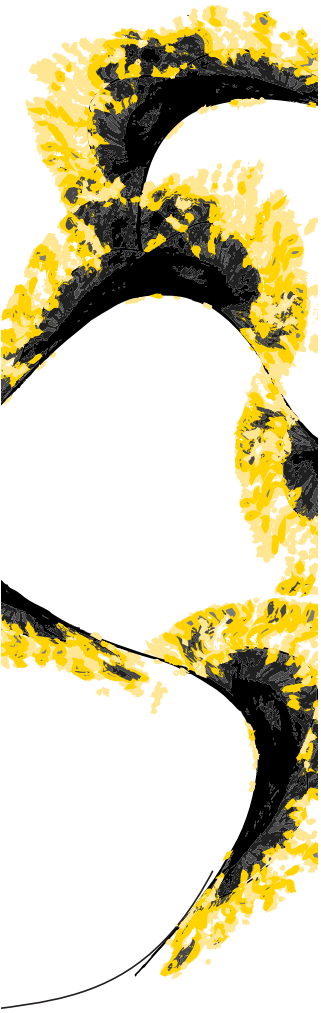


RAM ROBOTICS AND MECHATRONICS

SUBMODEL-AWARE TESTING

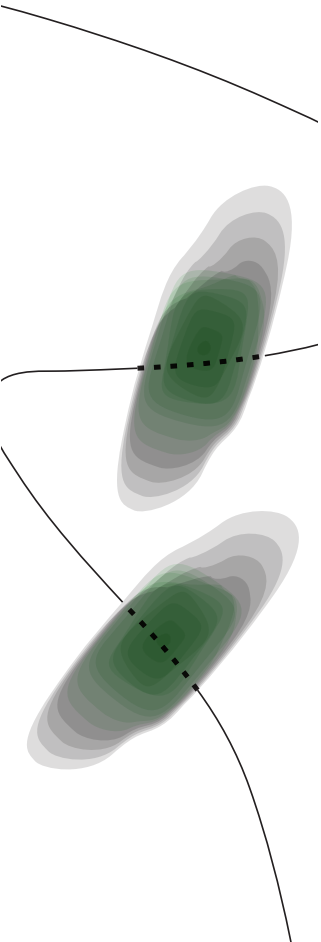
H. (Hanna) Haven

MSC ASSIGNMENT

Committee:
dr. ir. J.F. Broenink
T.G. Broenink, MSc
dr. ir. A.Q.L. Keemink

September, 2020

046RaM2020
Robotics and
Mechatronics EEMCS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands



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

Summary

An automatic test tool was developed by the Robotics and Mechatronics group (Jansen, 2019), (Broenink et al., 2020) in 2019. This tool supports all the basic test options but it was necessary to implement more options, specifically to implement submodel-awareness and co-simulation. Submodel-awareness and co-simulation make it easier to test cyber-physical systems. Both the addition of submodel-awareness and co-simulation add value to the test tool. In this research the test tool and existing testing methods were investigated in order to define the best adaptation such that submodel-awareness could be implemented as efficiently as possible. After analysis it was determined that step definitions (inspired by Gherkin (Cucumber, 2019)) were the best option. The step definitions ensured that the test would be easy to read, simple to understand and it would make the test tool easily expandable. Previously, the addition of submodel-awareness would have taken five steps, with the addition of step definitions it takes two. The addition of step definitions also improves the human readability and usability, as the test was sometimes hard to understand.
Additionally, to support co-simulation, the framework CoHLA (Configuring High Level architecture) is implemented in the test tool. After some research it was concluded that the test tool needed a code adjustment to make it easier to add frameworks or simulators in the future. After this adjustment the number of steps necessary for adding a simulator is reduced from four to one.
The test tool now supports CoHLA, and thus co-simulation. It has also been extended to support step definitions, which makes submodel-aware testing possible. The step definitions also make it easier to expand the test tool with new functionalities.
Improvements for the future include the addition of (error-) messages to the console and the expansion of the results. This last improvement allows for the result to contain graphs, tables and differences between the expected and the actual results.

Preface

Before you lies the master thesis "Submodel-Aware Testing", where a existing test tool will be improved and extended.

I would like to thank everyone who helped me with writing this thesis. This includes, of course, my day-to-day supervisor Tim Broenink. Whenever I had a question you answered immediately and was always available to help.

My thanks also goes to my colleagues at TRIMM. When I needed a change from working at home they made it possible to finish my thesis at work (of course following all the Corona regulations). To all my friends and family who helped me by giving feedback and having my back: thank you for motivating me to keep improving my thesis.

Hanna Haven
Enschede, 19-08-2020

Contents

1 Introduction **1**
1.1 Context and problem statement 1
1.2 Goals and approach 1
1.3 Outline 2

2 Background **3**
2.1 Testing 3
2.2 Current test tool 6
2.3 CoHLA 7

3 Analysis **8**
3.1 Analysis of the test tool DSL 8
3.2 Addition of the step definitions 8
3.3 Addition of submodel-awareness 9
3.4 Addition of a simulator 9
3.5 Addition of co-simulation 10
3.6 Requirements 11

4 Design and implementation **13**
4.1 Step definitions 13
4.2 Submodel-aware testing 16
4.3 Simulators 16
4.4 Implementation of CoHLA 17
4.5 Conclusion 18

5 Results **19**
5.1 Results of the step definition 19
5.2 Co-simulation/CoHLA 22
5.3 Automatic import 23
5.4 Conclusion 24

6 Conclusion and recommendations **25**
6.1 Conclusions 25
6.2 Recommendations 26

A Grammarfile **28**

Bibliography **30**

1 Introduction

1.1 Context and problem statement

Cyber-physical systems are systems which consists of hard- and software. An example is a controlled motor. This controlled motor consists of different sensors, motors and also includes software with control code. The early stages of modelling of this system is done using simulators. In the simulator different combinations of for example the sensors can be simulated and different kinds of controller software can be tested. By using the simulator it is not necessary to realize the system, first it is simulated in ideal and non-ideal environments to find the perfect combination of hard- and software. When the perfect combination is found the realization of the system can be done and real-life tests can be conducted.

In 2019 an automatic test tool for cyber-physical systems was developed in the Robotics and Mechatronics group (Jansen, 2019) (Broenink et al., 2020). This test tool is developed to support the development of cyber-physical systems and make it easier to model cyber-physical systems. This tool can test different scenarios for the cyber-physical systems, this means that models of cyber-physical systems can be automatically tested with different parameters and checked with constraints.

As is explained above a cyber-physical system consists of hard- and software. In the example of the controlled motor, the different parts of the system can also be defined as submodels. The sensors are all different submodels, just as the motor. It is important for the modelling of a new system to test different variations of submodels. In the example of the controlled motor in the previous paragraph, it is important to test different kind of sensors and controllers to find the perfect combination. It is desirable that the test tool is able to test different kind of submodels because it will make the modelling of the cyber-physical systems easier.

1.2 Goals and approach

Three goals are determined for the improvement of the test tool. These three goals are listed below:

1. The first goals is to implement submodel-aware testing. After research on the test tool (see Chapter 2.2) it is determined that the implementation of submodel-aware testing is possible.

2. The second goal is the improvement of flexibility. As can be seen in Chapter 2.2, the addition of new functionalities and simulators is relatively hard. By improving the flexibility the test tool will be improved in readability, ability to expand and ease of use. After research is was conducted that the addition of step definitions (Section 2.1.2) can improve the flexibility. The addition of new functionalities and simulators will be easier and less work than before.

3. The last goal is to implement the ability to support the testing of submodels which are modelled in different simulators. This is called co-simulation and is the simulation of models, which are modelled in different simulators, in parallel with each other, while the models have a real-time connection with each other. Different frameworks are researched (see Chapter 3.5) and CoHLA is chosen.

As can be seen in the goals above, the solution of the second goal will make the first and third goal easier. The addition of flexibility will make it easier to expand the test tool. This is why the the addition of the flexibility will be done first, it will make it easier to implement the other two goals.

1.3 Outline

The report consists of six Chapter. First is the background (Chapter 2) where the psychology of testing, the current test tool and CoHLA are explained. This background information is necessary for Chapter 3, the analysis. Here the information is used for the requirements. In Chapter 4 the design and implementation of the requirements is explained: the design and implementation of the step definitions and the implementation of CoHLA. The implementation is tested in Chapter 5, where different tests are executed to test the step definitions and the implementation of CoHLA. The report is concluded in Chapter 6. In the appendix the new grammar is presented.

