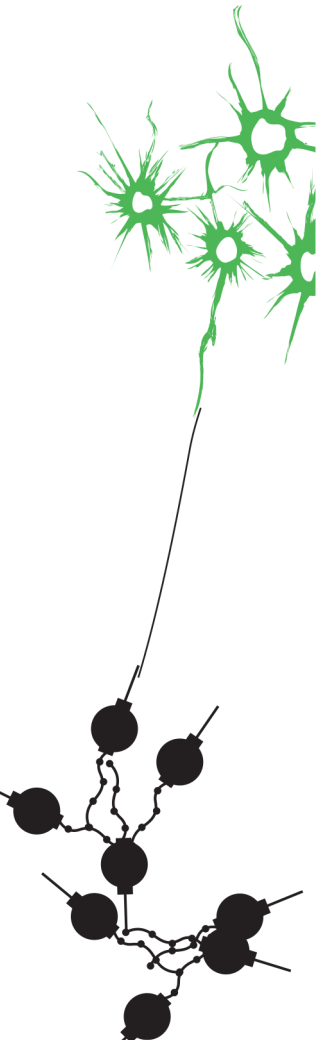

RAM ROBOTICS AND MECHATRONICS

AUTOMATED TESTING OF MODELS OF CYBER-PHYSICAL SYSTEMS

B. (Bas) Jansen

MSC ASSIGNMENT

Committee:
dr. ir. J.F. Broenink
T.G. Broenink, MSc
dr. A. Hartmanns

October, 2019

042RaM2019
Robotics and Mechatronics
EEMathCS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

UNIVERSITY OF TWENTE. | TECHMED CENTRE UNIVERSITY OF TWENTE. | DIGITAL SOCIETY INSTITUTE

Summary

With cyber-physical systems becoming more prevalent in critical environments, the testing of such systems becomes more important as well. This is a challenge due to the complexity, multidisciplinary nature and unique purpose of a cyber-physical system. In this research, a tool has been developed that is capable of testing a model of a cyber-physical system in an automated way, inspired by existing software testing techniques. With this tool, a user can define tests through a newly developed, Gherkin-style inspired testing language. This language is designed with reusability in mind to reduce repetitive writing. In this testing language, a user can define the temporal behavior of a cyber-physical system through a combination of LTL formulas, boolean equations, mathematical functions and model variables. Through simulation of a model of a cyber-physical system, the tool obtains testing data with which it can process test definitions and draw a **True** or **False** conclusion. The final product is modular to allow for switching out components and simulators to match the user's preferences. Multiple different simulators are supported so that the user can use the simulator fitting for their project.

Samenvatting

Cyber-physical systems worden steeds meer gebruikt in kritieke omgevingen. Daardoor wordt het testen van deze systemen ook steeds belangrijker. Dit is een uitdaging vanwege de complexiteit, het multidisciplinaire karakter en unieke doel van een cyber-physical system. Met inspiratie opgedaan uit bestaande software testtechnieken, is in dit onderzoek een tool ontworpen waarmee modellen van cyber-physical systems automatisch doorgetest kunnen worden. Met deze tool kan een gebruiker tests definiëren door middel van een nieuw ontwikkelde testtaal die gebaseerd is op de bestaande Gherkin-stijl. Om repetitief schrijven te verminderen is deze taal zo ontworpen dat (delen van) test definities herbruikbaar zijn. Met deze testtaal kan een gebruiker de temporal behavior van een cyber-physical system beschrijven door middel van LTL formules, boolean vergelijkingen, wiskundige functies en modelvariabelen. Door middel van een simulatie van een cyber-physical system kan de tool test data verkrijgen waarmee het een **True** of **False** conclusie kan trekken over de gegeven test definitie. Het uiteindelijke product is modulair, zodat de gebruiker gemakkelijk componenten en simulators kan verwisselen naar hun voorkeur. Meerdere verschillende simulatoren zijn bruikbaar, zodat een gebruiker de passende simulator voor hun project kan gebruiken.

Preface

A thank you to all who helped me from learning to write, to writing this Master thesis.

Contents

1 Introduction

1.1 Context

Cyber-physical systems are multi-disciplinary systems consisting of mechanical and electrical aspects often controlled by software. They have been integrated into aerospace, industrial manufacturing, transportation, and critical infrastructure, according to Rajkumar et al., 2010. Grimm et al., 2014 describes cyber-physical systems as a system linked directly with the physical world. They detect environmental changes and realize the real-time control of the system behavior. Compared with other traditional complex systems (e.g. mechatronic systems), the environmental influences (i.e. variances and uncertainties in the environment or created by human beings) play a significant role in cyber-physical systems. From the computation perspective, cyber-physical systems are considered to be developed based on embedded systems. But due to their link to the physical world, cyber-physical systems become much more complex than ordinary embedded systems, according to Lee and Seshia, 2015. According to Yagan et al., 2012, placing these complex systems in a critical environment can lead to catastrophic results if not tested correctly. This calls for a reliable and thorough way a cyber-physical system can be tested. This is a challenge due to the complexity, multidisciplinary nature and unique purpose of a cyber-physical system.

Software testing techniques like the ones described in Sawant et al., 2012, can form a starting point for this research to find a way to streamline and automate the testing of cyber-physical systems. Most software testing techniques have one thing in common. They produce a **True** or **False** answer based on set constraints that the software at hand adheres to or not. Software can be tested in controlled, isolated environments which is not the case with cyber-physical systems due to environmental influences playing a large role in the functionality of such a system. Influences such as warm-up times of motors, wear & tear and natural effects such as wind and water all can have their influence on how the system functions. As such, their performance can change depending on the situation or environment, which is important to be tested.

A simulator is capable of providing a controlled, digital environment for a cyber-physical system to be tested in. Zander, 2013 and Saglietti et al., 2015 use Simulink to achieve a simulation cycle and simulation state retrieval during the implementation of a system model. They track the approximate error to improve the performance and reliability of the system and evaluate the efficiency level of design execution. This shows the potential a simulator has to improve the testing phase of a cyber-physical system. Because a simulator is in full control of an environment, multiple test scenarios can be executed regardless of any dangers, such as collisions or unforeseen behaviors. On top of that, properties such as path tracking and pose accuracy are a lot easier to track as a simulator can constantly read out values of each part of the model. This allows for tests on submodels that would otherwise be inaccessible, hard to measure or just very costly to test. Even though simulators come with a larger initial investment in time in the form of creating a realistic model, it is believed that this cost is saved due to the failures a simulator can prevent. The better the quality of the model, the more failures can be prevented.

Although it is possible to digitalize a cyber-physical system, software testing techniques cannot be applied one-to-one. Zhou et al., 2018 gives insight into the testing of cyber-physical systems. Unlike software testing, tests for cyber-physical systems are typically behavior-based, with requirements working with margins and not exact values to indicate correct behavior. **True** or **False** answers can be obtained by checking whether values are in a margin or not. But since a

cyber-physical system is a continuous system that changes over time, having just one **True** or **False** as an answer to if the system once was within the required range during its run time will not suffice. If a robot performing surgery was within its safe margin for only a brief moment, its behavior can still have fatal results. This research, therefore, extends the testing of these ranges with a technique already existing within software engineering called Linear Temporal Logic, described by Schmaltz, 2017. LTL allows for test definitions over time, of which a more extensive explanation can be found in Appendix A.

1.2 Design Objectives

This research streamlines the testing phase of cyber-physical systems by providing a tool capable of thoroughly testing the temporal behavior of a cyber-physical system based on existing software testing techniques. A user can provide a test definition to the tool detailing the tests that need to be performed. A simulation of the cyber-physical system is used to perform this test on instead of having to run a test on the physical setup, reducing failure consequences and costs. By processing simulation data the tool can draw test conclusions. It is believed that by automating these steps through this tool, less time is lost to manually performing tests and comparing test data.

The tool is aimed to be used by multidisciplinary teams. Making the tests reusable allows team members to share their tests among each other, or makes it possible to combine smaller tests into one large test. Since there are types of tests, setups, and simulators, it is required that the tool is modular. Allowing users to modify the tool to their own needs by switching out components or simulators makes the tool more versatile and independent of the simulator, model or test at hand.

1.3 Approach

This tool will consist of three base concepts. First, it needs a way to define tests that apply to cyber-physical systems. This research uses LTL to test the temporal behavior of a cyber-physical system. By designing a new language for the definition of these tests, flexibility is provided to the user as they can define their specific tests by using the building blocks provided by this language. The user can define these tests in a Gherkin-style inspired manner, as described in Cucumber, 2019, in which the user can define settings for simulation and LTL formulas.

A controlled, digital environment is needed to thoroughly test cyber-physical systems in. To provide such an environment, the tool is extended with functionality to communicate with a simulator. 20-sim is a prevalent simulator used at the department of RaM, but for the sake of modularity and flexibility, interfacing will be extended to multiple simulators to allow each user to use their preferred simulator. Examples of such simulators are Simulink, V-REP, ARGoS, and Gazebo, which are compared and described in Pitonakova et al., 2018.

To turn these test definitions into a simulation performed by a selected simulator, a third component is required. One that can read out and interpret provided tests and take control of the simulator. It is also tasked with providing a **True** or **False** conclusion to the test definitions after simulation.

The validation of the tool is done through testing a set of tests that each test a piece of functionality in the system. The results outputted by the tool are compared to results obtained when manually performing the test, to check if the tool is correct in its calculations.

Further detail of these components is provided in Chapter 3.

1.4 Outline

This thesis is built up in the following way:

- An analysis of the problem at hand is made in Chapter 2
- The final design and design choices are explained in Chapter 3
- Obtained results are shown in Chapter 4
- Conclusions are drawn and recommendations are given in Chapter 5

2 Analysis

The structure of this tool can be categorized into three parts: A way to define tests, a simulator and a component that can interpret and process tests, as was described in Section 1.3. This chapter analyzes these three to lay a base for the final design of the tool.

2.1 Testing Guidelines

To find inspiration for testing definitions, this section looks into existing guidelines and testing techniques for cyber-physical systems. Jeswiet and Helferty, 1995, Kumičáková et al., 2016 and RoboDK, 2019 state that ISO 9283 and ANSI/RIA R15.06-2012 are the most prevalent standards set for testing manipulating industrial robots. ISO 9283 outlines robot performance tests which are primarily intended for developing and verifying individual robot specifications but can also be used for such purposes as prototype testing, type testing or acceptance testing. The American National Standards Institute developed the ANSI/RIA R15.06-2012 standard together with the Robotic Industries Association which provides a guideline aimed towards safeguarding personnel and devices by describing training processes, safety procedures built into robots during the manufacturing process and training guidelines.

Despite there being many guidelines to test cyber-physical systems and their operations, they usually consist of large, precise ways of testing. Not only would it mean that it would take a lot of time to design a tool capable of running all these tests, it would also make the tool inflexible. The user will have to depend on the developer's interpretation of these tests instead of their own. Therefore, instead of choosing to design the tool to perform exactly one set of tests, it is believed to be better to provide building blocks with which the user can build tests themself. Using LTL formulas and boolean equations involving variables of a simulation model, the user can define tests specifically for their model. That way, the user can be inspired by guidelines rather than be bound by them.

2.2 Simulators

As was described in Section 1.1, a simulator can provide a safe environment for the tool to run tests in. To allow for as many simulators to be compatible with the tool as possible, only a few necessary requirements should be set. In order to actually run a test and process data, a simulator should at least be capable of tracking data during a simulation it runs. Popular simulators for cyber-physical systems are 20-sim, Gazebo, V-REP, Simulink, ARGoS, Webots, OpenRAVE, Simspark and RoboDK.

Simulator	**External Interface**
20-sim	Python API
Gazebo	ROS & C++
V-REP	ROS & RemoteAPI
Simulink	MATLAB
ARGoS	Lua & C++
Webots	C/C++, Java, Python, MATLAB, TCP/IP
OpenRAVE	Octave, MATLAB, Python, Perl
Simspark	Network Protocol
RoboDK	Python, TCP

Table 2.1: Simulators and their external interfaces

Table 2.1 shows that all these simulators meet the requirement of having an external interface. Each simulator will require a specific tool-to-simulator interface, as each simulator has their own types of interfacing.

Test definitions can be extended with extra functionality to allow for more simulation control. Simulator-specific extensions can be implemented to make full use of a simulator that is capable of more than just the basic requirements. The more of such functionality can be used and controlled by the user through a test definition, the less manual work is needed to be done.

This research uses 20-sim and V-REP to show that it is possible to make use of multiple simulators with different types of interfacing.

