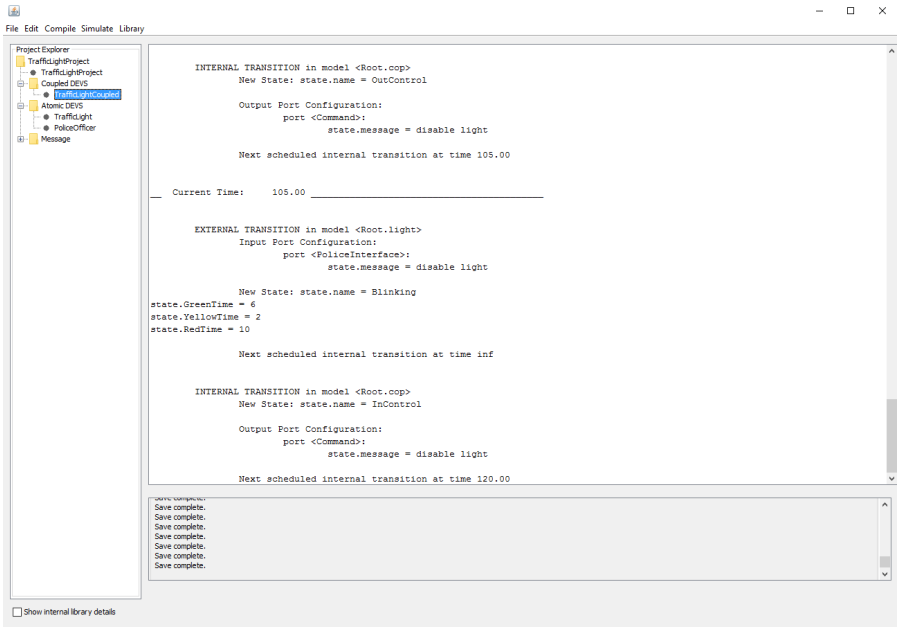

Figure 43: Tutorial: Simulation output in the tool

```
__  Current Time:       0.00 _____________________________________________


  INITIAL CONDITIONS in model <Root.cop>
    Initial State: state.name = OutControl

    Next scheduled internal transition at time 45.00


  INITIAL CONDITIONS in model <Root.light>
    Initial State: state.name = Green
state.GreenTime = 6
state.YellowTime = 2
state.RedTime = 10

    Next scheduled internal transition at time 6.00


__  Current Time:       6.00 _____________________________________________


  INTERNAL TRANSITION in model <Root.light>
    New State: state.name = Yellow
state.GreenTime = 6
state.YellowTime = 2
state.RedTime = 10

    Output Port Configuration:
    Next scheduled internal transition at time 8.00


__  Current Time:       8.00 _____________________________________________
```

```
33
34
35    INTERNAL TRANSITION in model <Root.light>
36      New State: state.name = Red
37  state.GreenTime = 6
38  state.YellowTime = 2
39  state.RedTime = 10
40
41      Output Port Configuration:
42      Next scheduled internal transition at time 18.00
43
44
45  __  Current Time:      18.00 _____________________________________________
46
47
48    INTERNAL TRANSITION in model <Root.light>
49      New State: state.name = Green
50  state.GreenTime = 6
51  state.YellowTime = 2
52  state.RedTime = 10
53
54      Output Port Configuration:
55      Next scheduled internal transition at time 24.00
56
57
58  __  Current Time:      24.00 _____________________________________________
59
60
61    INTERNAL TRANSITION in model <Root.light>
62      New State: state.name = Yellow
63  state.GreenTime = 6
64  state.YellowTime = 2
65  state.RedTime = 10
66
67      Output Port Configuration:
68      Next scheduled internal transition at time 26.00
69
70
71  __  Current Time:      26.00 _____________________________________________
72
73
74    INTERNAL TRANSITION in model <Root.light>
75      New State: state.name = Red
76  state.GreenTime = 6
77  state.YellowTime = 2
78  state.RedTime = 10
79
80      Output Port Configuration:
81      Next scheduled internal transition at time 36.00
82
83
84  __  Current Time:      36.00 _____________________________________________
85
86
87    INTERNAL TRANSITION in model <Root.light>
88      New State: state.name = Green
89  state.GreenTime = 6
```

```
state.YellowTime = 2
state.RedTime = 10

    Output Port Configuration:
    Next scheduled internal transition at time 42.00


__  Current Time:      42.00 _____________________________________________


  INTERNAL TRANSITION in model <Root.light>
    New State: state.name = Yellow
state.GreenTime = 6
state.YellowTime = 2
state.RedTime = 10

    Output Port Configuration:
    Next scheduled internal transition at time 44.00


__  Current Time:      44.00 _____________________________________________


  INTERNAL TRANSITION in model <Root.light>
    New State: state.name = Red
state.GreenTime = 6
state.YellowTime = 2
state.RedTime = 10

    Output Port Configuration:
    Next scheduled internal transition at time 54.00


__  Current Time:      45.00 _____________________________________________


  EXTERNAL TRANSITION in model <Root.light>
    Input Port Configuration:
      port <PoliceInterface>:
        state.message = disable light

    New State: state.name = Blinking
state.GreenTime = 6
state.YellowTime = 2
state.RedTime = 10

    Next scheduled internal transition at time inf


  INTERNAL TRANSITION in model <Root.cop>
    New State: state.name = InControl

    Output Port Configuration:
      port <Command>:
        state.message = disable light

    Next scheduled internal transition at time 60.00
```

```
147
148
149 __  Current Time:      60.00 ___________________________________________
150
151
152   EXTERNAL TRANSITION in model <Root.light>
153     Input Port Configuration:
154       port <PoliceInterface>:
155         state.message = disable light
156
157     New State: state.name = Blinking
158 state.GreenTime = 6
159 state.YellowTime = 2
160 state.RedTime = 10
161
162     Next scheduled internal transition at time inf
163
164
165   INTERNAL TRANSITION in model <Root.cop>
166     New State: state.name = OutControl
167
168     Output Port Configuration:
169       port <Command>:
170         state.message = disable light
171
172     Next scheduled internal transition at time 105.00
173
174
175 __  Current Time:     105.00 ___________________________________________
176
177
178   EXTERNAL TRANSITION in model <Root.light>
179     Input Port Configuration:
180       port <PoliceInterface>:
181         state.message = disable light
182
183     New State: state.name = Blinking
184 state.GreenTime = 6
185 state.YellowTime = 2
186 state.RedTime = 10
187
188     Next scheduled internal transition at time inf
189
190
191   INTERNAL TRANSITION in model <Root.cop>
192     New State: state.name = InControl
193
194     Output Port Configuration:
195       port <Command>:
196         state.message = disable light
197
198     Next scheduled internal transition at time 120.00
```