2 Background

This Chapter provides the background information on a few different subjects. First testing itself, the psychology behind testing and automated testing are explained. Then the current test tool is explained, how it works and how it is expanded. Finally the basics of CoHLA are explained.

2.1 Testing

Software testing is the most important part of software development (Bhatti et al., 2019). The testing of developed software improves the certainty that there bugs or errors in the developed software. The difference between errors and bugs are that errors is faulty software, the code breaks and an error appears. Bugs are unwanted or undesired behaviour, which do not necessarily break the code

First the psychology behind testing is explained followed by automatic testing.

2.1.1 Psychology behind testing

Most developers think about testing the wrong way, they think the purpose of testing is to make sure the functions of the program work, or to make sure there are no errors in the program (Myers et al., 2011). In The Art of Software Testing (Myers et al., 2011) the definition of testing is stated as follows:

"Testing is the process of executing a program with the intent of finding errors"

Testing is used to make sure the functionalities work, but the goal of testing itself is to find errors. It is stated that there are errors and bugs in every software program. These errors and bugs must be found and dealt with to improve the quality of the program. Test cases are developed specifically for the program. These test cases contain what the program should do when certain actions are performed plus what the result of the test case should be. These test cases must be written before any testing is done to make sure all the important functions get tested and to make sure the test cases have structure, are repeatable, and are feasible. These test cases consist of features, which define different scenarios to test the functions of the software program.

An example of a software program with test cases is stated in The Art of Software Testing: "The program reads three integer values from an input dialogue. The three values represent the lengths of the sides of a triangle. The program displays a message that states whether the triangle is scalene, isosceles, or equilateral."

The test cases include the scenarios where, for example, negative numbers are entered, the entry of the number one or the insertion of valid entries which should not give an error (in contrast to the first two scenarios). This is important, not only the invalid entries need to be tested (entries which should give an error) but also the entries which are the valid values. The test cases must include every possible entry, but not every entry can be tested. As for the example above, when the input is `1,1,1` it can be said the output will be the same as for the input of `2,2,2`.

In The Art of Software Testing 10 principles are stated, these are recalled below with an additional explanation if needed:

1. A necessary part of a test case is a definition of the expected output or result.
2. A programmer should avoid attempting to test his or her own program.

3. A programming organization should not test its own programs.
4. Any testing process should include a thorough inspection of the results of each test.
5. Test cases must be written for input conditions that are invalid and unexpected, as well as for those that are valid and expected.
6. Examining a program to see if it does not do what it is supposed to do is only half the battle; the other half is seeing whether the program does what it is not supposed to do.
7. Avoid throwaway test cases unless the program is truly a throwaway program.
8. Do not plan a testing effort under the tacit assumption that no errors will be found.
9. The probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section.
10. Testing is an extremely creative and intellectually challenging task.

In addition, in the Tester Foundation Level Syllabus (Muller and Friedenberg, 2011) there are seven different principles:

1. Testing shows presence of defects, testing can show defects, but if the testing does not show defects it is not guaranteed that there are no defects.
2. Exhaustive testing is impossible, it is impossible to test every test case.
3. Early testing, testing should be done as early as possible to detect defects.
4. Defect clustering, testing effort should be focused proportionally on the defects density that is expected but only found later on. If there are a lot of defects at one place in a program this should be tested thoroughly once the defects found early on have been fixed.
5. Pesticide paradox, if the same tests are done over and over again, then the test will not find any more defects. The tests should be reviewed, revised and expanded to find more defects.
6. Testing is context dependent.
7. Absence-of-errors fallacy, if the system is built unusable finding and fixing defects do not help.

The second list shows the most common principles, and are displayed on various websites (Guru99, 2014), (Kumar, 2019), (SM, 2016). The principles from the syllabus are the ones which are generally applicable for testing, where the principles out of the The Art of Software Testing are more specific to the methods of testing itself.

In the syllabus the reason why testing is necessary is because defects in software can lead to problems. These problems can be something small, such as losing lots of money but also something as significant as injury or even death. On the other hand testing is important because it also can save a lot of money. When a defect or bug is recognized early in the process, the lower the costs of correcting it and the higher the probability of correcting them (Myers et al., 2011). In The Art of Software Testing the reason why testing is important is similar:

"Software testing is a process, or a series of processes, designed to make sure computer code does what it was designed to do and, conversely, that it does not do anything unintended."

2.1.2 Automatic testing and Gherkin

Automatic testing is done with a programmed test script, which is executed every time a new feature is introduced in the program (or an old feature deleted). This is called regression testing, all software (even the part which is not changed) is tested again to check if the new software has broken a functionality. Automatic testing is mainly done with a dedicated DSL (Domain Specific Language), the DSL which is used mostly is the Gherkin syntax (Cucumber, 2019). The three main words are 'Given', 'When' and 'Then'. After these words steps are defined (step definitions) where different steps execute different commands to test the functionality. The step definitions are written out of the users point of view, because the test is done from the users point of view. An example:

```
Given I am on the homepage
When I log in
Then I should be on the logged-in page
```

Listing 2.1: Example of a test with Gherkin-grammar

This is an example of an automatic test for a website. The user is not logged in and with the test the user logs in. If, for example, a function is added but the addition of that function causes the log-in function to break, then this test fails. When these kinds of tests fail the developer can see fairly quickly what is broken and can act accordingly.

This test itself consist of a feature, this feature is divided into different scenarios and every scenario consists of multiple steps, with these steps a flow can be specified. An example of this is shown in Listing 2.2:

```
Feature "Feature1"
        Scenario "Scenario1"
                Given When Then

        Scenario "Scenario2"
                Given When Then
```

Listing 2.2: Example of a test with a feature and scenarios

In the example in Listing 2.2 two different scenarios are specified, in one feature. This is the way the scenarios and features are declared. Scenarios can fail independently, this means that even if `scenario1` fails, `scenario2` is executed.

The design of a new DSL requires guidelines. Four guidelines are presented in Micallef and Colombo (2015):

1. The DSL should be simple; people should be quick in understanding the language.
2. The DSL should exhibit similarity to another language; most of the times this is English.
3. The DSL should be highly domain-specific and parsimonious. This is so that any notions not related to the domain in question are omitted. Yet the designers should strive to make the language as complete as possible.
4. The DSL should be easily extensible and reusable. It should be easy to add features to the language. It should also be reusable in that the same grammar can be used as the base of a new language when required.

Research was done on the use of DSLs for automated software testing (Micallef and Colombo, 2015). Five studies in the industry where two use the Gherkin syntax and the three use a more rigidly defined language grammar. The main four lessons learnt from this research are as follows:

1. In the interest of long-term feasibility a dedicated language owner is a requirement. This means that the standard grammar of the DSL will be maintained by this person, who will make sure there is consistency within the language, avoids duplication etc.

2. Having a development process which caters and makes space for DSL development. Here it is also stated that an ad hoc approach is unlikely to work, especially when deadlines approach.

3. As the language grows, a tool might be handy. Something like a dictionary where all the grammar rules and functions are specified. This can help new users for learning an existing DSL.

4. Finally management buy-in is essential. This is because when the process of developing and maintaining is in competition with software delivery, management will almost certainly abandon the testing process.

Number 4 is the main reason (automatic) testing is not done in many different applications, for example in the industry. If the software development is past the deadline, testing is almost always omitted in the process to shorten the development time. The same happens when the budget is exceeded.

2.2 Current test tool

The current test tool (Jansen, 2019) is developed in Python and can simulate in the simulators 20-sim and V-REP. In this Chapter the test tool and the steps for expanding the current test tool are explained.