2.3 Gherkin

The test definition consists of two parts, one that describes settings for the simulator and one that describes the temporal behavior to be tested. To pack these settings and behaviors into one test definition, Gherkin-style inspired definitions are used. Gherkin-style tests are used to define tests in a certain environment with three keywords; **Given**, **When**, **Then**. A test definition is structured as follows: **Given** a certain circumstance, **When** something occurs, **Then** the following action needs to happen. Due to the simplicity of Gherkin, its style has been adapted for this research. The **Given** keyword can be used to describe the simulation settings. Since the processing happens after simulation, the **When** keyword would always come down to **When** a simulation has occurred. This adds nothing, so this keyword is omitted. The **Then** keyword is used to describe the temporal behavior that will be tested. The new test structure is then as follows: **Given** model X **Then** proposition Y needs to hold. This **Given** keyword can be extended to allow for optional, simulator-specific settings. Gherkin also features the keywords **Feature** and **Scenario**, which can be used to label tests and run multiple test scenarios in one test, respectively. It is believed that defining tests this way makes them readable even for people that do not have programming experience, making tests more easy to reuse among team members.

2.4 Processing

As was brought up in Section 1.3, a component is required that can interpret tests and process their outcomes. Therefore there will be two processing steps in the tool occurring before and after simulation. Before the simulation, the tool must interpret the tests defined by the user and then communicate these through to the correct simulator. After the simulation, another processing step is required for combining the simulation data and LTL formulas to return answers to the user. These processing steps need to happen in a scalable, dynamic way to allow the user to define varied tests of variable size. A platform with the following properties is required to handle these processing steps:

- Scalable and dynamic.
- Has building blocks for the user to define their tests with.
- Can handle Gherkin-style inspired test definitions.

Due to how specific these requirements are, no existing library could be found. Therefore, a new testing language is designed. One that is dynamic enough to allow the user to extend tests by as many constraints and options as they want, but strict enough that the test definitions conform Gherkin and have no ambiguity. After the simulation, the tool can reuse the original interpretation of the test definition to combine it with obtained simulation data to fill out variables and draw test conclusions.

2.5 Requirements

To sum it up, the tool is required to consist of a way to define tests through a new language. Based on simulation settings defined in the test definition, a simulation is run. The simulation data and behavior defined in the test are combined after the simulation to produce an answer back to the user. This behavior is defined through the use of LTL formulas, boolean equations, and mathematical functions to test whether model variables are within correct margins over time.

This leads to the following list of requirements:

- The tool must combine simulation data with the defined tests to produce answers.
- The tool must be modular.
- The testing must be automated.
- The tests must be reusable.
- The test definitions must be Gherkin-style inspired.
- The test definitions must support boolean equations.
- The test definitions must support LTL formulas.
- The test definitions must support model variables.
- The tool should support multiple simulators.

The next chapter will go into more detail about how the tool is designed to meet these requirements.

3 Design & Implementation

3.1 Introduction

The functional design depicted in Figure 3.1 features the main three design objectives. A way to define tests, a simulator, and a component that can interpret test definitions, control the simulator and process outcomes. These three tasks have been split up in their own components named *Interpreter*, *Simulation Control* and *Post-processing* to improve modularity of the system.

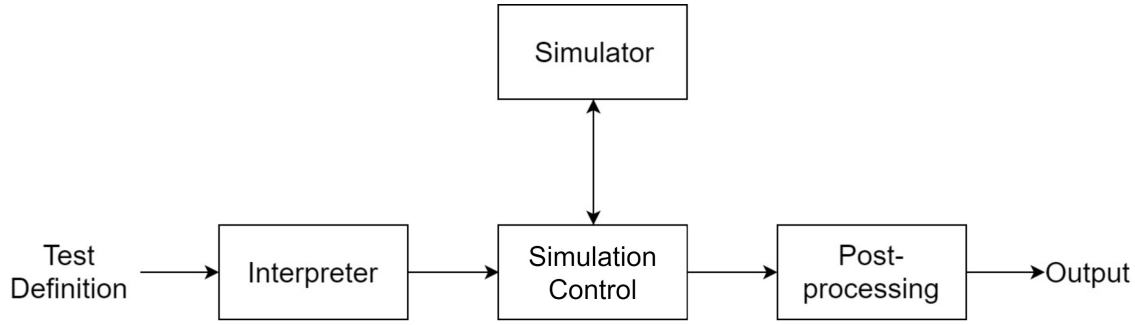

Figure 3.1: Functional design of the tool.

First, a design is made for the definition of a test. The structure of such a test has an influence on the way the tool should interpret it and in turn, also influences how the simulation is run. The designed structure of a test definition is shown in Listing 3.1.

```
Feature "feature name"
Scenario "scenario name 1"
    Given "model name"
        Settings X,Y,Z
        Then constraint
        Then constraint
          ...

Scenario "scenario name 2"
    ...
```

Listing 3.1: Basic test structure.

Where the keywords **Feature**, **Scenario**, **Given** and **Then** reflect the Gherkin-style inspiration described in Section 2.3. The **Given** keyword reflects the part of the test definition where simulation settings can be set for a **Scenario**. The **Then** keyword reflects the part of the test definition where the user can define temporal behavior through constraints. Constraints are LTL formulas and boolean equations applied to variables in the model used for the test.

For the tool to interpret this test definition, a new testing language will be designed, as was concluded in Chapter 2. The structure of a test determines the rules for the grammar that will define the new testing language. From this grammar, a parser can be derived using ANTLR. A parser can build a parse tree from a user's input test definition based on the grammar. Such a parse tree can be interpreted by software and through the use of a listener, data such as simulation settings and constraints can be extracted from it. A full explanation of how grammars, parsers, parse trees, listeners and ANTLR work is given in Appendix B.

The Interpreter can then package the data in an understandable way for the Simulation Control component. With this data, the Simulation Control component can select the correct simulator and initialize settings provided by the user through the test definition. The selection of the correct simulator can be done by selecting the correct tool-to-simulator interface, through which the Simulation Control component can communicate with a simulator. After the simulation, the Simulation Control component will pass on the received simulation data to the post-processing component.

For the post-processor to combine the simulation data with the original test definition, it can reuse the parse tree created during the Interpreter stage. By making the post-processor a listener, it can traverse the original parse tree. The listener can perform calculations over a node's children and store the answers in the parse tree's node. By performing the correct operation, the listener can calculate answers to LTL formulas, boolean equations, and mathematical equations. Starting in the leaves and working the data up the tree eventually results in an answer to the test definition. A more elaborate example of this concept is described in Section B.3.1.

A way to define tests and display outcomes is still missing in the functional design of the tool. By adding a user interface component to the tool, the user will have a way to interact with it. Not only can such an interface serve as a display for answers, but it can also serve as a platform for the user to define tests in and run those tests from. Since test definitions are text files, the user interface could be extended with functionality such as saving, modifying or deleting these files, depending on what the user needs.

Splitting up the tool in multiple components to improve modularity can lead to the tool becoming more complex as well. Due to the pipeline structure of the tool, each component relies on inputs and outputs of other components. Without exactly knowing what a component can expect from other components, it becomes a complex task to switch them out. To solve this, a responsibility-based approach is taken. This means that each component in the tool gets its responsibility regarding what type of data they consume, what type of data they produce and what is expected of them to do with the data between consumption and production. When each component holds itself to these responsibilities, it will be easier to swap out components that have different implementations as long as the consumption and production responsibilities are upheld. To extend this, the fewer components a component has to communicate with, the fewer responsibilities a component has to uphold. For this reason, an extra component is added to the structure called Core. It only passes on data from one component to the next but could be extended with functionality such as data validation at a later stage. The final structure of the tool is depicted in Figure 3.2

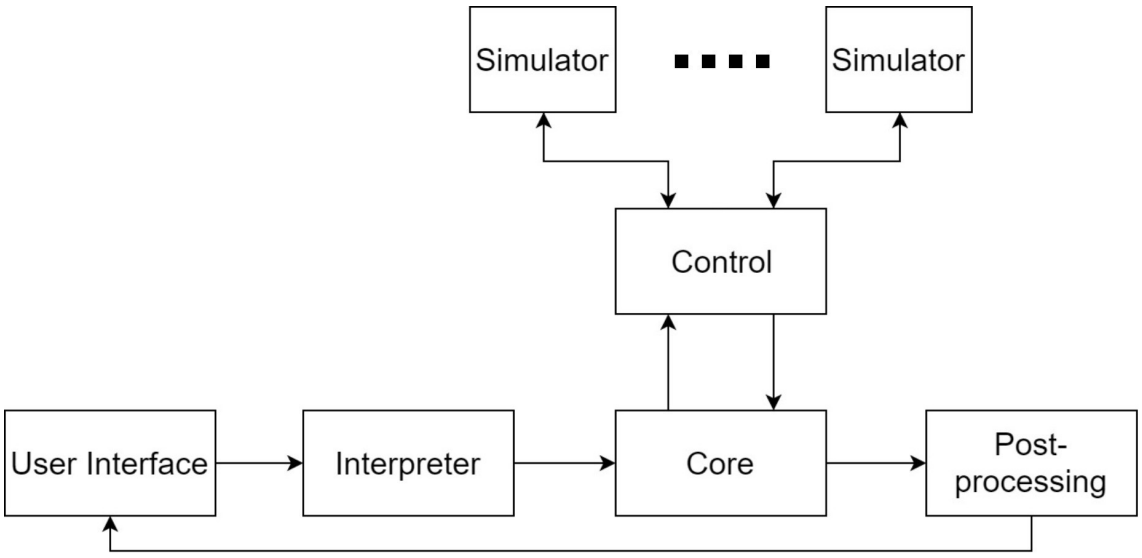

Figure 3.2: Structure of the tool.

3.2 Design Choices

In this section are the design choices made for each component in Figure 3.2 described, along with the description of a few implementations. The full description of the implementation of each component is written in Appendix D, along with the dataflow of the tool.

3.2.1 Interpreter

The Interpreter consists of two parts. One is the grammar dictating the rules to which a test definition needs to adhere, the second is the listener implemented to extract data from test definitions. First, the grammar is described.

Grammar

From the requirements can be deducted that the grammar needs to support LTL formulas and boolean equations that can be used to compare model variables over time, and the ability to retrieve settings for the simulation. The user can test values of the model by using model variable names. The inclusion of these variables is essential to allow users to compare and test values from the simulation.

On top of that are file inclusions and methods required to improve the reusability of tests. File inclusions added allow the user to make more fragmented tests where parts can be reused in other tests. Method definitions and calls have been added to allow a user to reuse a set of constraints at multiple locations inside the test.

For more thorough testing is more extensive mathematical functionality required. Multiplication, division, addition, subtraction, power, and square root allow the user to perform mathematical operations over numbers and variables. On top of that, mathematical functions such as *abs*, *min* and *max* are needed to further extend the tool's mathematical functionality. This functionality could be extended by allowing for three-dimensional values for orientations or positions, where currently only decimals are supported.

The Gherkin-based grammar then needs to support:

- LTL formulas.
- Boolean equations.
- Mathematical expressions such as power, square root, multiplication, division, addition and subtraction.
- Simulation settings.
- Inclusions of other test and constraint files.
- Functions such as minimum, maximum and absolute values.
- Methods.

The structure of a test definition supporting these requirements is depicted in Listing 3.2, serving as the base for the grammar.

```
Feature "feature name"

method name(argument1,argument2,argument3, ... ):
    constraint
 ...

Scenario "scenario name 1" (filler variable 1, filler variable 2, ... )
    Given "model name"
        with "option1" = X
        with "option2" = Y
         ...
        include "file 1"
        include "file 2"
         ...
        in simulator
        for X time unit
        Then constraint
        Then constraint
         ...

Scenario "scenario name 2"
    ...
```

Listing 3.2: Test structure.

Keywords such as **with**, **in** and **for** have been added to allow the user to get more control over the simulation. These keywords allow the user to set initial variables, a simulator, and for how long the simulation should run, respectively. As such, the testing becomes more automated and the time spent manually setting values in the simulator is reduced. To increase the reusability of the test definition, **method** definitions, **filler variables** and the keyword **include** have been put in place.

The full grammar is defined in Appendix C, along with an in-depth description of each grammar rule. With a fully defined grammar, it is possible to generate a lexer, parser, and listener as is described in Appendix B. The next section describes how this listener can be used to interpret tests based off of this defined grammar.