Code 9: Simulation output of the TrafficLight model

BIBLIOGRAPHY

[1] Federico Bergero and Ernesto Kofman. PowerDEVS: a tool for hybrid system modeling and real-time simulation. *Simulation*, 87(1-2):113–132, 2011.

[2] Laurent Capocchi, Jean François Santucci, Bastien Poggi, and Celine Nicolai. DEVSimPy: A collaborative python software for modeling and simulation of DEVS systems. In *2011 20th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 170–175. IEEE, 2011.

[3] Chiril Chidisiuc and Gabriel A Wainer. CD++ Builder: an eclipse-based IDE for DEVS modeling. In *Proceedings of the 2007 spring simulation multiconference-Volume 2*, pages 235–240. Society for Computer Simulation International, 2007.

[4] Michaël Deckers. DEVS: usability analysis of existing tools and library research. Technical report, University of Antwerp, 2016.

[5] Rhys Goldstein, Simon Breslav, and Azam Khan. DesignDEVS: Reinforcing theoretical principles in a practical and lightweight simulation environment, 2016.

[6] Sungung Kim, Hessam S Sarjoughian, and Vignesh Elamvazhuthi. DEVS-suite: a simulator supporting visual experimentation design and behavior monitoring. In *Proceedings of the 2009 Spring Simulation Multiconference*, page 161. Society for Computer Simulation International, 2009.

[7] Donald E. Knuth. Computer Programming as an Art. *Communications of the ACM*, 17(12):667–673, December 1974.

[8] Theo Mandel. *The elements of user interface design*, volume 20. Wiley New York, 1997.

[9] MathWorks. *SimEvents User's Guide*. The MathWorks, Inc., 2005.

[10] Gauthier Quesnel, Raphaël Duboz, and Éric Ramat. The Virtual Laboratory Environment – An operational framework for multi-modelling, simulation and analysis of complex dynamical systems. *Simulation Modelling Practice and Theory*, 17: 641–653, April 2009.

[11] 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.

[12] Eugene Syriani, Hans Vangheluwe, Raphael Mannadiar, Conner Hansen, Simon Van Mierlo, and Hüseyin Ergin. AToMPM: A web-based modeling environment. In *Demos/Posters/StudentResearch@ MoDELS*, pages 21–25. Citeseer, 2013.

[13] Imagine That. *Extend: User's Manual for Extend, Version 5*. Imagine That, 2000.

[14] Simon Van Mierlo, Yentl Van Tendeloo, Sadaf Mustafiz, and Bruno Barroca. Debugging parallel DEVS. Technical report, November 2014. URL http://msdl.cs.mcgill.ca/people/simonvm/devsdebuggerreport.pdf.

[15] Yentl Van Tendeloo. Activity-aware DEVS simulation. Master's thesis, University of Antwerp, 2014.

[16] Hans Vangheluwe. The Discrete EVent System specification (DEVS) formalism. *Course Notes, Course: Modeling and Simulation (COMP522A), McGill University, Montreal Canada*, 2001.

[17] Bernard P Zeigler, Herbert Praehofer, and Tag Gon Kim. *Theory of modeling and simulation: integrating discrete event and continuous complex dynamic systems*. Academic press, 2000.

DECLARATION

I declare that this thesis is a presentation of my own original research work and that it has never been submitted before for any degree or examination. Whenever contributions of others or external sources are involved, every effort is made to indicate and reference this clearly.

Belgium, 2016

Deckers Michaël, June 10, 2016