The current grammar is a variation on the Gherkin-syntax. The Gherkin syntax is explained in Section 2.1.2, the alteration which is made is to expand the grammar to support LTL (Linear Temporal Logic) formulas. An example of a test-feature can be seen in listing 2.3. The Feature, Scenario, Given and Then are the same as is in the normal Gherkin syntax. The difference starts at the 'WHEN', this command is omitted at all. Instead the following keywords are declared: `include`, `with`, `in` and `for` (only `with`, `in` and `for` are used in the example in listing 2.3).

```
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 2.3: Example of a test with the adjusted Gherkin-grammar

The `Given` imports the model, where the whole path to the model must be given. The `in` specifies the simulator in which the simulation must be executed. After the specification of the simulator the different settings can be specified. There are two kind of settings, therefore there are two keywords: `with` and `for`. `for` is to set the time in which the simulation must be executed. The keyword `with` is for the settings of the model. Different parameters can be set (or changed) with this keyword. The names of the parameters must be the same as the ones which are in the model. The last keyword, `include`, is to include different files which can contain settings, constraints or other tests.

The constraints are being indicated with the `Then` keyword. These constraints can be defined using LTL-indication. After the execution of the simulation, the test tool collects the data and checks the constraints of the test. The output of the tool can be: `TRUE`, `SOMETIMES TRUE` and `FALSE`.

2.2.1 Addition of a new functionality and keyword

Currently there are two different keywords which can alter the simulation: `with` to set a variable and `for` to set the simulation time. The addition of a keyword requires a couple of steps. These steps are stated in the next list but the in-depth explanation is given in Jansen (2019).

1. The addition of a keyword to the grammar. This keyword needs to refer to the functionality which is added in step 5.
2. The generation of the Parser, Listeners and Lexer. This is done with the tool ANTLR (Parr, 1989).
3. Implementation of the function in the Listeners.
4. Implementation of the function in the code.
5. Implementation of the function in the interface-file, where the simulator is controlled.

2.2.2 Addition of a simulator

The two simulators for which the test tool now works are 20-sim and VREP, to add an additional simulator the following steps need to be taken:

1. The simulator must be added to the grammar.
2. The generation of the Parser, Listeners and Lexer. This is done with the tool ANTLR (Parr, 1989).
3. In the general code the simulator must be added so the right interface-file can be imported.
4. A new interface-file must be made and the functions need to be implemented so the simulator can be controlled.

2.3 CoHLA

CoHLA (Nägele (2020b)) is a framework which is able to co-simulates models. CoHLA uses FMU's (Functional Mock-up Unit) and is able to co-simulate these FMU's. Co-simulation is the simulation of submodels in parallel, where information is shared between those submodels. These submodels can be the same but can also differ depending on the simulation (Nguyen et al., 2017). CoHLA has its own DSL and is a command line tool. To use CoHLA an installation manual and user manual are available (Nägele, 2020a).

CoHLA is a terminal-controlled system, different settings can be specified in the command. All settings and models are defined in the `.cohla` file, in so-called Federate and Federateclasses. From this `.cohla` file the config files are generated. Two important settings are the situation and the scenario. These settings are defined in the Federate, the Federate makes sure all the different models (called Federateclasses) are connected and the simulation can execute. A situation is a file in which different settings can be specified. A scenario is a file where the input signal(s) can be specified in terms of time. The signal changes according to the times and values specified in the scenario-file.

3 Analysis

In this Chapter the principles which were stated in Chapter 2 are taken into account against the current test tool.

3.1 Analysis of the test tool DSL

In Section 2.1.2 the four guidelines for a DSL design are specified. In the next list, the same guidelines are specified with an addition whether this guideline is applied in the current test tool:

1. *The DSL should be simple, people should be quick in understanding the language.*
   The grammar is simple enough, but by just using keywords the readability (and understanding) of a test is not simple. When keywords are connected to functions in the test tool it is possible that the exact meaning of the keyword becomes unclear, especially when there are a large (20+) number of keywords implemented.
2. *The DSL should exhibit similarity to another language, this is most of the times English.*
   It has a similarity to English, but there are no sentences, only keywords with a lot of math behind it. It can be concluded that while it contains some English words, there is no similarity to English.
3. *The DSL should be highly domain-specific and parsimonious. This is so that any notions not related to the domain in question are omitted. Yet the designers should strive to make the language as complete as possible.*
   The DSL is highly domain-specific and parsimonious, where it is not as complete as possible, this guideline is followed.
4. *The DSL should be easily extensible and reusable. It should be easy to add features to the language. It should also be reusable in that the same grammar can be used as the base of a new language when required.*
   The DSL is reusable, but it is not extensible. If a function (or simulator) is added to the test tool (as can be read in Section 2.2.1 and Section 2.2.2) there is no quick or easy way to do so.

As can be seen in the list above, three of the four guidelines are not followed in the test tool. The test tool is not easily extensible or reusable, there is no similarity to another language and the understanding of a test it not simple.

The test tool can be improved on three points: simplicity and understanding (point 1 in the list above), similarity to another language (point 2 in the list above) and extensibility: adding a functionality and simulator (point 4 in the list above). The addition of step definitions improves the test tool on the points 1, 2 and 4 (functionality wise). Of the last point only the implementation of additional functionalities can be solved by implementing step definitions.

3.2 Addition of the step definitions

The main reason for adding step definitions is to improve readability and similarity (point 1 and 2 in the guidelines of Chapter 3.1). The keywords alone do not provide the readability. As more functions are added, more keywords are also added. More keywords confuse the user. By adding the step definitions the readability improves (point 1 of the guidelines in Chapter 3.1) and it exhibits more similarity to English (point 2 of the guidelines). For example, currently when a user wants to change a value in the test, he uses `with`. When using a step the user can define it as listed in Listing 3.1.

```
Now I set parameter PWM.frequency to 50
```

Listing 3.1: Example of a possible step

The step in Listing 3.1 is clearer than the keyword `with`. An example of setting a variable with the keyword `with` is listed in Listing 3.2. Where for the setting of the variable in Listing 3.2 might need additional information to explain what `with` means, the step in Listing 3.1 is immediately clear.

```
with "PWM.frequency" = 50
```

Listing 3.2: Example of a possible step

With the addition of step definitions it becomes easier to add a new functionality, which complies to guideline 4. The new functionality is called by a step instead of by a keyword. The addition of new functionality and new keyword to the test tool takes five steps, as can be seen in Chapter 3.1; for this addition knowledge of the tool ANTLR and the code is necessary. In Section 2.1.2 Gherkin was explained where generic steps would be defined and to those steps functions were linked to those steps. With the addition of step definitions it takes two steps to add a new functionality.

3.3 Addition of submodel-awareness

The addition of submodel-awareness to the test tool is done using step definitions. Submodel-awareness is added for 20-sim, because 20-sim has a function which can replace submodels. With the addition of step definitions it is only necessary to define a step and to declare the 20-sim function `replace_submodels` (ControllabProducts, 2015) in the interface-file of 20-sim. The step connected to this functionality is defined in Listing 3.3.

For the replacement of the submodels a couple of aspects are necessary: the submodels need to be interchangeable, this means that the imported submodel needs to have the same amount of ports and needs to accept the signals which are going to be connected. This requirement, the submodels must comply with the model, is the responsibility of the user. If the user imports a submodel which is not compatible with the model, the test tool does not give a specified error. The function `replace_submodels` of 20-sim needs two variables: the name of the submodel in the model itself and the path of the submodel which is going to be imported. These two requirements are the name of the submodel which needs to be replaced and the path to the submodel which is replacing the first submodel. In Listing 3.3 the step is defined. The word `replace` is chosen because the submodel is literally replaced. This step is connected to the 20-sim function `replace_submodel`.