Listeners

The Interpreter is tasked with interpreting test definitions as defined in the grammar. The Interpreter needs to handle file inclusions and extract simulation data such as the name of the model, which simulator is being used, etc. When the test arrives at the Interpreter before the handling of inclusions, not all data is present within the test definition. Simulation settings or whole scenarios can be included, so the first step that needs to be taken is resolving all these inclusions. For this, an extra listener called the pre-processor has been added. The Interpreter component now consists of two listeners: the pre-processor and the main processor. The pre-processor substitutes in all the data gathered from the file inclusions, before sending it through to the main processor. The pre-processor is currently not recursive, so inclusions cannot contain more inclusions. After the pre-processor is done it is assumed that all data is available for the main processor. The main processor then retrieves all simulation settings from the **Given** parts of the test definition. From the **Then** parts it extracts all variables that need to be tracked during the simulation. With the introduction of methods, the main processor has obtained an extra responsibility: retrieving information about the methods and calls in a test. The main processor needs to identify the location of each method call, along with the name of the method called and arguments passed along. This is done so that after simulation, the results of a method call can be filled out in the correct place.

3.2.2 Method Handling

Before the post-processor can get to work, all method calls must have been handled. That way, the post-processor only needs to go over the Interpreter's parse tree and fill out the missing data. A listener can only enter and exit a node in the parse tree once. But since a method can be called multiple times, the listener is required to go over one method multiple times. Instead of letting the listener go over the entire parse tree, the subtree of a method is taken out. For each method call, the correct variables and arguments are set in the subtree and the answer is calculated. These answers are stored so that the post-processor can then retrieve them when traversing the entire parse tree after all method calls have been calculated. By filling the answers in, the post-processor only has to go over each node once, solving the issue of having enter or exit a node multiple times. A more in-depth explanation of the method handling is given in Section D.2.

3.2.3 Post-processing

The post-processor is tasked with calculating the answers to a given test definition. This processor can calculate answers to:

- LTL Formulas
- Boolean Equations
- Mathematical Equations
- Mathematical Functions

The calculation of these answers is done by reusing the parse tree created by the Interpreter. Since the test definition did not change during the simulation, the parse tree is still applicable. With the simulation data available, the post-processor can fill in the variables and start with the calculation. The calculation of a test definition happens in four phases. At first, all mathematical equations and functions are calculated so that they are turned into just one value. It is then possible to compare two values to obtain a **True** or **False**. Since LTL formulas are timebound and boolean equations are not, having just a **True** or **False** is not enough to perform LTL formulas on. On top of that, LTL formulas work with discrete states, whereas

cyber-physical systems are continuous.

Since a simulator is software, it is inherently discrete. This means that a simulator progresses through timesteps, rather than a continuous flow of time. Each time the simulator steps, it calculates all data for that moment in time and repeats this until the simulation is complete. By retrieving variable data along with the time-stamp on each simulation step, it is possible to create a time series of the data. The smaller the time interval between each step is, the more accurate the resulting data is. Such time-stamped data can be visualized by a graph. An example graph for an example simulation output is shown in Figure 3.3.

Figure 3.3: Example output graph.

Say that the LTL formula that needs to be performed on this output is **F**($output >= 2$). First, each point in the graph is checked for whether it is greater than two or not. For this graph it can be said that $output >= 2$ is **False** from $t = 0$ until $t = 4$ and **True** from $t = 4$ up to and including $t = 6$, where the simulation ends. LTL formulas can be performed over such timestamped booleans. **F**($output > 2$) is then **True**, since $output > 2$ is eventually **True**, namely from $t = 4$ to $t = 6$. The exact implementation of each mathematical, boolean and LTL operator can be found in Section D.1.

Output If a boolean equation or LTL formula is only **True** for part of the simulation, then the post-processor returns **Sometimes True** as an answer. The top-level constraint always only returns **True** or **False**. If the top-level constraint is an LTL formula, it is checked if it holds from $t = 0$, resulting in either **True** or **False**. If a constraint only consists of boolean equations then the top-level constraint is checked with a **G** operator. Since no time is bound to a boolean equation it must either always be **True** or always be **False**. A **Scenario** is **True** if all top-level constraints defined in it are **True**, otherwise it is **False**. A **Feature** is **True** if all **Scenarios** in it are **True**, otherwise it is **False**.

These results are forwarded to the User Interface, which can display these for the user to see. The next section details the design of the User Interface.

3.2.4 User Interface

The User Interface needs to provide a platform for the user to have control over what simulations are run with which tests. The User Interface in itself is a rather self-contained component, as the only thing it needs to communicate to the rest of the system is what tests need to be run. Options such as creating, modifying and deleting tests all happen outside of the rest of the tool. That way, it provides the user with the freedom to design the User Interface in a way that is most applicable to the situation the tool is used in, as long as it can tell the Core component to run a test.

The implemented User Interface has a few examples of additional features. One of them is output filtering. The user can set a certain output level which indicates how verbose the output of the tests will be. One user may be interested in just the final result of a test, where another would like to see in-depth info, returning the result of every layer of a test. This could at a later stage be extended with more information such as graphs, tables or other formats to improve the use of the data returned by the tool.

By giving the user the possibility to preset paths, repetition in writing tests is reduced. These paths point towards folders where for example models, tests and constraints are stored. This way the user does not have to type out the full path each time they want to refer to one. To display the versatility of the User Interface component, two interfaces were developed. One is a GUI allowing the user to create, modify, delete and run tests with, where the user has more visual feedback. The other is a command-line interface for small applications or batch programs, where the user can only run a test. Their implementations are detailed in Section D.3.

3.2.5 Core

All tasks described in the requirements are handled by the other components. This does not mean that the Core component has become obsolete. With the Core component in place, others will have to interact with at most two other components. That way, when a component is switched out, only the responsibilities towards the Core need to be upheld. The Core, therefore, improves the modularity of the entire tool. With a centralized Core, functions for ensuring the correctness of data could be put in place.

3.2.6 Simulation Control

With everything happening before and after the simulation in place, it is possible to look into what actions the Simulation Control component needs to handle. The Interpreter component is expected to extract the following data from a test:

- Model name.
- Simulator name.
- Variables to initialize.
- Runtime of the simulation.
- Variables that need to be tracked during the simulation.

The post-processor requires timestamped data to perform LTL formulas. The Simulation Control component needs to obtain this data from the simulation so it can be forwarded to the post-processor.

The Interpreter instructs the Simulation Control component on how to set the simulation before running it. The Interpreter also tells the Simulation Control component what the variables are that need to be tracked so the post-processor obtains all the timestamped data required for further processing. By making the Simulation Control component a separate block from the Simulator, it is possible to run multiple simulations on multiple simulators in quick succession. The current implementation runs all simulations sequentially independent of which simulator is used. An improvement here could be that when two tests are run on two separate simulators, the simulations are run in parallel. But before the Simulation Control can interact with a simulator, tool-to-simulator interfaces are required. These are described in the next section.

3.2.7 Simulator

The Simulator is in place to obtain timestamped data that can be used to process test definitions. In Section 2.2 it became clear that there are different simulators with different interfaces. Each simulator, therefore, requires their specific tool-to-simulator interface through which the Simulation Control component can reach the simulator. A simulator is compatible with the system if it can provide the following functionality through any means of interfacing:

- Opening a model.
- Tracking variables.
- Running a simulation.
- Providing data and timestamps.

Extra options can be set through the use of the **for** and **with** keywords to set the runtime of the simulation and initial variables, respectively. With simulator-specific tool-to-simulator interfaces in place, the Simulation Control component only needs to select the correct interface instead of communicating with each specific simulator by itself. The functions called are then translated by the selected tool-to-simulator interface to the simulator-specific commands, meaning that to add a new simulator, only a new tool-to-simulator interface needs to be added.

Another effect is that the programming language of a simulator's external interface does not have to be the same as that of the rest of the tool, as long as the translation occurs correctly. To showcase this, two simulators have been built into the tool. One being 20-sim, where the interfacing happens in Python. Another being V-REP, which has an external interface in Java. JNI is used to set up communication between the tool written in Python and V-REP's Java Interface.

With each component implemented, actual tests can be performed. Example tests are discussed in the next chapter to check if the tool meets the requirements set in Chapter 2.

4 Results

To test if the tool as expected, four test definitions have been developed. These test definitions showcase the following in the same order:

1. The basics of writing and running a test.
2. The reusability of tests.
3. The compatibility of the tool with other simulators.
4. The mathematical capabilities of the tool.

4.1 Basics Test

Starting off to show the basics of a test, the test definition depicted in Listing 4.1 is used as an example.

```
Feature "PWM Test"
    Scenario "PWM Check"
        Given "PWM Conversion.emx"
        in 20-sim
        with "PWM.f" = 10
        for 0.11 seconds
        Then G(("PWM.output" == 2 U
                ("PWM.output" == -2 U "PWM.output" == 2)))
```

Listing 4.1: PWM Check

Where a feature called "PWM Test" is tested that has only one **Scenario**, "PWM Check". In this scenario, a model that does PWM conversions is given. This model converts a sine wave to a PWM signal. The selected simulator is 20-sim and the PWM frequency is initialized to 10Hz. The simulation is set to run for 0.11 seconds. The constraint then states that it should globally hold that the PWM output is 2 until the PWM output is -2 until the PWM output is 2 again. During simulation is the PWM output captured from 20-sim as is visible in Figure 4.1. The red line denotes the PWM output. Here it can be seen that the red line starts at 2 and stays there until it goes down to -2. Then, it goes back up to 2 again where it stays until the end.

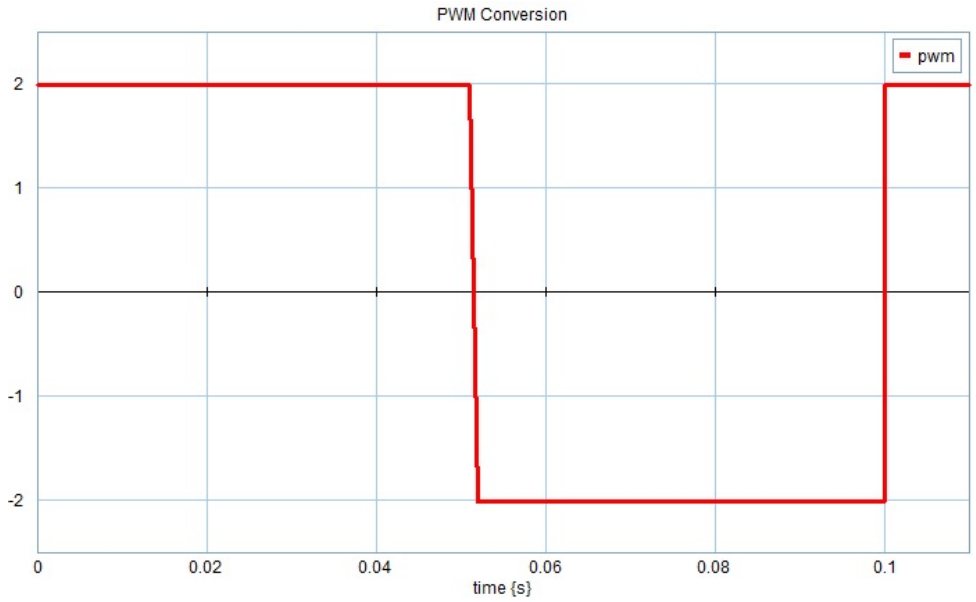

Figure 4.1: 20-sim graph of the PWM output.

The result can then be deducted as follows:

- **"PWM.output" == 2** : **True** in time ranges [0,0.05] and [0.1,0.11]
- **"PWM.output" == -2** : **True** in time ranges [0.05,0.1]
- **"PWM.output" == -2 U "PWM.output" == 2** : Is then **True** in time range [0.05,0.11]
- **"PWM.output" == 2 U**
  ("PWM.output" == -2 U "PWM.output" == 2) : Is then **True** in time range [0,0.11]
- **G(("PWM.output" == 2 U**
  ("PWM.output" == -2 U "PWM.output" == 2))) : Is **True** since the proposition holds from start to end.

So from this it can be expected that the tool returns "True" as the final result and "Sometimes True" for "PWM.output" == 2 and "PWM.output" == -2. The result is displayed in Figure 4.2, which is exactly as expected.

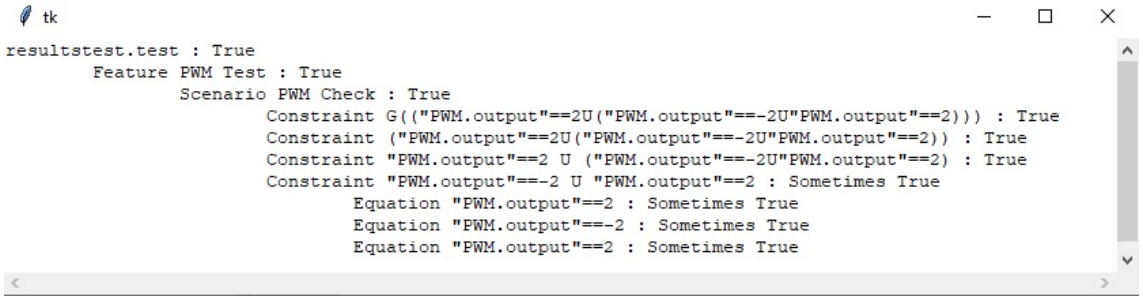

Figure 4.2: Tool output of the PWM Test.

4.2 Reusability Test

The next test is designed to showcase the reusability of a test definition. This test is defined in Listing 4.2. All functions implemented into the tool to improve the reusability of tests are integrated into this test definition. These functions are:

- Allowing multiple **Scenario** keywords in one **Feature**.
- Allowing multiple **Then** keywords in one **Scenario**.
- Importing a constraint from the constraint file described in Listing 4.3.
- Importing a scenario through the **import** keyword from the scenario2.test file depicted in Listing 4.4.
- Using a **filler variable** to fill out a variable in the second scenario.
- Importing test settings described in Listing 4.5 through the **include** keyword.
- Defining a **method** called range at the top that is called twice.

The pre-processor handles all the imports and substitutions and combines these three files into one large test definition, which is depicted in Listing 4.6.

```
Feature "Reusability"

method range(var,first,second):
  (%var > %first and %var < %second)

Scenario "Scen1"
    Given "double_slider_control.emx"
        include "./testSettings.test"
        Then "./then.constraint"
        Then range("Controller.SignalLimiter2.output",0,0.2)

import "./scenario2.test" (0.005)
```

Listing 4.2: Reusability Test

```
range("axis.position_real.output",-1,0.1)
```

Listing 4.3: then.constraint

```
Scenario "Scen2"
    Given "C:\Program Files (x86)\20-sim 4.7\Models\slider_bigmotor.emx"
    in 20-sim
        Then G (X 1 seconds
            ( abs("Carriage.position" - "Setpoint.amplitude") < &0 ))
```

Listing 4.4: scenario2.test

```
in 20-sim for 1 seconds
with "mode.C" = 1
with "setpoint.C" = 0.05
with "axis.position_real.initial" = 0
```

Listing 4.5: testSettings.test

```
Feature "Reusability"

method range(var,first,second):
    (%var > %first and %var < %second)

Scenario "Scen1"
    Given "C:\Program Files (x86)\20-sim 4.7\Models\double_slider_control.emx"
    in 20-sim
    for 1 seconds
    with "mode.C" = 1
    with "setpoint.C" = 0.05
    with "axis.position_real.initial" = 0
        Then range("axis.position_real.output",-1,0.1)
        Then range("Controller.voltage",0,0.2)

Scenario "Scen2"
    Given "C:\Program Files (x86)\20-sim 4.7\Models\slider_bigmotor.emx"
    with "Setpoint.amplitude" = 0.05
    in 20-sim
        Then G (X 1 seconds
            ( abs("Carriage.position" - "Setpoint.amplitude") < 0.005 ))
```

Listing 4.6: Reusability Test after substitutions.

The first **Scenario** of this test definition tests the controller of a setup consisting of two sliders. The graph in Figure 4.3 depicts the data of the variables "axis.position_real.output" and "Controller.voltage" after simulation.

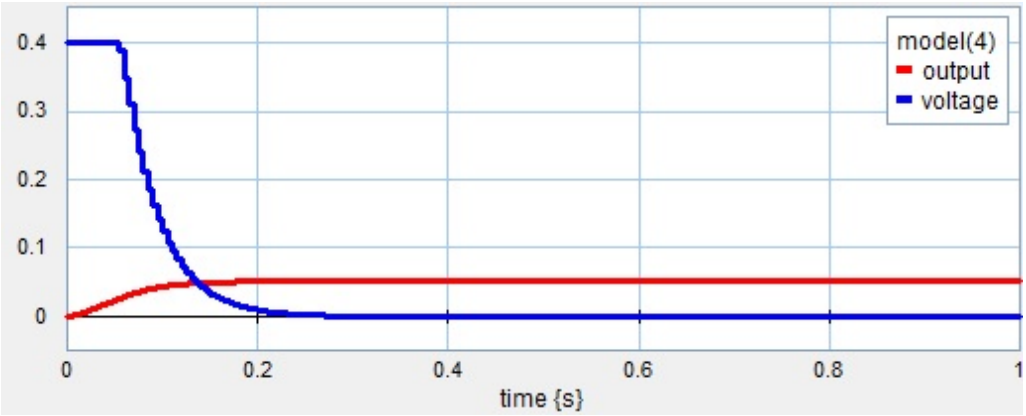

Figure 4.3: Graph of variable data of Scen1.

Where the "axis.position_real.output" value is within the range of -1 and 0.1 for the entirety of the simulation, so this is expected to be **True**. "Controller.voltage" is in the range of 0 and 0.2 a bit before $t = 0.1$ until the end of the simulation. This equation is expected to result in **Sometimes True**. Since this constraint only consists of boolean equations, this constraint will still result in **False** as the constraint does not hold globally. In turn, the **Scenario** is expected to fail.

The second **Scenario** tests the motor of a slider. The graph in Figure 4.4 depicts the data of the variables "Carriage.position" and "Setpoint.output".

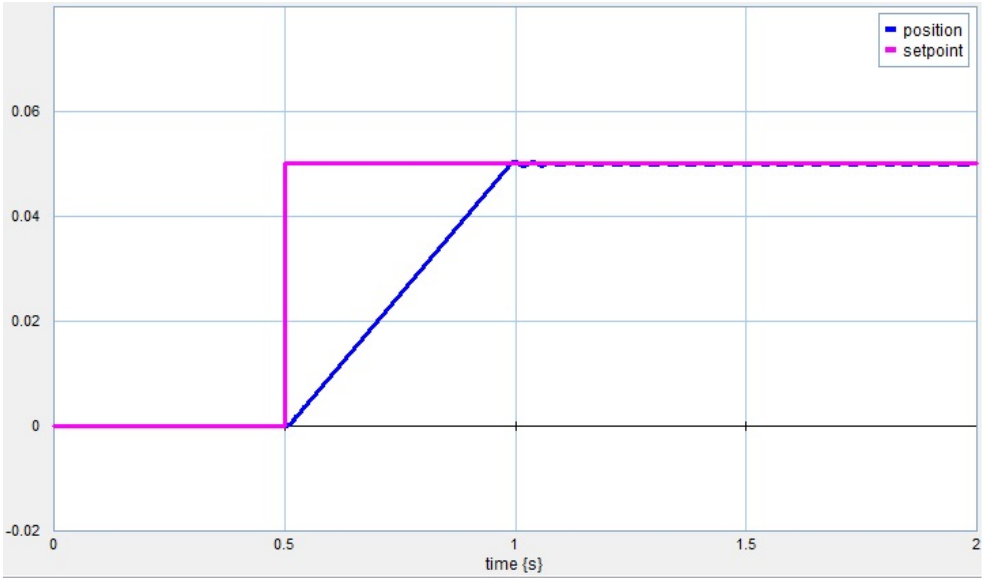

Figure 4.4: Graph of variable data of Scen2.

Here it is tested if it globally holds that in the next second, the absolute value of "Carriage.position" − "Setpoint.amplitude" is less than 0.005. In the graph can be seen that the position of the carriage is almost equal to the setpoint from a bit before the 1-second marker until the end of the simulation. The constraint is therefore expected to return **True**.

The result of the test is shown in Figure 4.5, which matches the expectations.

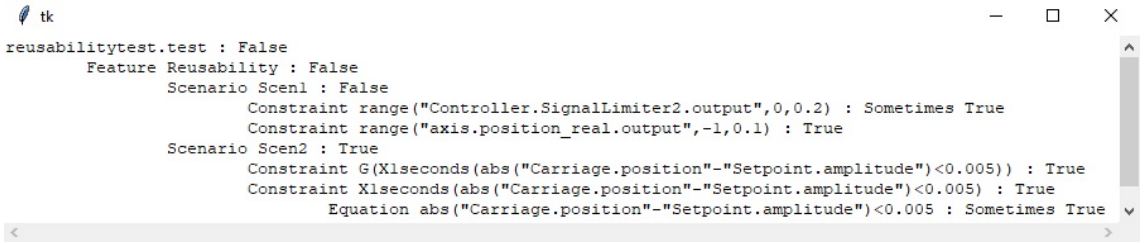

Figure 4.5: Tool output of the Reusability Test.

4.3 V-REP Test

To display that the tool supports multiple simulators, the test in Listing 4.7 is defined, where V-REP is used instead of 20-sim, like in the other tests. By selecting a different simulator, a different model type and different way of naming variables is required too. Where 20-sim models have the .emx file extension and make use of dots to show component hierarchy in the variable naming, V-REP uses .ttt files and more basic variable names.

The model depicted in Figure 4.6 is tested with the test definition described in Listing 4.7. It consists of a self-driving car that uses a visual sensor to detect obstacles. With the car walled in by obstacles, it should eventually detect one with its visual sensor. When it does, it should back up within the next 0.5 seconds.

```
Feature "VREP Test"
Scenario "VREP"
    Given "VREPTEST.ttt"
        in V-REP for 40 seconds
        Then F((F("resSig" < 0) and X 0.5 seconds ("backing" < 0)))
```

Listing 4.7: VREP Test

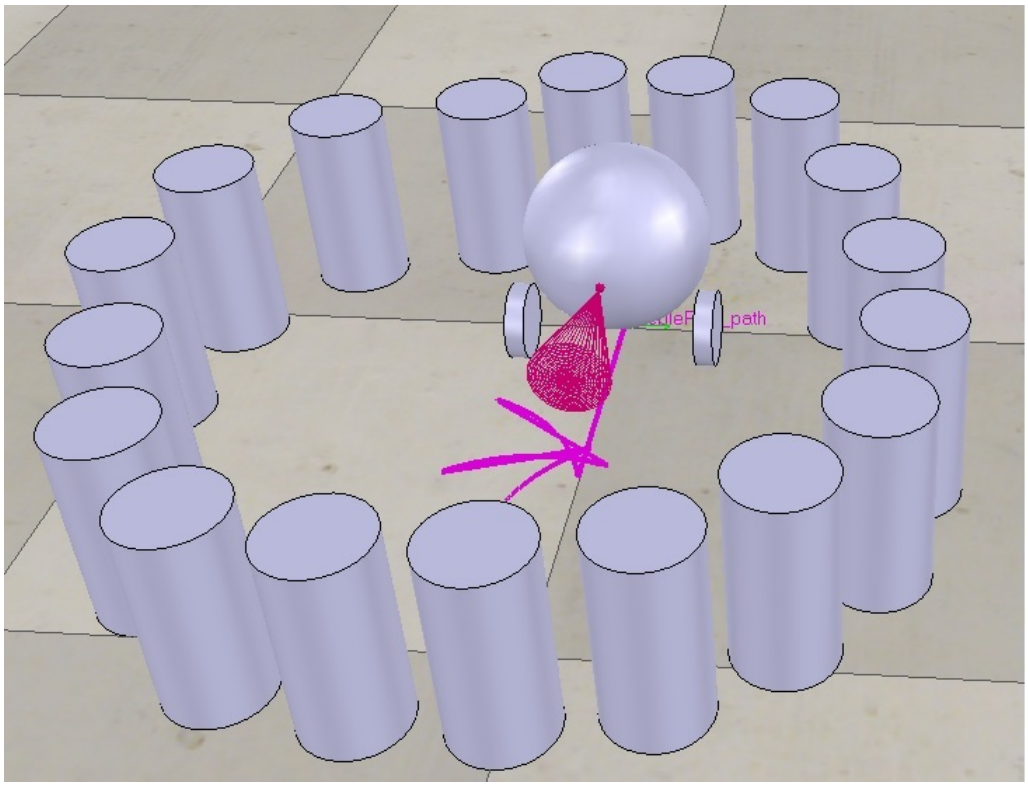

Figure 4.6: V-REP model.

Running the simulation results in the graph depicted in Figure 4.7. The spikes up to 1 show when the vision sensor detected an obstacle. The other signal is the velocity of the car. The car backs up almost instantly when the vision sensor detects an obstacle, so the result of this test is expected to be **True**.