```
I replace existing_submodel with path/to/external_submodel
```

Listing 3.3: The submodel-awareness step

The addition of a step is easier than the addition of a functionality and keyword. By adding step definitions to the test tool it takes two steps instead of five steps to add a functionality and keyword. Fewer knowledge is necessary: only knowledge of how to add a step, basic knowledge of Python and knowledge of the simulator. The knowledge about the tool ANTLR and the in depth knowledge about the test tool is not necessary any more.

3.4 Addition of a simulator

The addition of a new simulator to the test tool takes four steps. The first two steps are the same as when adding a new functionality and keyword to the test tool: addition to the grammar and the generation of the lexers, listeners and parsers. This should be easier (guideline 4 of Chapter

3.1). These four steps can be reduced to one step, the last one: *A new interface file must be made and the functions need to be implemented so the simulator can be controlled.* This is the only step which cannot be omitted. The user always needs to make the interface file and connect the functions in this file. The other three steps are always the same and it is possible to automate them.

If the user only needs to make and fill the interface file, then it is sufficient to have knowledge of Python and the new simulator. This advantage complies with guideline 4, this extension makes it easier to add new simulators to the test tool.

3.5 Addition of co-simulation

In Chapter 2.3 co-simulation is briefly explained, and it is a goal that a co-simulation framework is added to the test tool so models can also run in co-simulation. For the addition of a co-simulation framework, different frameworks are researched. There are a three options:

- CoHLA
- TERRA
- DEIMOS

CoHLA (Nägele, 2020b) is a framework where cyber-physical systems can be simulated using FMI (Functional Mock-up Interface). CoHLA supports every simulator which can export models to FMI. CoHLA has a number of advantages, including the ability to run on Windows, Linux and Mac; the documentation is complete (Nägele, 2020a), also it is fast and supports multiple modelling tools. But the main advantage is that the tool is easily maintained and extendible. Which are two of the main guidelines of a test.

TERRA is a framework for model-driven development (Bezemer, 2013) which supports co-simulation. The models can be designed using blocks and after that can be translated to code. It uses LUNA (Bezemer et al., 2011) for model-to-code translation. TERRA supports models from other simulators (e.g. 20-sim) and uses FMI (just as CoHLA) to simulate these models (Kok, 2016).

Deimos is a framework which supports co-simulation (Zandberg, 2020). It is a simulator in which the cyber-physical system can be designed and simulated. It does not support models of other simulators (e.g. 20-sim), it does support FMU's but is limited to SDF's (Synchronous Data Flow).

The frameworks are filled in table 3.1 to give an overview.

	Co-simulation	Simulator support	OS	Documentation
CoHLA	✓	✓	Windows, Linux, MacOS	✓
TERRA	✓	✓	Linux	✗
Deimos	✓	✗/✓	Linux, MacOS	✓

Table 3.1: The comparison of CoHLA, TERRA and Deimos

The three frameworks all have their similarities and limitations. All three support co-simulation, this means that all three are suitable for addition to the test tool. The limitations of TERRA are, because it uses LUNA, the limitation in operating system, it only works on Linux. The second limitation is the documentation, there is insufficient documentation available so TERRA is not selected. Deimos has the limitation that is only supports its own models. As can be seen CoHLA is the best choice, it is supported on every OS and it supports FMI, which means it supports every simulator which can export to FMU.

3.6 Requirements

This Chapter describes the requirements. First of all the requirements by Jansen (2019) still hold. This is because the test tool is expanded, the current functionalities must still work.

The requirements by Jansen (2019) are:

1. The tool must combine simulation data with the defined tests to produce answers.
2. The tool must be modular.
3. The testing must be automated.
4. The tests must be reusable.
5. The test definitions must be Gherkin-style inspired.
6. The test definitions must support boolean equations.
7. The test definitions must support LTL formulas.
8. The test definitions must support model variables.
9. The tool should support multiple simulators.

Regarding the requirements for the expansion of the test tool, the conclusion is that the test tool must conform to the guidelines which are stated in Chapter 3.1. The two main adjustments to the test tool are the addition of step definitions and the change so that the addition of simulator is simplified. The addition of the step definitions is also to make the addition of submodel-awareness simpler.

As last the framework CoHLA is added to the test tool, because the addition of a simulator is made simpler this consists of only the addition of a interface-file for CoHLA. As CoHLA is a different framework and does not work with external interface. Point 13 in the following list determines that the basic functions are supported which means a simple simulation will be supported. It is not expected that every function of CoHLA will be implemented because of time limitations, this is why point 14 in the following list is a wish.

The additional requirements:

10. The guidelines of Micallef and Colombo (2015):
    - 10.1. The DSL should be simple, people should be quick in understanding the language.
    - 10.2. The DSL should exhibit similarity to another language, most of the times this is English.
    - 10.3. The DSL should be highly domain specific and parsimonious. This is so that any notions not related to the domain in question are omitted. Yet the designers should strive to make the language as complete as possible.
    - 10.4. The DSL should be easily extensible and reusable. It should be easy to add features to the language. It should also be reusable in that the same grammar can be used as the base of a new language when required, which includes:
        - 10.4.1. Addition of a function must be easy.
        - 10.4.2. Addition of a simulator must be relatively easy.
11. The test tool must support submodel-awareness testing.
12. CoHLA will be added as a co-simulation framework.

13. The test tool will support the basic function of CoHLA.
14. The different functions for CoHLA should be implemented.

4 Design and implementation

In this Chapter the design of the step definitions, the simplification for the addition of simulators and the implementation of CoHLA is discussed. First the design of the step definitions is explained, after that the addition of submodel-awareness is explained then the simplification of the addition of simulators is explained and then the implementation of CoHLA is explained.

The design of the addition of submodel-awareness is done after the design of the step definitions. The reason for this is because the submodel-awareness is being designed using step definitions.

4.1 Step definitions

The implementation of step definitions has different design challenges. First the addition to the existing grammar is explained, how the step definition is going to be addressed. After that the matching of the step definition, the design of the step definition in the code are explained. As last the automated import and the data-flow are explained.

4.1.1 Addition to the grammar

To add the step definition to the test tool, the grammar needs to be extended. As the step definitions supports both settings and constraints, one keyword is added. The keyword chosen for the step definitions is `step:`. This is to make it clear that what follows is a step, the step is designed to be between quotes. Another reason for the keyword is so the step can be recognized by the test tool. The addition to the grammar are line 13, 22 and 36 in the grammar file, which is located in appendix A.

The step definitions for a setting and a constraint are shown in Listing 4.1.

```
step: "I set a setting"
Then step: "I check the results with a constraint"
```

Listing 4.1: Definition of how the steps can be defined in a test

The reason that the constraint step needs a `Then` is because this is the rule of the grammar. All the constraints have the keyword `Then` in front. The additional reasons why `Then` is kept before the constraint is because it is easier to parse and by keeping the keyword the old tests (with the LTL-equations) are still compatible.

4.1.2 Match the step definition

With the adjustment of the grammar, the step definition can be recognized in the test tool but the step itself is not matched to anything and therefore no information can yet be taken from the step. The matching of the step is done using Regular Expression (RegEx)(Friedl, 2006). Using RegEx it is possible to match a step where some parts of the step can be different. A short example is the match of a step with a (variable) number. This number is an integer, this means that with Regex it can be matched with `\d+`. The step can be `The number is 6`, to match it, the step definition is `The number is \d+`, this way the number can vary but the step will always match. This can also be implemented for textual patterns.

Information needs to be extracted from the step as well: the value, the variable name and the path. This can also be done using RegEx. To get the number out of the example step earlier this Chapter, all integers are found using `d+`. As the values given to the step definition are also decimal a RegEx notation was formed to extract every possible number. This RegEx notation is stated in Listing 4.2. This RegEx notation also allows for numbers like 5, +0.6, −6.8 and $24e-9$.

```
[-+]?(?:(?:\d*\.\d+)|(?:\d+\.?))(?:[Ee][+-]?\d+)?
```

Listing 4.2: RegEx notation to find all values

The variable name and path also need to be extracted from the step, but unfortunately RegEx cannot be used for this application. This is because there are different variations for the variable name and path. For every new variable name and path a new RegEx needs to be configured. To allow the variable name and path to be extracted a solution has been proposed. Both the variable name and the path are between special characters. Then every other character is accepted between these special characters. For the variable name curly brackets were chosen, for the path brackets. Neither of these characters occurs in the grammar which means they are used by the step definitions. In Listing 4.3 the syntax of the variable name and the path is shown.

```
step: "I declare a variable: {variable}"
step: "I declare a path: [path/to/folder/or/files]"
```

Listing 4.3: Example of a step with a variable name and path

4.1.3 Decorators, constraints and settings and the execution of the function

When the step is matched to the step definition, the function connected to that step needs to be executed. For this decorators are used. Decorators also accept arguments to the function. This is necessary because the function corresponding to the step definition requires the step to get the information.

As is stated in Section 4.1.1, the setting and constraint step definitions are separated. This means the decorators are also separated for settings and constraints. The decorators here are defined with the step definition as an argument. An example of a decorator with step definition and function is shown in Listing 4.4. The decorator is `STEP.stepSet`, this example is a setting, a constraint has the decorator `STEP.stepCon`. When the step definition is matched the function `step` is executed.

```
@STEP.stepSet("Here is the step definition")
def step(step,interface):
  code to execute
  return interface.functionofsimulator
```

Listing 4.4: Example of a decorator with step definition and function

As is in Listing 4.4 the function needs arguments to get the different variables out of the step. For this three functions are defined: `get_name` to get the variable name, `get_number` to get the number and `get_path` to get the path. These functions accept the step and provide the information (if there are multiple variable names, all are returned in a list). Also if the function needs to execute a function of the simulator, the simulator also needs to be given to the function, hence the `interface`. When the step definition is a constraint, the function does not need the simulator, as the simulation has already been done. Instead it requires the results, this is why the constraint-function also needs the results of the simulation. An example of a constraint-decorator with function is shown in Listing 4.5. The function needs the step to get the variable name, values of the variable and the results to check this variable.