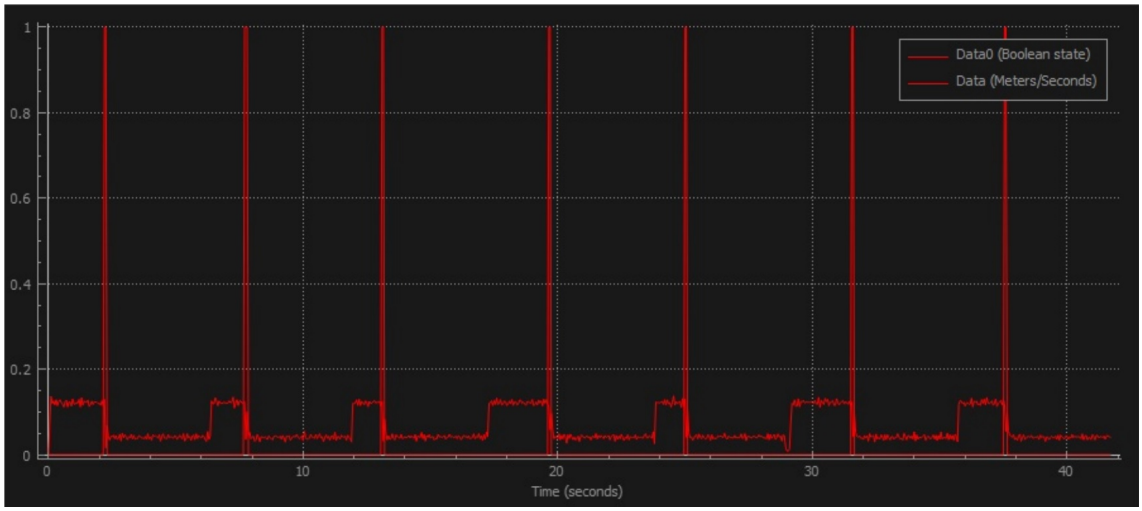

Figure 4.7: Graph of variable data of VREP Test.

The output of the tool is shown in Figure 4.8, showing the expected **True** as a result.

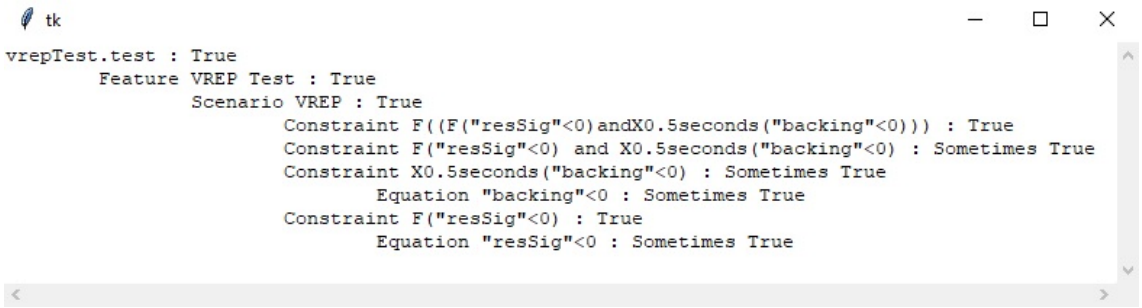

Figure 4.8: Tool output of the VREP Test.

4.4 Math Test

Lastly, the test defined in Listing 4.8 is in place to show the mathematical operations the system can perform. A controller's constant is used to compare the tool's answer.

```
Feature "Math Test"
Scenario "Math"
    Given "double_slider_control.emx"
        in 20-sim
      with "Controller.Constant.C" = 707281
      for 1 seconds
            Then "Controller.Constant.output" == (sqrt(5*(5^3))-3+7)^5/29
```

Listing 4.8: Math Test

The tool returns **True**, meaning it correctly calculated the answer to the equation, as can be seen in Figure 4.9.

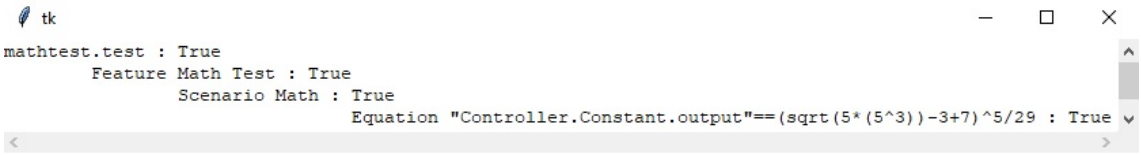

Figure 4.9: Math Test Result.

Conclusions will be drawn based on these test results in the next chapter.

5 Conclusions & Recommendations

5.1 Conclusions

Restating from Chapter 2, the tool must be able to combine simulation data with the defined tests to produce answers. This requirement is met through the implementation of the post-processor. It is shown that these answers are correct in Chapter 4. In the same chapter, the following requirements are met:

- The test definitions must be Gherkin-style inspired.
- The test definitions must support boolean equations.
- The test definitions must support LTL formulas.
- The test definitions must support model variables.
- The tool should support multiple simulators.
- The tests must be reusable.

It was shown how a user can define tests consisting of Gherkin-style inspired keywords allowing for the user to define behaviors through a combination of LTL formulas, boolean equations and model variables. Here it was also shown that it is possible to perform tests on multiple simulators. Another test in this chapter showed that reusability is in place through the use of **import**, **include** and **method** keywords, along with the possibility to use **filler variables** and the ability to import constraints from constraint files.

The modularity of the tool is reflected by how the tool

- Is separated into small blocks that have their tasks.
- Has a responsibility-based communication between components.
- Has simulator-specific interfaces through which the Simulation Control component can quickly select the correct simulator.
- Has no component other than the Core that needs to interface with more than two other components.

The automation of the tool is ensured through the fact that the user only needs to provide a test definition. Everything else gets handled automatically by the tool, requiring no other input. This was shown in Chapter 4, where the tool only required a test definition to produce results.

Therefore it can be concluded that all set requirements are met. But due to the intention to design the system in such a way that it can be modified to suit every user's needs, more features can be implemented beyond the current basis. The following sections look into future work.

5.2 Recommendations

It is believed that the greatest optimization that could be made within the system is the addition of parallel testing. If multiple simulators could run their tests at the same time, testing times can be greatly reduced.

Another extension could be made within the values that are possible to be used. Currently, only decimals are supported, but this could be extended to other data types such as 3D values for orientations and positions allowing for even more in-depth testing.

With different data types, it also becomes possible to display data differently. Due to this research having the aim to just have a list of **True** and **False** as a result of the tests, this was omitted. Extensions that could be made are for example display of graphs, tables or timelines to give a better idea to the user at what timestamps something is **True** or **False**.

Path handling currently is only done once in the pre-processing phase. This means it is not recursive, so when an imported file contains an import, this import is not handled anymore. This could be fixed by running the substitution of these paths repeatedly until no more paths are left. A better, more scalable solution would be one involving the listener to load in and substitute files.

Methods could become more extensive by including method overloading, recursion, storing variables and extra functionality that can make methods more useful.

Appendices

A LTL

Linear Temporal Logic is a temporal logic that allows one to reason about state properties in a Labelled Transition System by defining formulae that describe the future of paths. With it, it is possible to for example describe that eventually, something will hold **True**, or something always holds **False**. For the sake of simplicity, the only LTS that will be used in this research will be in the shape of a single one-directional path, where each state relates to a timestamp within the simulation. This is the advantage of the discrete timestamps that simulations use in comparison to the continuity of time in real life.

It is assumed that one run of a simulation cannot have multiple possible paths after simulation, as one is already taken. This taken path effectively becomes a timeline of moments where certain propositions do or do not hold. This could be compared to a digital signal that over time can be set to high or low depending on whether a proposition is **True** or **False**. An example timeline can be seen in Figure A.1 If the following states were to represent a timeline of t = 0, t = 1, t = 2 and t = 3, we can now say that:

- φ is **True** at t = 0, t = 1 and **False** at t = 2, t = 3.
- ψ is **True** at t = 2 and **False** at t = 0, t = 1 and t = 3.

Figure A.1: Example timeline.

So, it can be said that when looking at this timeline, starting from t = 0, eventually ψ will be **True**. To write this in a more concise and descriptive way, the syntax of LTL formulas is introduced, according to Schmaltz, 2017.

LTL makes use of four operators: **X**,**F**,**G**,**U**. For this research, **W** and **R** are also used. The **X**, *neXt*, operator denotes that from a given state, a proposition is supposed to hold in a state directly following it. **X**φ holds **True** in a state if proposition φ is **True** in a directly following state, as is depicted in Figure A.2.

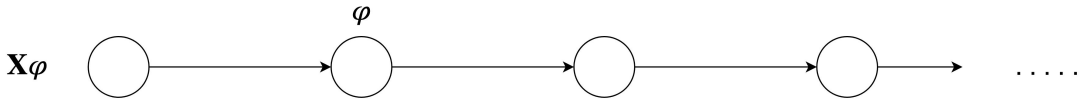

Figure A.2: **X**φ timeline which holds **True**.

The **U**, *Until*, operator denotes that a formula needs to hold **True** until another formula is **True**. For φ**U**ψ this means that at some point proposition ψ needs to hold and preceding to that point proposition φ needs to hold on every preceding state in the path. This also means that φ does not necessarily need to hold **True**. If ψ holds **True** in the first state, φ**U**ψ holds **True**. This is depicted in Figure A.3

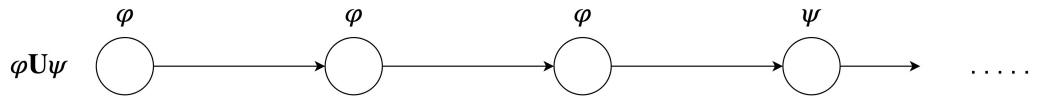

Figure A.3: $\varphi \mathbf{U} \psi$ timeline which holds **True**.

$\mathbf{F}\varphi$ means that *eventually* φ must hold in the future. This is depicted in Figure A.4

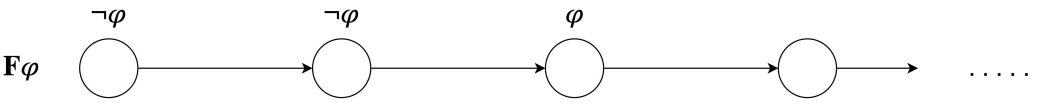

Figure A.4: $\mathbf{F}\varphi$ timeline which holds **True**.

The *Globally* operator is denoted as **G**. $\mathbf{G}\varphi$ means that for every state in the path, proposition φ needs to hold as is depicted in Figure A.5

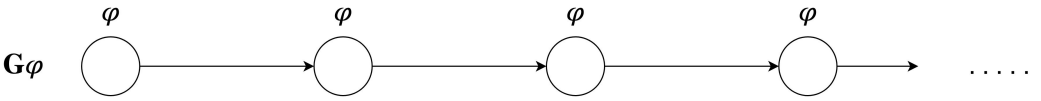

Figure A.5: $\mathbf{G}\varphi$ timeline which holds **True**.

The **W** and **R** operators are a variant on the **U** operator. φ **W** ψ denotes a *Weak until*. φ has to hold at least until ψ, which is the same as the **U** operator. An addition is that if ψ never becomes **True**, then φ must remain **True** forever. That way, it is a combination of the **U** and **G** operator.

The **R**, *Release*, operator is similar. With φ **R** ψ, ψ has to be **True** until and including the point where φ first becomes **True**; if φ never becomes **True**, ψ must remain **True** forever.

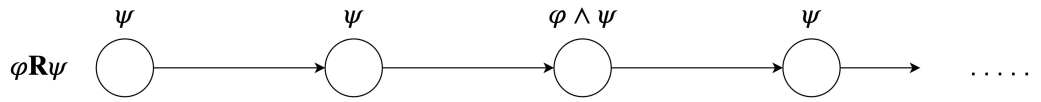

Figure A.6: $\varphi \mathbf{R} \psi$ timeline which holds **True**.

Due to the atomic propositions that can be defined within the LTL formulas, it is now possible to extend the basic boolean equations with these LTL formulas. LTL formulas can also be combined again with boolean equations.

With this, it is believed that a wide variety of tests can be implemented. Examples could be:

- $\mathbf{F}(ControllerError < 0.01)$, eventually the ControllerError is smaller than 0.01.
- $\mathbf{G}(ErrorState \neq Reached)$, globally it must hold that the ErrorState is not Reached.
- $MotorRunning == True$ **U** $CurrentPosition == DesiredPosition$, the motor must be running until the current position equals the desired position.
- $\mathbf{G}(Motor \neq Overheated) \wedge \mathbf{F}(Orientation == DesiredOrientation)$, globally it must hold that the motor is not overheated and eventually the orientation should be equal to the desired orientation.

B Grammars, Lexers, Parsers, Parse Trees, Listeners & ANTLR

B.1 Grammar

At the base of a parser lies a grammar. Just like how spoken languages have grammars dictating what correct sentences are, the same concept is used within parsing. Grammars are sets of rules consisting of two sides usually separated by a colon. One is the name of the rule on the left-hand side, the other side being a set of other rules or fragments that constitutes the rule on the left-hand side. For example, in English grammar, the rule for a correct sentence could be defined in the way of *sentence : subject verb object.* Then, if we were to parse the input *I like parsers* with this grammar, it would be seen as a correct sentence.

B.2 Lexer & Parser

The lexer is the first step required to parse an input according to a grammar. The lexer's job is to take in a user's input and attempt to turn them into tokens. A token usually consists of a token name followed by a token value. With again the example input "I like parsers", the lexer would return a list of tokens in the form of [(subject, I), (verb,like), (object,parsers)]. This has to be done so that the parser can easily match these tokens with the grammar rules to check if an input can be parsed correctly. The parser takes in this sequence of tokens and tries to build up a parse tree from these tokens.

B.3 Parse Trees

A parse tree is an ordered, rooted tree consisting of nodes and leaves. With the tokens provided to the parser, it will attempt to build such a tree by matching sequences of tokens with the right-hand sides of grammar rules. Upon finding a matching rule, it connects these tokens as the children of this rule, making this rule a node in the tree. This node now functions as another token within the sequence and can be used to combine with other tokens again. The parser repeats this process until it reaches the start rule. If the parser manages to produce a parse tree having the starting rule at the top of the tree, then that means that the parsed input is correct according to the grammar.

The following grammar is used as an example.

```
grammar example;
sentence : subject verb object;
subject : 'I' | 'You' | 'We';
verb : 'like' | 'dislike';
object: 'parsers' | 'lexers' | 'grammars';
```

Listing B.1: Example grammar.

The starting rule here is *sentence*, as it is the only rule that is never featured on the right-hand side of any other rule. If we let the parser parse the following test, we expect to get a parse tree having leaves that show the entirety of the input, and the top of the tree to be a node for the rule *sentence*. When parsing the sentence *I like parsers* with this grammar, the following Parse Tree is produced, as can be seen in Figure B.1.

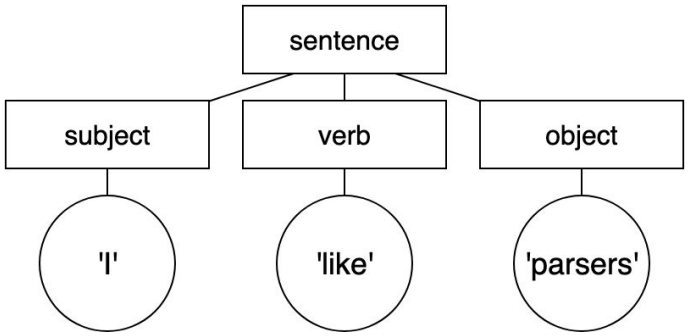

Figure B.1: Example parse tree.

Where in the leaves each character of the given input can be found and the *sentence* rule is at the top, so the input conforms the grammar rules.

B.3.1 Listeners

Each listener traverses a given parse tree. This is done by making use of a walker that calls functions within the listener based on a node or leaf it enters or exits. For example, when exiting the node "Expression", the exitExpression function will be called. Due to the structure of the grammar, this equation will have two children. The listener is informed about the children, its parent and other contextual information relevant for it through a ParserRuleContext object. An issue with this is that when it is attempted to get the children from this "Equation" node, all that is returned is the context of the child. This means it is possible to get the text stored in this node, but when the walker is at the Expression node, it does not scale well to have to delve deep into the children to find an eventual variable name or number. The solution to this is the use of a dictionary. This dictionary consists of the unique context of a node, stored with a value. This is used as follows, say the expression "5*8" is parsed and results in the following parse tree.

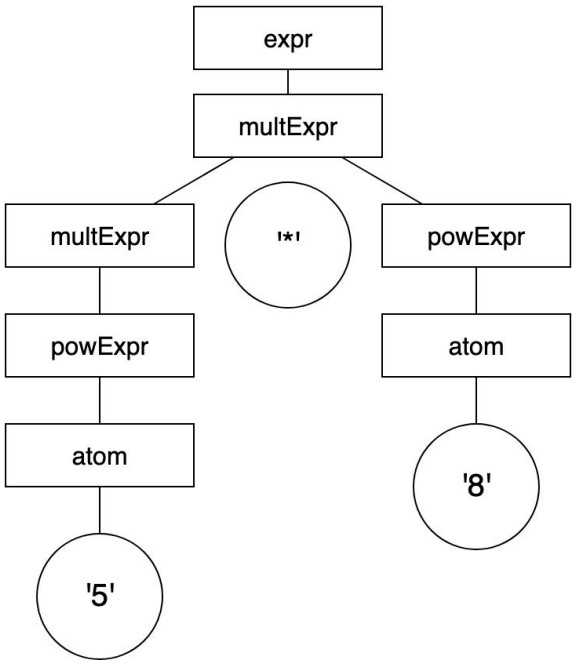

Figure B.2: Parse tree of the expression 5*8.

When walking over this tree in a depth-first search, the walker first enters the node *expr*. When the walker enters a node, it does not know of any values the children may have. Unless this

node influences the children, the enter function is skipped. Continuing to walk over the tree eventually causes the walker to reach the leaf with the number 5. From there, the walker starts with exiting the nodes. First, it exists the leaf with the number 5 in it, then it exits the atom node. Upon exiting this node, the exitAtom function is called. This function first establishes how many children it has and of what type, since, as can be seen in the grammar, the atom rule has seven subrules. Due to the implementation of the grammar, each of these subrules is unique, so the correct action for the correct subrule can be performed. In this case, its only child is of the type *NUMBER*. It stores its context along with the value obtained from its child, the number 5, in the dictionary. After this action, the walker continues and exits the powExpr, which repeats the same action, but instead of retrieving the value from its child, it retrieves its child's set value from the dictionary. In this case, this node does not need to do anything other than passing the value along to the top, as no power was parsed in this expression. These actions are repeated by every node until the multExpr node at the top is reached. It detects that it has three children. A multExpr, a * and a powExpr. Based on the middle child being a *, it knows that it has to perform a multiplication. It retrieves the values of the multExpr and powExpr from the dictionary and multiplies them with each other. The result of this multiplication is once more stored in the dictionary under this multExpr's context.

By using this concept throughout the entire tree, calculations can be performed in the middle of the tree from which the answers can be passed along so they can be used for other calculations again. By storing this data somewhere it can be retrieved by the Parse class too, whereas it could otherwise not reach anything stored in the parse trees. This concept is used in every listener and influenced the structure of the grammar a lot, as the structure of the grammar once again had an influence on if calculations would be performed correctly with the correct children.

B.4 ANTLR

To simplify the amount of work that is required a tool called ANTLR, Another Tool for Language Recognition is used. ANTLR can read in a grammar and derive lexers, parsers and template listeners from it. Using ANTLR only leaves the user with the tasks to define a correct grammar and eventually fill out the listeners based on what the user wants to do with the read input by walking over the built Parse Trees. ANTLR, in this case, is only an example of a parser that can be used, but the tool is not limited to ANTLR alone.

C Grammar

C.1 Grammar Definition

```
grammar Grammar;
entry       : feature scenario+;
feature     : 'Feature' NAME methods;
scenario    : 'Scenario' NAME
                (LPAR fVar (',' fVar)* RPAR)? given then+ |
              imprt (LPAR fVar (',' fVar)* RPAR)?;
imprt       : 'import' (PATH | NAME);
methods     : method*;
given       : 'Given' model settings*;
settings    : ('for' time) |
              ('with' initVars) |
              ('in' simulator) |
              ('include ' (PATH | NAME)) ;
time        : expr 'nanoseconds' |
              expr 'microseconds' |
              expr 'milliseconds' |
              expr 'seconds' |
              expr 'minutes' |
              expr 'hours';
initVars    : initVar (',' initVar)*;
initVar     : NAME '=' (expr | boolean);
boolean     : 'False' |
              'True' |
              'true' |
              'false';
simulator   : '20-sim' | 'V-REP';
then        : 'Then' constraint ;
model       : PATH | NAME;
fVar        : expr;
constraint  : LPAR constraint op constraint RPAR |
              tempSing LPAR constraint RPAR|
              tempDouble |
              CONSPATH |
              'not' constraint |
              constraint 'for' time |
              equation |
              methodCall;
tempDouble  : LPAR constraint tempDoub tempDouble RPAR |
              LPAR constraint tempDoub constraint RPAR;
func        : 'max' | 'min' | 'abs';
tempSing    : 'F'|'G'|'X' time |
              'eventually' | 'globally' | 'next' time;
tempDoub    : 'U'|'R'|'W' |
              'until' | 'release' | 'weakrelease';
op          : 'and' | 'or' | '->' | '<>';
equation    : cons EQ cons |
              cons LT cons |
              cons LTE cons |
              cons GT cons |
              cons GTE cons |
              cons NOT cons;
```

```
cons         : function |
               boolean |
               methodCall;
function     : func LPAR function RPAR |
               expr;
methodCall   : ID LPAR callArgs RPAR;
callArgs     : expr (',' expr)*;
method       : 'method' ID LPAR args RPAR ':' constraint;
args         : ID(','  ID)+;
fillVar      : '&' NUMBER;
functionVar  : '%' ID;
expr         : expr (PLUS | MINUS) multExpr |
               multExpr ;
multExpr     : multExpr (TIMES | DIV) powExpr |
               powExpr ;
powExpr      : powExpr POW atom | atom;
atom         : MINUS atom #negAtom |
               LPAR atom RPAR #parAtom |
               NUMBER #numAtom |
               LPAR expr RPAR #exprPar |
               fillVar #fVarAtom |
               functionVar #funcVariable |
               NAME #nameAtom;

LPAR         : '(';
RPAR         : ')';
PLUS         : '+';
MINUS        : '-';
DIV          : '/';
TIMES        : '*';
POW          : '^';
SQRT         : 'sqrt';
LT           : '<';
LTE          : '<=';
GT           : '>';
GTE          : '>=';
NOT          : '!=';
EQ           : '==';
QUOTES       : '"';
CONSPATH     : QUOTES ('a' ..'z' | 'A'..'Z' | '0'..'9' |
               '(' | ')' | '.' | '\\' | '/' | ':' | '-' |
               ' ' | '_' )+ '.constraint' QUOTES;
NAME         : QUOTES ('a' ..'z' | 'A'..'Z' | '0'..'9' |
               ' ' | '.' | '_')+ QUOTES;
PATH         : QUOTES ('a' ..'z' | 'A'..'Z' |
               '0'..'9' | '(' | ')' | '.' | '\\' |
               '/' | ':' | '-' | ' ' | '_' )+ QUOTES;
ID           : ('a'..'z' | 'A'..'Z')+;
NUMBER       : ('0'..'9')+ ('.' ('0'..'9')+)?;
COMMENT      : '/*' .*? '*/' -> skip;
LINE_COMMENT: '//' ~[\r\n]* -> skip;
WS           : [ \t\r\n]+ -> skip;
```

C.2 Grammar Rules

C.2.1 Grammar

The implementation of the grammar is based on the shape the eventual tree will have. Every operator needs to have certain children. The LTL formulas are always performed over another LTL formula or a constraint. These constraints are made up of equations and these equations are made up of numbers and variables. Implementing the grammar this way allows for better structure in the parse trees used by the listeners. The following sections will go through the keywords and how they are used.

C.2.2 Expressions

At the bottom, it can be seen that mathematical expressions are supported. They follow the correct mathematical rules within precedence and associativity. The leaves will always be or become numbers. It is possible to make use of variables that are later filled out by the answers given from the simulation, except when an expression is used for for example the time that the simulation needs to run, as then the value needs to be known beforehand. Then, the grammar follows the precedence rules by first calculating powers and square roots, followed by multiplication and division followed by addition and subtraction. Working up the tree a rule *function* can be found. This allows the user to call mathematical functions over an expression, such as min, max and abs, as can be seen with the *func* rule.

C.2.3 Equations

These mathematical expressions can then be combined into an *equation*. Such an equation is a boolean equation. The rule contains a few comparators such as ==,<,> and <=. A user can use this rule to compare values, booleans and answers of methodcalls which are explained later.