```
@STEP.stepCon("Here is the step definition")
def step(step,results):
  code to check results
  return True or False, depending on check
```

Listing 4.5: Example of a decorator with step definition and function

4.1.4 Automated import

The user can add a step to an existing step file, but it is also possible to add a new step file. This is useful for overview when a lot of different steps are added. To make it easier for the user to add a step file an automated import function has been implemented. Without this function the user would need to import the new step file by hand (i.e. adding a piece of code to the mainStep file). This is an extra action which is not preferred.

Only the step files which are in the folder `stepdefinitions` are imported by the automated import-function. The import is done at start-up of the test tool. The textual info in the terminal at start-up tells the user which files are not imported. If a step file is not imported and the test uses a step out of that step file, then test tool gives an error-message that the step cannot be found.

4.1.5 Change of data-flow

The dataflow of the test tool is changed to make sure the step definitions can control the simulator. Previously only the MainControl-file (which is the Control-block in the dataflow-figure 4.2) could control the simulator. With the addition of the step definitions, it is necessary that the step definitions also controls the simulator. This is because otherwise every function used in the steps also needs to be declared in the mainControl-file. As this is not desired (because every extra step makes the addition of a function harder) the choice was made to let the steps also control the simulators. This lead to the decision to let the mainControl be the only file which controls the step files. The old and new data-flow is presented in figure 4.1 and figure 4.2.

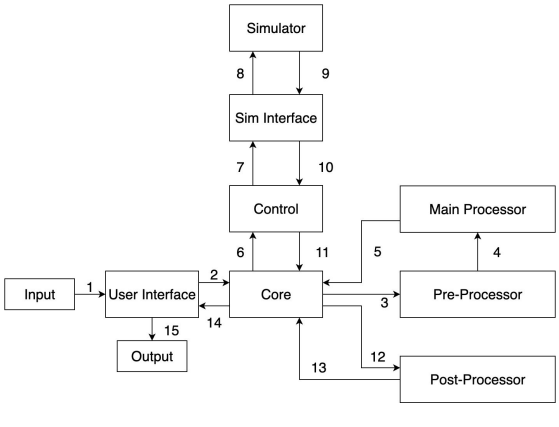


Figure 4.1: The old dataflow

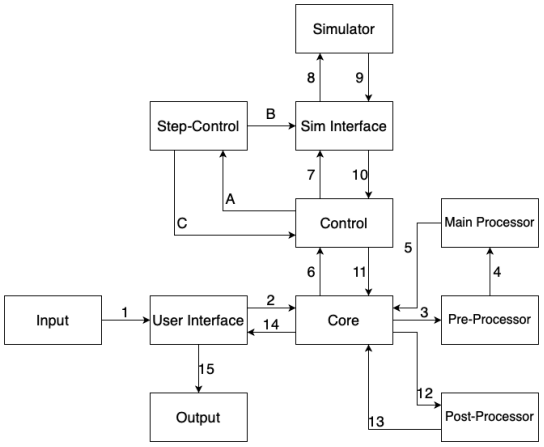


Figure 4.2: The new dataflow

What happens when a setting step is set is as follows: the Control-block recognizes there is a settingstep, this step is send to the Step-Control (line A). The Step-Control matches the step and executes the corresponding function in the block Sim-Interface (line B). When all the settings are verified and set, the Control-block starts the simulation.

The constraint check is done in the Post-Processor block, but this is not possible for the constraintsteps as the Post-Processor has no connection to the Step-Control-block. The reason for this is that the only block which has a connection to the Step-Control-block is the Control-block. The LTL-equations are all checked in the Post-Processor: this functionality has not been changed. The stepconstraint is checked in the Step-Control block. When the result has returned, the Control-block does some processing with the results. Here the steps are also checked. Via line C in figure 4.2 the result of the constraintstep is returned and is put in the

results. This all is sent through to the Core and Post-Processor. There the LTL-equations (if available) are checked and the results of the constraintsteps and LTL-equations are shown.

4.1.6 Step definition conclusion

Where at first it took five steps to add a new function to the test tool, this is reduced to two. First the step definition of the new step needs to be added to the step definitions, this can be done in a new step file or in an existing one. Then the new functionality needs to be added to the interface-file. This functionality needs to be called in the step definition.

4.2 Submodel-aware testing

In this Chapter the step which adds submodel-awareness is designed. This step replaces one submodel with another. The submodel-aware step is first implemented for 20-sim which is given in Listing 4.6.

```
def replaceSubmodel(submodel, submodelrep):
    return my20sim.replace_submodel(submodel, submodelrep)
```

Listing 4.6: Example of a defined step

20-sim has a `replace_submodel` function (which are shown on the controllab package-page of ControllabProducts (2015)) which connects to the step. This function in Listing 4.6 needs two variables: the name of the sub-model which needs to be replaced and the path of the sub-model which needs to be imported. In Listing 4.7 is the designed step definition, which was designed in Chapter 3.3.

```
Now I replace \{(.*?)\} with \[(.*?)\]
```

Listing 4.7: Example of a defined step

The full step definition with decorator is shown in Listing 4.8

```
@STEP.stepSet("Now I replace \{(.*?)\} with \[(.*?)\]")
def step(stepinput,interface):
  submodelold = STEP.getName(stepinput)[0]
  submodelnew = STEP.getPath(stepinput)[0]
  return interface.replaceSubmodel(submodelold, submodelnew)
```

Listing 4.8: Example of a defined step

The variable name is the name of the sub-model in the given model for the test. The sub-model with which the sub-model in the model is swapped needs to be indicated with the path to the sub-model, hence the brackets. An example for this step within a test is shown in Listing 4.9

```
step: "Now I replace {PWM} with [C:\Program Files (x86)\20-sim 4.7\
    Models\Library\Signal\Block Diagram Non-Linear\SignalLimiter-Limit.
    emx]
```

Listing 4.9: Example of a defined step

Here the PWM-submodel is replaced with a signal limiter.

4.3 Simulators

If the user wants to add a new simulator to the test tool, four steps needed to be taken (see Section 2.2.2). This is reduced to one step: the addition of a interface-file. The grammar is changed so it accepts all text. Previously it only accepted 20-sim and V-REP. As is presented in the grammar in appendix A, line 24 is changed to `TEXT`, this does mean the simulator names now also need to be between quotes. The grammar is changed so that every text behind the

`in` keyword is accepted. Before, the name would be checked in the tool, but the addition of a simulator is only the addition of the interface file. This is why the simulator check is done during runtime. The check is done using an extra function which is added to the interface-file. This function is the `getName` function, which returns the name of the simulator.

If an additional simulator is added, the interface-file needs to be imported in the test tool. To omit this part, the automatic import discussed in Section 4.1.4 is implemented for the simulators. The automatic import is changed a bit, as the interface-object needs to be saved so the Control file and the Step-control file are able to call it. The object is saved in a dictionary with the name of the interface (returned by the `getName` function). This dictionary is returned to the Control-file, where as the test has started the simulator can be selected and can be controlled. If a simulator file has not been imported a warning message (see Listing 4.10) appears in the terminal, like it does when a step file could not be imported.

```
vrepinterface could not be imported
```

Listing 4.10: The error message in the terminal when a simulator could not be imported

4.4 Implementation of CoHLA

In this Chapter the implementation of CoHLA is discussed. Because a CoHLA test is a command line tool, the implementation of CoHLA works a bit differently than the previously discussed 20-sim implementation, which is controlled via an external interface. The execution of a CoHLA test is explained in the first section. Here the three main functions are also explained, which are the bare minimum to let a CoHLA test execute. In the sections after the additional functions are explained and as last the functions which are not implemented are explained, as is the reason why those are not implemented.

4.4.1 The main functions

CoHLA works differently than the simulators which are currently integrated in the test tool. CoHLA is a command line tool. This means that a CoHLA test is executed with a single command.

1. First is the connect and open function, for CoHLA this function only needs to connect. CoHLA is a command line tool and this means no external interface needs to be opened. This function makes sure the simulator can be controlled and checks whether a connection is possible. If the connection fails or the simulator cannot be controlled, the test tool stops the simulation. To make sure CoHLA can be controlled the help-command is executed and the test tool expects CoHLA to return the help-info. This means that the command `run.py -h` is executed. The response should include `=== GENERIC FLAGS ===`, if it does not the test stops the simulation. Here the check to determine which operating system is used is also done. This is because the commands are different for Windows than for OSX (and Linux). This check makes sure the commands in the rest of the simulator file is set for the specific operating system.

2. The second implementation is the run function, here the command to run the test is constructed. Settings (if set) are added to the command. Then the command is executed and the test is done. The run-function must give back the results of the test, for the constraints. After the test, CoHLA puts the result in a CSV file in a separate folder. To import the CSV and put it into a dictionary (required format) the function `csvToDict` is made.

3. The last function imports the CSV file out of the folder, the name of this folder is the date and time of the simulation. This name is filtered out of the terminal-output given by CoHLA. Then the CSV file is imported and put in a dictionary. The name of the csv file

can be specified in the test, this is done because CoHLA can give multiple csv files, but the program can only handle one csv file. If only one csv file is generated it is not necessary to specify the name of the csv file, if there are more csv files generated it is necessary to specify the name. The step to specify the csv file is in Listing 4.11.

```
I use logger: \{(.*?)\}
```

Listing 4.11: Step for the specification of the csv file

After the csv file is imported it is converted to a dictionary and returned to the Control-block, then the constraints are being checked.

4.4.2 Addition of a scenario or situation file

It is possible to add a scenario or situation file to the test, either can be specified with the steps in Listing 4.12. This is added to the command made in the run-function.

```
I use the situation which is located at: \[(.*?)\]
I use the scenario which is located at: \[(.*?)\]
```

Listing 4.12: Example of a step how the function is defined

4.4.3 Setting of a variable

The test tool also has a function `setValue`, this function is implemented for CoHLA. CoHLA uses situation files for settings of variables. The design for implementation is as follows, if there is no situation file defined a new one is made in a temporary folder (which the user can delete afterwards) where the setting is set. When there is a situation file defined, the program checks if the variable is already in the file. If it is, the line is deleted and the new value with variable is added. If it is not already in the file the variable and value is added. The variable name is the instance with the variable name. So if the name of the federate is `INS` and the variable name is `SIGNALA` then the name which must be put in the step is `INS:SIGNALA`.

4.4.4 Not implemented functions

Not all the functionalities of the test tool could be implemented. This is due to various reasons, where the main reason is time-related. There was no time left to implement the functionality for submodel-awareness and the setting of the simulation time.

Another reason for not implementing a functionality is that it is simply not possible. It is not possible to implement the functionality `close`. This functionality closes the external interface after the simulation, CoHLA does not have an external interface so it is not possible to close it.

4.5 Conclusion

The improvements made by adding step definitions and simplifying the addition of a simulator had immediate benefits. Only an interface file addition was necessary to support CoHLA with the test tool. CoHLA needed a couple of additional functionalities (naming the loggerfile, including the situation or scenario file) and with the implementation of the step definitions this is done quick and easily.

5 Results

The three goals which were determined in Chapter 1.2 are the implementation of submodel-awareness, the improvement of the flexibility and the implementation of CoHLA. The second goal, the improvement of the flexibility, is realized by the addition of step definitions. In the following Sections different demonstrations are done to show that these goals are achieved. The submodel-awareness is implemented using the step definitions and the demonstration is done using the tests out of Broenink et al. (2020).

At last the demonstration with CoHLA is done. For this a model is made, the model has one input and one output. The model inverts this number and multiplies it with a number. This number can be set in the .cohla file itself or can be changed by the test.

5.1 Results of the step definition

Different demonstrations are shown in this Chapter to see the workings of the step definitions. First the addition of a setting is shown, and after that the addition of submodel-awareness. Then the addition of a constraint step is shown and as last the error-messages in case a mistake is made in the step.

5.1.1 Addition of a submodel-aware settingstep

This function needs two variables: the name of the submodel in the given model and the path of the submodel which needs to replace the submodel. The defined step with the function can be seen in Listing 5.1.

```
@STEP.stepSet("Now I replace \{(.*?)\} with \[(.*?)\]")
def step(stepinput,interface):
  submodelold = STEP.getName(stepinput)[0]
  submodelnew = STEP.getPath(stepinput)[0]
  return interface.replaceSubmodel(submodelold, submodelnew)
```

Listing 5.1: Example of a defined step

The function `replaceSubmodel` is linked to the function `replace_submodel` in the 20-sim interface file. The name and the path-variables are returned from the `STEP.getName`- and `STEP.getPath`-functions, as was explained in Section 4.1.3.

To make sure the addition of the submodel-aware settingsstep is working properly a test is run. This test is also done by Broenink et al. (2020). The test tests the torque of DC-motors with different parameters. In the test of Broenink et al. (2020) two models are used, where the only difference is the parameters of the two motors. The two testscripts used are shown in Listing 5.2 and Listing 5.3.

```
Feature "Maximum Torque"
  Scenario "Constant voltage"
    Given "TorquetestA.emx" in 20-sim for 2 seconds
    with "CurrentSource.I"=3.125
    Then F("DCmotor.p2.T">0.58)
```

Listing 5.2: Feature one of the motor test

```
Feature "Maximum torque"
  Scenario "Constant current"
    Given "TorquetestC.emx" in 20-sim for 2 seconds
    with "CurrentSource.I"=3.125
    Then F("DCmotor.p2.T">0.58)
```

Listing 5.3: Feature two of the motor test

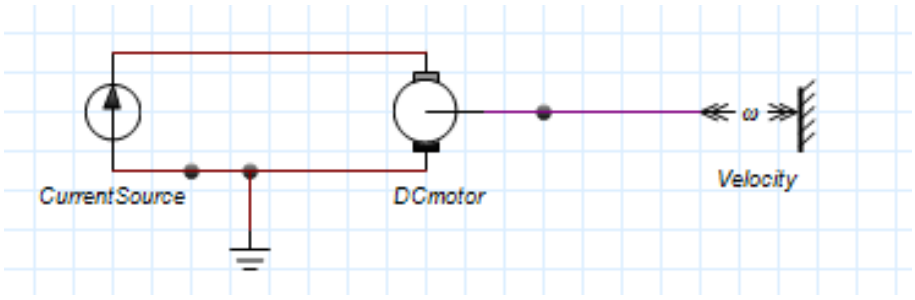


Figure 5.1: The model of Broenink et al. (2020)

In figure 5.1 the model can be seen which was tested in the test of Broenink et al. (2020). The overall model is the same, only the submodel motor (part `DC-motor`) had different parameters for the two different tests. For this two models were made. With the addition of the step the test can be done on one model and the test can replace the one motor for one scenario. This test is in Listing 5.4, only one model is given, and in scenario `motorCtest` the motor (`DCmotor`) is replaced by the motor with different parameters (`DCMotorC`).

```
Feature "Motortestfeature"

  Scenario "motorAtest"
  Given "Z:\models\TorquetestA.emx" in "20-sim"
  with "CurrentSource.I"=3.125
  Then F("DCmotor.p2.T">0.58)

  Scenario "motorCtest"
  Given "Z:\models\TorquetestA.emx" in "20-sim"
  step: "Now I replace {DCmotor} with [Z:\models\DCMotorC.emx]"
  with "CurrentSource.I"=3.125
  Then F("DCmotor.p2.T">0.58)
```

Listing 5.4: Test with both motor tests where submodel-awareness is applied function

The first scenario uses the motor which is provided in the model, the second scenario uses a different motor, which is imported from `Z:\models\DCMotorC.emx`. The output of the test in Listing 5.4 is as is stated in Listing 5.5.

```
Motortest.test : False
  Feature Motortestfeature : False
    Scenario motorAtest : False
      Constraint F("DCmotor.p2.T">0.58) : False
        Equation "DCmotor.p2.T">0.58 : False
    Scenario motorCtest : True
      Constraint F("DCmotor.p2.T">0.58) : True
        Equation "DCmotor.p2.T">0.58 : True
```

Listing 5.5: Output of the test of Listing 5.4

Different models can be tested with different submodels without losing the parameters. When the test is completed the models return to the state they were before the tests. This is exactly what is wanted, as submodels can now be replaced without changing the saved model-file.

5.1.2 Addition of a constraint step

A constraint step can also be defined, this is a step which checks the results of the test. The added step checks whether the results do not exceed a certain value. This step and function are defined in Listing 5.6.