C.2.4 LTL

The rules *tempSing* and *tempDoub* are in place to support the parsing of LTL formulas. Where *tempSing* refers to LTL formulas with a single argument; **F**, **G** and **X** and *tempDoub* to LTL formulas with two arguments; **U**, **R** and **W**. An extra rule *tempDouble* is in place to ensure the right-associativity of the **U**, **R** and **W** formulas. For user-friendliness and readability the options to fully write out the keywords is supported too with **eventually**, **globally**, **next**, **until**, **release** and **weakuntil**. An adaptation has been made to the **X** operator. Since the **X** operator is supposed to denote the next state and robotic systems not having such distinctive states as software programs have, it is decided to instead allow the user to define a time with the **X** operator. So, the user can define something along the lines of *in the next 0.2 seconds, proposition a needs to hold*.

C.2.5 Constraints

All these rules are eventually combined in the *constraint* rule. A constraint can be just a methodcall or equation, or a full LTL formula. Here, it is possible to compare entire constraints with each other by using logical and, or, implication or equality.

Another operation that can be done is using the logical not over a constraint. Aside from that, using the **for** keyword, the user can define for how long a constraint needs to hold for it to be seen as **True**. Lastly, the user can create constraint files that can be imported to allow for easy reuse of constraints by just giving the path pointing to a .constraint file.

C.2.6 Settings

Three settings can be set for a simulation: **for**, **with** and **in**.

The **for** keyword determines how long the simulation needs to be run. If no time is given, the default time of the simulator or model is used.

The **with** keyword allows the user to define initial variables within the simulation to easily change the environment in which a test occurs. It is possible to set multiple variables by repeating the **with** keyword or separating the variables with a comma.

The **in** keyword allows the user to define in which simulator the test should be run, allowing the user to easily switch between simulators merely by typing in a different name. Due to the modularity of the system, no other settings need to be changed or set when changing simulator, given that the names of variables are correct.

The **include** keyword allows the user to reuse settings from an external file by stating either a path pointing to such a file containing these settings or just the name of the file if it is stored in the same folder as the rest of the test files.

C.2.7 Scenario

The *scenario* rule serves as the definition of a scenario. In a scenario all of the above rules are packed together, allowing the user to define different constraints and settings per scenario while working in one file. It is possible to have multiple **then** keywords to define more independent constraints.

C.2.8 Feature

The *entry* rule then combines all these scenarios into one Feature to maintain the Gherkin-based structure as explained before.

C.2.9 Methods

It is possible to define methods for a feature. An example of a test using methods can be seen in Listing C.1.

```
Feature "Demo Test"

method ret(a,b,c):
    (%a < 10 and %b == %b)

Scenario "Overshoot tester"
    Given "double_slider_control.emx"
        in 20-sim for 1 seconds
        with "mode.C" = 1
        Then ret(0,10,"setpoint.C"
```

Listing C.1: Example method declaration and call.

The method needs to be defined at the very top of a feature. A function is declared according to the *method* rule in the grammar. The body of a method is a constraint that can make use of the provided arguments by using a % symbol followed by an identifier. The method can be called from a constraint as is defined by the *methodCall* rule. It is possible to call a method multiple

times with different arguments and since methods are in the scope of a feature, every scenario can make a call to it.

C.2.10 Filler Vars

These variables are used to provide more ease to reusing tests. These variables can be declared by using the & symbol followed by a number. This numbering starts at 0 and corresponds to the index of the list of the given arguments. The actual values given for these variables can be in the form of an *expr*, so a mathematical expression or variable. In Listing C.2 it can be seen that the arguments are the variable "axis.position" and the number 20.

```
Feature "Demo Test"

Scenario "Overshoot tester"
import "./test.file" ("axis.position",20)
```

Listing C.2: Test pre-substitution

In the "test" file that can be seen in Listing C.3 all the variables with &0 will be substituted with "axis.position" and &1 with 20.

```
Given "double_slider_control.emx"
        in 20-sim for 1 seconds
        with "mode.C" = 1
        Then &0 < 10 and "motor.angle" == &1
```

Listing C.3: Test.file

The output then becomes as is displayed in Listing C.4.

```
Feature "Demo Test"

Scenario "Overshoot tester"
    Given "double_slider_control.emx"
        in 20-sim for 1 seconds
        with "mode.C" = 1
        Then "axis.position" < 10 and "motor.angle" == 20
```

Listing C.4: After substitutions

C.2.11 Comments

Comments can be placed anywhere in the test. Line comments are preceded by //. Commenting out a certain set of characters can be done by placing them between /* and */.

D Implementation

D.1 Processing

Parse class The parse class functions as the wrapper around the listeners and interfaces with the Core. This is put in place so that the listeners are built independent from each other and only depend on information given by the Parse class, reflecting the responsibilities and modularity of the system.

The parse class has eight functions in total. The first function called in the system is the preproc function. Here, the class makes the Pre-processor's listener go over the input provided by the user. The model names, paths and filler vars stated in the user's input are substituted based on the values returned from the Pre-processor's listener. This substituted input is then put through to the Main processor's listener. Since the input will not change anymore between here and the Post-processor's listener, the parse tree is saved. This means that values and answers stored in the dictionary can be reused in the Post-processor's listener since the contexts of the nodes remain the same. This reduces the number of calculations that are required to be performed. Based on method calls returned by the Main processor's listener, the Parse class produces methods, which is explained more in detail in Section D.2. With the returned test data the Parse class builds Test classes.

Test class To improve interfacing and structure of the data passed between components, Test classes are introduced. These Test classes contain:

- **Modelname**: The name of the model the test will be checked against.
- **Test**: A list of variables that need to be checked during the simulation.
- **Result**: A dictionary that maps these variables to their results.
- **Time List**: A list with all timestamps of the simulation.
- **Init Vars**: A list of variables that are initialized before simulation.
- **Runtime**: A number representing the length of the simulation.
- **Simulator**: The name of the simulator the test will be run in.

As this test class is passed through between components, each component can take out the information that is required for their execution and fill out fields for which they have obtained values after execution. This keeps all the data for one test contained, so when running multiple tests at once, it can't happen that results are mixed up.

When postproc is called in the Parse class with a list of filled out Test classes, the Parse class handles the methods, runs the Post-processor's listener and cleans up the output produced by the Post-processor's listener by removing any variables. This output is later used by the User Interface to produce the output on the screen.

Expressions At the bottom of the tree are the expressions. With every variable filled out, these can be calculated first. Timestamps do not yet matter at this point so they are not taken into consideration yet.

It is possible to use three pre-defined functions over these expressions: abs, min, and max. The abs function turns every value in the list into an absolute value. The min and max functions

turn every value of the lists into what the minimum or maximum is respectively at that moment in time. For example, min([10,9,11,3,0,7,9]) will result in [10,9,9,3,0,0,0]. Even though the minimum becomes 0 at a later point in time, that is not yet the case at t = 0. This is also applied to the max function.

Equations When all expressions have been resolved, the boolean operators are reached higher up in the tree under the *equation* rule. Implemented comparators are <,<=, >, >=,! = and ==. Since from here and higher up the tree it only matters if statements are **True** or **False** and at what timestamps, this equation returns a list with **True** or **False** on index 0 and a list of timestamps for which the equation is **True** on index 1. Meaning that if the list of timestamps is empty, then a statement is generally **False**. To better the performance of the tool, these timestamps are truncated to time ranges, reducing the sizes of the lists passed around within the tree.

Constraints After all the equations have been resolved, constraints are the next step higher up in the tree. Its main entry point is through equations, as the other rules are all recursive back to the *constraint* rule, even method calls. Method calls are isolated constraints which are also at some point down the tree built up of equations. Therefore at this point in the grammar, each rule works with the same kind of data, allowing for recursion within the rule. That way, LTL formulas, method calls, and boolean operators can be combined to create larger constraints.

Boolean Operators There are five logical operators implemented. And, or, not, implication and biconditional. Each implemented function for these operators takes in two lists of time ranges and returns one with time ranges for which the operator holds, except for the not operator. The *not* operator takes just one list of time ranges and inverts it based on the full timelist.

The *and* operator is implemented through an intersection checker. This checks for overlap in time ranges in the two provided lists. The returned list consists of time ranges for which both sides of the and operator are **True**.

The *or* operator merges the two lists by checking for inclusive disjunction. Overlapping time ranges are merged to just one time range to prevent duplications.

Implication and *biconditional* are implemented by combining the checkers mentioned before. *Implication* has the following equivalence:

$$\varphi \Rightarrow \psi \equiv \neg\varphi \vee \psi$$

Therefore a combination of the inverter and inclusive disjunction checker can be used to produce an implication checker. *Biconditional* has the following equivalence:

$$\varphi \Leftrightarrow \psi \equiv (\neg\varphi \vee \psi) \wedge (\neg\psi \vee \varphi)$$

Therefore this checker can also consist of just inversions and inclusive disjunctions.

Timed Constraints When a constraint should not hold for just one moment in time, but for a longer period, it is possible to use the **for** keyword. The time can be defined ranging from nanoseconds to hours. A checker for this takes in a list of time ranges and checks if any of them are longer than the time set for it to hold. If so, a list is returned consisting of time ranges for which this holds.

LTL Formulas The LTL formulas are split in two rules. One for the singular formulas **F**, **G** and **X** and one for formulas that take in two arguments: **U**, **R** and **W**.

The **F** operator takes in a list of time ranges. If this list is empty, meaning that the statement is never **True**, the list is returned as empty. If the list is not empty, then that means that the operator holds from t = 0 until the last timestamp for which the provided proposition is **True**. The returned values then are **True**, along with that time range.

The **G** operator checks if the provided list of time ranges consists of one entry that spans the entire time for which the simulation has run by comparing it against the full timelist. If that is the case, then **True** is returned along with the original list. If not, **False** is returned with an empty list.

The **X** operator checks if a statement holds in the next given time. This is done by deducting that time from the starting time of a provided time range, as this operator starts holding **True** for that given time before the given proposition is **True**. **True** is returned along with the timelist with deducted values. If φ is never **True**, then $\mathbf{X}\varphi$ returns **False** and an empty list.

The **U** operator is implemented using its recursive expansion:

$$\varphi \mathbf{U} \psi = \psi \vee (\varphi \wedge \mathbf{X} (\varphi \mathbf{U} \psi)).$$

Which can be interpreted as ψ needs to hold, or φ needs to hold given that in the next state φ holds or ψ holds. This process is repeated until ψ finally holds and until that point, in every preceding state φ has to hold. This **X** operator is different from the **X** operator used in the grammar, as this one retrieves the state succeeding the current state and is not bound by any time parameters.

The **W** operator is similar to the U operator. The difference is that with $\varphi\mathbf{W}\psi$, ψ does not necessarily need to hold. If that is the case, then φ needs to hold globally. This leads to the following equivalence:

$$\varphi\mathbf{W}\psi \equiv (\varphi\mathbf{U}\psi) \vee \mathbf{G}\varphi \equiv \varphi\mathbf{U}(\psi \vee \mathbf{G}\varphi)$$

By reusing the already implemented **U** and **G** checker, the **W** operator is implemented.

A similar approach is taken for the **R** operator.

$$\varphi\mathbf{R}\psi \equiv \neg(\neg\varphi\mathbf{U}\neg\psi)$$

where a combination of the inverter and the **U** checker can be used.

Then, Scenario and Feature Rules The exit function of the *then* rule checks if the underlying constraint has returned **True** or **False**. Since from this point on, timestamps are no longer relevant, they are omitted. The top of a constraint may consist of just an equation, so no temporal formulas. By default is this constraint checked with the **G** operator here to determine if **True** or **False** will be returned. This is repeated by the exit function of the *scenario* rule, combining every *then* it contains. It is repeated once more by the exit function of the *feature* rule, that applies this process to every *scenario* it contains. This answer is also the final answer to the test.

Output

When all scenarios have been completed, the results are accumulated and combined with a logical and. The final result of the test then results in **True** or **False**. The output is extended with a more elaborate output that has been logged during the walking of the tree. In each equation, constraint and scenario, it is checked if a certain proposition holds. If it does, it is checked with the **G** checker to see if something is Always True or just Sometimes True. This output is eventually displayed on the screen as the final answer to the tests.