```
@STEP.stepCon("The \{(.*?)\} should not be more than [-+]?(?:(?:\d*\.\d
    +)|(?:\d+\.?))(?:[Ee][+-]?\d+)?")
def step(step_input,result):
    var = STEP.getName(step_input)
    var1 = STEP.getNumber(step_input)
    resultlist = result[var[0]]
    resultcon = all(i <= var1[0] for i in resultlist)
    return resultcon
```

Listing 5.6: Definition of the constraint step

In Listing 5.7 is the test with the constraint out of Listing 5.6. The output of the PWM-signal should not exceed 2.5.

```
Feature "TestconstraintStep"

Scenario "Test1"
  Given "C:\Program Files (x86)\20-sim 4.7\Models\Examples\Electric
      Motors\Components\PWM Conversion.emx" in "20-sim"
  with "PWM.f" = 100
  for 0.11 seconds
  Then step: "The {PWM.output} should not be more than 2.5"
```

Listing 5.7: Test of the constraintstep

```
Test1scen.test : True
Feature TestconstraintStep : True
  Scenario Test1 : True
    Step step:"The {PWM.output} should not be more than 2.5" : True
```

Listing 5.8: Result of the test of the constraintstep

The result of test of Listing 5.7 is in Listing 5.8. The output is `True`, to check this the simulation has been paused when the results came back. This list of values did not contain any value which was more than 2.5. The constraint is working like it should be.

5.1.3 Error-messages

If a step is not defined correctly in the test, such that there is no matching step definition, an error-message appears in the terminal. This error-message is shown in Listing 5.9. The test of Listing 5.4 is used, only the step `"Now I replace {DCmotor} with [Z:\models\DCMotorC.emx]"` is changed into `"Now I replaceeee {DCmotor} with [Z:\models\DCMotorC.emx]"`.

```
Settingstep not found
Now I replaceeee {DCmotor} with [Z:\models\DCMotorC.emx]
An error occurred in the simulation.
Returned data is incorrect.
```

Listing 5.9: Error message when the test has an incorrect step syntax

As can be seen the error message also prints the step which cannot be found, this is so the user can check quickly what is wrong with the step. When the step is fixed the test executes without any problems.

5.2 Co-simulation/CoHLA

For CoHLA not all the functions are integrated in the test tool. The three functions which are integrated are (as can be seen in Chapter 4.4): Change value, add scenario, add situation. For these three functions a test is written, this test is shown in Listing 5.10.

```
Feature "testCoHLA"

Scenario "Noscenarionosituation"
  Given "/Users/hanna/eclipse-workspace/PWM/src-gen/PWM/conf/
      multiplierblock.topo" in "CoHLA"
  step: "I use logger: {Logger}"
  step: "I set a variable: {GEN.Modulator} to -50"
  Then step: "The {GENOutput} should not be more than 20"
  Then step: "The {GENOutput} should be more than -9"


Scenario "situation"
  Given "/Users/hanna/eclipse-workspace/PWM/src-gen/PWM/conf/
      multiplierblock.topo" in "CoHLA"
  step: "I use the situation which is located at: [conf/Multiplierblock/
      Standard.situation]"
  step: "I set a variable: {GEN.Modulator} to -50"
  Then max("GENOutput") == 0.0

Scenario "scenario"
  Given "/Users/hanna/eclipse-workspace/PWM/src-gen/PWM/conf/
      multiplierblock.topo" in "CoHLA"
  step: "I use logger: {Logger}"
  step: "I use the scenario which is located at: [conf/Multiplierblock/
      firstTest.scenario]"
  step: "I set a variable: {GEN.Modulator} to 1"
  Then step: "The {GENOutput} should not be more than 5"


Scenario "situationscenario"
  Given "/Users/hanna/eclipse-workspace/PWM/src-gen/PWM/conf/
      multiplierblock.topo" in "CoHLA"
  step: "I use logger: {Logger}"
  step: "I use the situation which is located at: [conf/Multiplierblock/
      Standard.situation]"
  step: "I use the scenario which is located at: [conf/Multiplierblock/
      firstTest.scenario]"
  step: "I set a variable: {GEN.Modulator} to 4"
  Then step: "The {GENOutput} should not be more than 21"
```

Listing 5.10: The test where the three different functions can be tested

```
DemoCoHLA.test : True
Feature testCoHLA : True
  Scenario Noscenarionosituation : True
    Step step:"The {GENOutput} should be more than -9" : True
    Step step:"The {GENOutput} should not be more than 20" : True
  Scenario situation : True
    Equation max("GENOutput")==0.0 : True
  Scenario scenario : True
    Step step:"The {GENOutput} should not be more than 5" : True
  Scenario situationscenario : True
    Step step:"The {GENOutput} should not be more than 21" : True
```

Listing 5.11: Results of the test of CoHLA

The result of the test in Listing 5.10 can be seen in Listing 5.11. For CoHLA `.csv`-files are generated, in these files the test-constraints can be checked. This has been done to check if the results are read in correctly and the test tool checks the result. This is the case. The results of the csv-files comply with the constraints given in the test. As can be seen in scenario `Noscenarionosituation` the function `setValue` is also being called. After the test the temporary situation file is checked and the value is what it was supposed to be: -50.

5.3 Automatic import

The automatic import is made so when extra step files or simulators are added, the user does not need to this to an import list. The automatic import is done at the start of the test tool, so when there are files which could not be imported, the file-names appear in the terminal. In Listing 5.12 this output is listed as it is in the terminal.

```
Importing the step files
Importing the simulator-interface files
example_interface could not be imported
[errormessage]
vrepinterface could not be imported
[errormessage]
```

Listing 5.12: Automatic import, feedback

The first interface, `example_interface`, is an example code-file and cannot compile thus also not import. In the second interface, `vrepinterface`, there is some faulty code which it also makes it unable to compile. All the step files are compiled and imported. When a step file cannot be imported, the output is as in Listing 5.13, here it is shown that the step file `set_steps` is not imported.