D.2 Method Handling

Methods consist of two parts. The definition and calls. Only one method can be defined under a certain name, meaning method overloading does not work. A method can be called multiple times, which means that a simple substitution does not work here. On one end are these calls that provide arguments and expect an answer to this call in the same place. This needs to be known before the simulation, therefore this data is retrieved by the Main processor. On the other end is the definition of the method, which requires these arguments before it can calculate and return an answer. Since substitution does not work, a better solution needs to be implemented to bridge this gap. This is done by the Parse class in the following order:

1. It collects the calls and arguments lists from the Main processor.
2. It transforms these lists into one list consisting of the method's name, locations of the method calls, a mapping of a method's arguments with the provided arguments from the call and a scenario number.
3. Before the Post-processor is called, the answers to the methods are calculated, so the Post-processor does not have to do anything special when reaching a method call.

Since it is not possible to tell the walker to re-walk a certain part from within the listener, the Parse class has to do this from the outside. It only walks over the methods branch of the parse tree to achieve this effect. Every method is walked as many times as it is called. For these calls, the listener is provided with argument locations in the method along with their value. It works with the results and timelist obtained from the simulation for the corresponding scenario, as this influences in what environment the method is called.

Finally, when the answer for each call has been obtained, the Parse class combines the call location and answer and passes it on to the Post-processor. By providing the exact location in the parse tree along with the value, the Post-processor can use these answers for later calculations.

D.3 User Interface

The Graphical User Interface is created in Python, making use of the library Tkinter. Reiterating from the requirements, the user interface should have the possibility to run, modify, add and delete tests. In Figure D.1 below can be seen that buttons are in place to allow for this sort of functionality, along with some extensions.

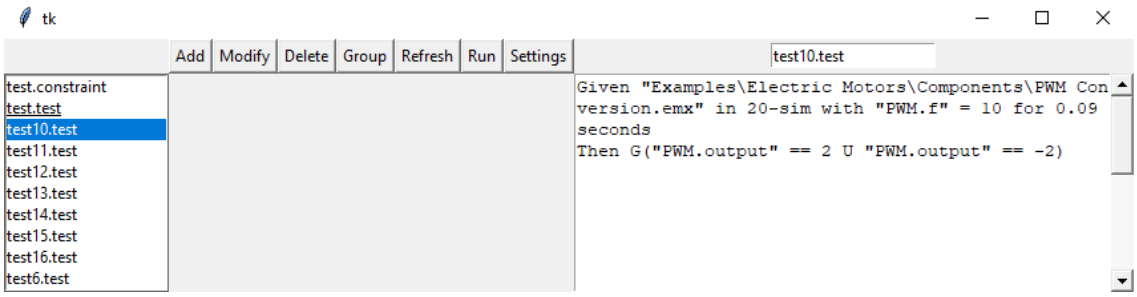

Figure D.1: The GUI.

To the left is a list showing currently saved tests. The user can select any of these and then click on *Modify*, *Delete*, *Group* or *Run* to do the corresponding action. Selecting two tests and clicking the Group button allows the user to merge two tests into one, which is put in place so that users can easily combine smaller tests to make a larger, more expansive test put together in a single test file.

To the right are two text fields. The one at the top allows the user to define the name of the test and below it is a field where the user can define the test itself. When the test has been fully described by the user, they can press the *Add* button for the test to be saved under the given name and will then be visible in the list to the left. When a test has been selected on the left and the *Run* button is pressed, the simulation will be run and the answers to the test will be returned. An example output can be seen in Figure D.2.

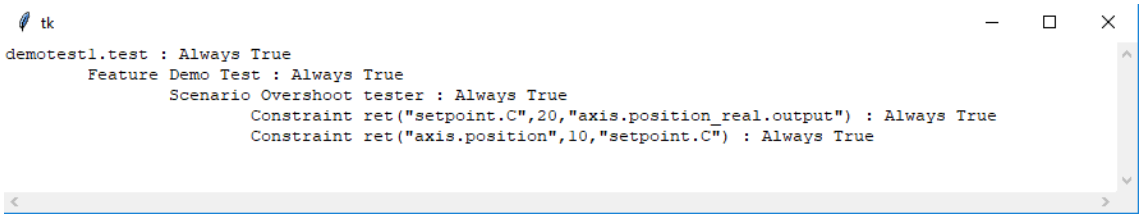

Figure D.2: Example output.

Where the test name is stated at the top, along with Always False, Sometimes True or Always True, depending on the result. The deeper levels are indented to indicate their level, also followed by their corresponding answer. The *Settings* button is put in place to lead to the dialog in Figure D.3.

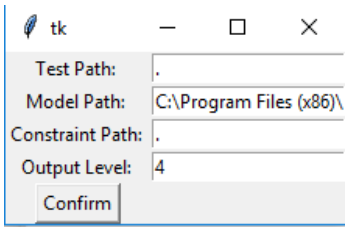

Figure D.3: Settings dialog.

These four settings are entirely optional and only intended to make the tool easier to use. *Test Path* defines the location of the saved tests. *Model Path* and *Constraint Path* are two variables used during testing to point to the locations of saved models and constraints, so that the user does not have to type in full paths each time. *Output Level* is an option used by the User Interface detailing how verbose the output is preferred to be. These settings are stored in a small

text file so that the settings are preserved between sessions and the user does not have to fill them out again every time.

D.3.1 Command-line

Since the User Interface only needs to provide a path to the test file and optionally some settings such as *Model Path*, *Constraint Path* and *Output Level*, a much simpler User Interface would suffice. If the user does not feel like having to click through a GUI or wants to integrate the tests into another system, the User Interface could be slimmed down to just a call from the command-line, like in Figure D.4 below.

```
>py main.py "./demotest1.test","./demotest2.test" mpath="C:\Program Files (x86)\20-sim 4.7\Models" cpath="." olvl="3"
demotest1.test : Always True
        Feature Demo Test : Always True
                Scenario Overshoot tester : Always True
                        Constraint ret("setpoint.C",20,"axis.position_real.output") : Always True
                        Constraint ret("axis.position",10,"setpoint.C") : Always True
demotest2.test : Always True
        Feature Demo Test : Always True
                Scenario Includes : Always True
                        Constraint ret(0,10,"setpoint.C") : Always True
```

Figure D.4: Running a list of tests from command-line.

Here, multiple test files can be tested through at once by separating them with a comma. They are followed by optional settings. The system provides two outputs. One textual output that can be printed to the command-line window and a boolean which corresponds to the failing or succeeding of a set of tests. This boolean could, for example, be used in more elaborate batch scripting.

D.4 Dataflow

Putting all these components and responsibilities together in one picture gives what can be seen in Figure D.5. The tool works as a long tube through which data passes and is transformed in each component, going from user input to output without any branches.

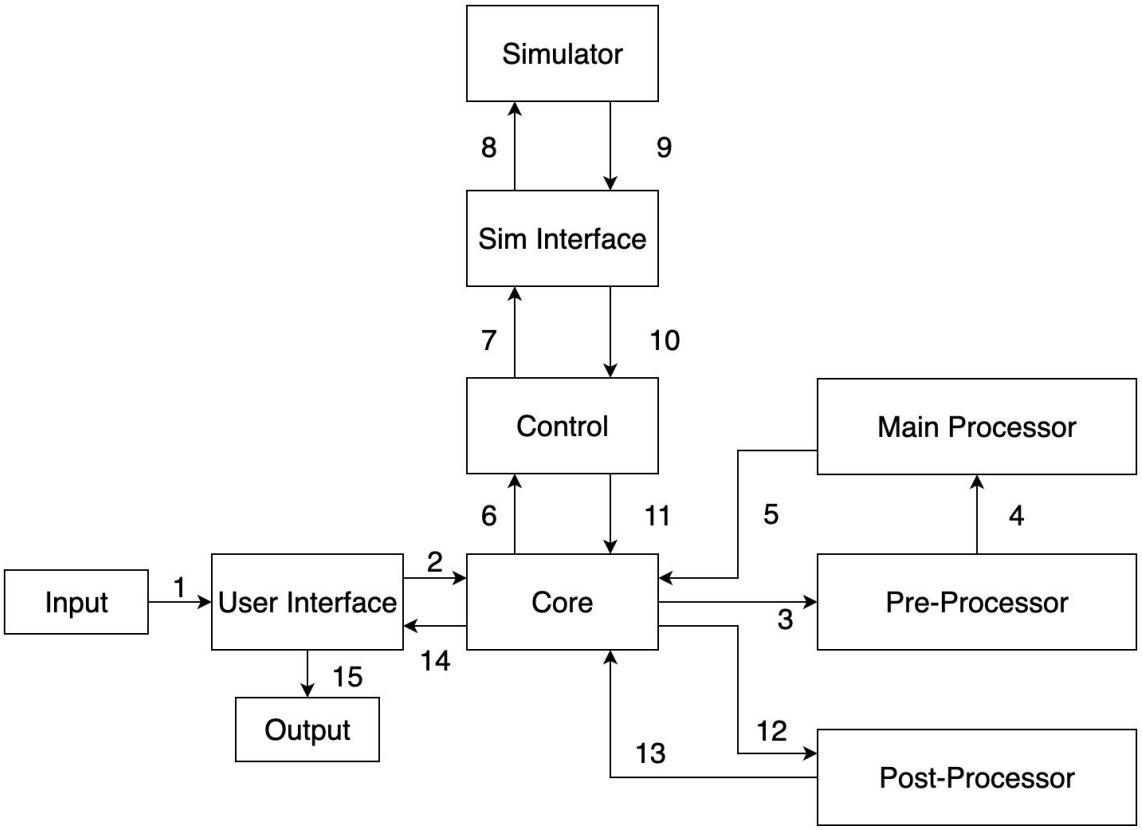

Figure D.5: Dataflow overview of the system.

Each transfer of data is marked with a number showing the steps it takes to be processed through the tool.

1. Input is provided by the user.
2. The user pressed run and the test is forwarded to the Core.
3. The taken input is put through towards the Pre-processor in the Parser.
4. The test is put through to the Main processor after substitutions.
5. The Main processor produces a list of these Test classes that are returned to the Framework.
6. This list of Tests is forwarded to the Simulation Control component.
7. The Simulation Control module provides the correct simulator interface with simulation data.
8. Internal communication between the sim interface and the simulator itself.
9. Simulator returning requested data.
10. The sim interface returns the timestamped results of the tracked variables during the simulation.
11. The Simulation Control component fills out these results into the Test classes in the original list of tests and returns the list of Test classes to the Core.
12. This list of tests is provided to the Post-Processor.

13. The Post-processor returns answers to all the tests.
14. The Core forwards these results to the User Interface.
15. The results are outputted.

Bibliography

Cucumber (2019), Gherkin Documentation, https://cucumber.io/docs/gherkin/, accessed: 2019-09-30.

Grimm, M., R. Anderl and Y. Wang (2014), Conceptual approach for multi-disciplinary cyber physical systems design and engineering, in *Proceeding of TMCE 2014*.

Jeswiet, J. and R. Helferty (1995), Measuring robot repeatability an application of ISO and ANSI standards.

Kumičáková, D., V. Tlach and M. Císar (2016), Testing the Performance Characteristics of Manipulating Industrial Robots.

Lee, E. and S. Seshia (2015), Introduction to embedded systems: a cyber-physical systems approach.

Pitonakova, L., M. Giuliani, A. Pipe and A. Winfield (2018), Feature and Performance Comparison of the V-REP, Gazebo and ARGoS Robot Simulators.

Rajkumar, R., I. Lee, L. Sha and J. Stankovic (2010), Cyber-physical systems:The next computing revolution, in *Proc. Design Automat. Conf.*, pp. 731–736.

RoboDK (2019), ISO9283 Performance Testing, https://robodk.com/doc/en/Robot-Validation-ISO9283.html, accessed: 2019-02-24.

Saglietti, F., D. Föhrweiser, S. Winzinger and R. Lill (2015), Model-based designand testing of decisional autonomy and cooperation in cyber-physicalsystems, in *Proc. 41st Euromicro Conf. Softw. Eng. Adv. Appl.*, pp. 479–483.

Sawant, A. A., P. H. Bari and P. M. Chawan (2012), Software Testing Techniques and Strategies.

Schmaltz, J. (2017), Software Specification.

Yagan, O., D. Qian, J. Zhang and D. Cochran (2012), Optimal Allocation of Interconnecting Links in Cyber-Physical Systems: Interdependence, Cascading Failures and Robustness.

Zander, J. (2013), Model-based testing for execution algorithms in the simulationof cyber-physical systems, in *Proc. IEEE AUTOTESTCON*, pp. 1–7.

Zhou, X., T. Huang, X. Gou and S. Yang (2018), Review on Testing of Cyber Physical Systems: Methods and Testbeds.