```
Importing the step files
set_steps has not been imported
[errormessage]
Importing the simulator-interface files
example_interface could not be imported
[errormessage]
vrepinterface could not be imported
[errormessage]
```

Listing 5.13: Automatic import, feedback when a step file is not imported

5.4 Conclusion

What can be concluded from the results is that the addition of the steps is an improvement, the addition of new functions and simulators is easier and takes less steps. The steps to add a new keyword and functionality is reduced from five to two steps. The in depth knowledge of the test tool is not needed any more, it is necessary to have (some) knowledge of RegEx and of Python. For the addition of a new functionality, knowledge of the simulator is needed to connect the new step definition to the functionality. The addition of the steps made it easy to add the submodel-aware function in the test tool, and more importantly it also works.

The addition of CoHLA is working and the results can be checked. All the tests of Jansen (2019) can still be executed, this can be seen in the test in Listing 5.4. A step is defined but also a setting and constraint as was defined by Jansen (2019), the test executes which means that all the old tests are still compatible with the new test tool.

6 Conclusion and recommendations

6.1 Conclusions

In the introduction (Chapter 1) three goals were determined:

1. The implementation of submodel-awareness
2. The improvement of the flexibility
3. The addition of co-simulation

The first goal is achieved, it is possible to replace sub-models in the test.

The second goal is achieved by the two modifications: the addition of the step definitions and the modification of the code to make the addition of new simulators simpler. The addition of the step definitions makes it simpler to add new functionalities and keep the test easy to read and understand. Instead of multiple (confusing) keywords steps can be used which are easy to understand.

The third goal is achieved. With the use of the modification of the code to make the addition of new simulators possible, the user only needs to add a new simulator file and connect the functions. This modification made it possible to easily add the co-simulation. This is because the framework, CoHLA, could be identified as a new simulator. So by adding CoHLA as a simulator, co-simulation is also added to the test tool, which means the third goal is achieved.

In Chapter 1 the requirements were determined. All the requirements listed are achieved. The first nine requirements are from the previous version of the test tool (Jansen, 2019):

1. The tool must combine simulation data with the defined tests to produce answers.
2. The tool must be modular.
3. The testing must be automated.
4. The tests must be reusable.
5. The test definitions must be Gherkin-style inspired.
6. The test definitions must support boolean equations.
7. The test definitions must support LTL formulas.
8. The test definitions must support model variables.
9. The tool should support multiple simulators.

These requirements still hold, the test tool supports these requirements. For the additional requirements stated in the analysis (Chapter 3.6):

10. The guidelines of Micallef and Colombo (2015):
    - 10.1. The DSL should be simple, people should be quick in understanding the language.
    - 10.2. The DSL should exhibit similarity to another language, most of the times this is English.
    - 10.3. The DSL should be highly domain specific and parsimonious. This is so that any notions not related to the domain in question are omitted. Yet the designers should strive to make the language as complete as possible.

10.4. The DSL should be easily extensible and reusable. It should be easy to add features to the language. It should also be reusable in that the same grammar can be used as the base of a new language when required, which includes:

10.4.1. Addition of a function must be easy

10.4.2. Addition of a simulator must be relatively easy

11. The test tool must support submodel-awareness testing

12. CoHLA will be added as a co-simulation framework

13. The test tool will support the basic function of CoHLA

14. The different functions for CoHLA should be implemented.

From the results, it is concluded that the guidelines of Micallef and Colombo (2015) are met. The first one, the DSL should be simple, point 10.1, is met because of the step-definitions. The tests are simple to understand. The second one, the DSL should exhibit similarity to another language, is also fulfilled. The steps are in English and therefore the DSL exhibits similarity to another language. The third guideline was already met, the test tool is highly domain specific and parsimonious, and it still is.

The fourth guideline, point 10.4 in the list, is met because of the step-definitions but also because of the alterations made for the simplification of the addition of a new simulator. Because of this the DSL is now easily extensible. It takes less effort to add new functionalities with corresponding step definitions. Only two steps need to be taken, instead of the previous five, and the steps take less knowledge and effort. Requirement 10.4.1 and 10.4.2 are met, so the four guidelines are met.

Requirement number 11 is fulfilled, the test tool is able to replace sub-models and thus can test with submodel-awareness.

As last the requirements about CoHLA. CoHLA is implemented in the test tool and the basic functions are supported. The last requirement is met halfway, it is possible to add a situation and/or scenario to the test but more functions of CoHLA should be implemented, for example the adjustment of a scenario, or the submodel-awareness. The implementation of these functions was not conducted due to time-constraints.

6.2 Recommendations

The test tool can be improved further in different aspects. The first recommendation is the last requirement of this project: *The different functions for CoHLA should be implemented.* Not all functions for CoHLA are implemented and it would be an improvement of the test-tool to add all the functions of CoHLA to the test-tool.

Another aspect to improve the test tool is usability, as an extension of the addition of the step definitions the test tool can be improved on the GUI and the visualization of the results. This includes the following points:

- An improvement could be the the upgrade of the GUI (Graphical User Interface). This could show the error messages which are now shown in the terminal. The layout can be improved, a large space in the GUI is not used. This could be used for the display of the error messages or a whole new design could be made.
- The first improvement is the results, the results of the test now are the three keywords `True`, `Sometimes True` or `False`. This could be expanded by adding a graph or table to the output.

- Another improvement is that if the result is `False` the differences between the wanted value and the real value is shown (if possible). For example the maximum value of a signal must be under a certain value, say 2, it might be useful to see what the maximum signal was so that when the signal is close to the maximum value (2.01) or not (5).
- The last improvement can be the addition of the path at the start of the test. This would result in smaller paths in the test itself and will make the test clearer.

A Grammarfile

```
grammar myStepsGrammar;
entry : feature scenario+;
feature: 'Feature' TEXT methods;
scenario : 'Scenario' TEXT (LPAR fVar (',' fVar)* RPAR)? given then+
        | imprt (LPAR fVar (',' fVar)* RPAR)?;
imprt : 'import' (PATH | TEXT);
methods : method*;
given : 'Given' model settings*;
settings : ('for' time) |
        ('with' initVars) |
        ('in' simulator) |
        ('include' (PATH | TEXT)) |
        ('step:' steptext);
time: expr 'nanoseconds' |
        expr 'microseconds' |
        expr 'milliseconds' |
        expr 'seconds' |
        expr 'minutes' |
        expr 'hours';
initVars: initVar (',' initVar)*;
initVar: TEXT '=' (expr | boolean);
steptext : TEXT;
boolean : 'False' | 'True' | 'true' | 'false';
simulator : TEXT;
then: 'Then' constraint;
model : PATH | TEXT;
fVar : expr;
constraint : LPAR constraint op constraint RPAR |
        tempSing LPAR constraint RPAR|
        tempDouble |
        CONSPATH |
        'not' constraint |
        constraint 'for' time |
        equation |
        methodCall |
        ('step:' steptext);
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' | 'weakuntil');
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;
```

```
55 callArgs : expr (',' expr)*;
56 method : 'method' ID LPAR args RPAR ':' constraint;
57 args: ID(',' ID)+;
58 fillVar : '&' NUMBER;
59 functionVariable : '%' ID;
60 expr : expr (PLUS | MINUS) multExpr | multExpr ;
61 multExpr : multExpr (TIMES | DIV) powExpr | powExpr ;
62 powExpr : powExpr POW atom | SQRT atom | atom;
63 atom : MINUS atom #negAtom |
64         LPAR atom RPAR #parAtom |
65         NUMBER #numAtom |
66         LPAR expr RPAR #exprPar |
67         fillVar #fVarAtom |
68         functionVariable #funcVariable |
69         TEXT #nameAtom
70         ;
71 LPAR : '(';
72 RPAR : ')';
73 PLUS: '+';
74 MINUS: '-';
75 DIV: '/';
76 TIMES: '*';
77 POW: '^';
78 SQRT: 'sqrt';
79 LT : '<';
80 LTE: '<=';
81 GT: '>';
82 GTE: '>=';
83 NOT: '!=';
84 EQ : '==';
85 EXPOINT: '!';
86 QUOTES: '"';
87 CONSPATH : QUOTES ('a' ..'z' | 'A'..'Z' | '0'..'9' | '(' | ')' | '.' |
88         '\\' | '/' | ':' | '-' | ' ' | '_' )+ '.constraint' QUOTES;
89 TEXT : QUOTES ('a' ..'z' | 'A'..'Z' | '0'..'9' | '(' | ')' | '.' |
90         '\\' | '/' | ':' | '-' | ' ' | '_' | '{' | '}' | '[' | ']')+
91         QUOTES;
92 PATH : QUOTES ('a' ..'z' | 'A'..'Z' | '0'..'9' | '(' | ')' | '.' |
93         '\\' | '/' | ':' | '-' | ' ' | '_' )+ QUOTES;
94 ID : ('a'..'z' | 'A'..'Z')+;
95 NUMBER : ('0'..'9')+ ('.' ('0'..'9')+)?;
96 COMMENT: '/*' .*? '*/' -> skip;
97 LINE_COMMENT: '//' ~[\r\n]* -> skip;
98 WS : [ \t\r\n]+ -> skip;
```

Bibliography

Bezemer, M. (2013), *Cyber-physical systems software development: way of working and tool suite*, Ph.D. thesis, University of Twente, Netherlands, doi:10.3990/1.9789036518796.

Bezemer, M. M., R. J. Wilterdink and J. F. Broenink (2011), LUNA: Hard Real-Time, Multi-Threaded, CSP-Capable Execution Framework., in *CPA*, pp. 157–175.

Bhatti, I., J. A. Siddiqi, A. Moiz and Z. A. Memon (2019), Towards Ad hoc Testing Technique Effectiveness in Software Testing Life Cycle, in *2019 2nd International Conference on Computing, Mathematics and Engineering Technologies (iCoMET)*, IEEE, pp. 1–6.

Broenink, T., B. Jansen and J. Broenink (2020), Tooling for automated testing of cyber-physical system models, *ICPS*.

ControllabProducts (2015), *20sim modeling and simulation*, https://www.20sim.nl (accessed August 10, 2020).

Cucumber (2019), *Gherkin Documentation*, https://cucumber.io/docs/gherkin/ (accessed March 10, 2020).

Friedl, J. E. (2006), *Mastering regular expressions*, " O'Reilly Media, Inc.", ISBN 9780596528126.

Guru99 (2014), *7 Principles of Software Testing*, https://www.guru99.com/software-testing-seven-principles.html (accessed August 10, 2020).

Jansen, B. (2019), *Automated Testing of Models of Cyber-Physical Systems*, Master's thesis, University of Twente, the Netherlands, 042RaM2019.

Kok, K. (2016), *TERRA support for architecture modeling*, Master's thesis, University of Twente, 040RaM2016.

Kumar, D. (2019), *Software Engineering, Seven Principles of software testing*, https://www.geeksforgeeks.org/software-engineering-seven-principles-of-software-testing/ (accessed August 10, 2020).

Micallef, M. and C. Colombo (2015), Lessons learnt from using DSLs for automated software testing, in *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, IEEE, pp. 1–6.

Muller, T. and D. Friedenberg (2011), Certified tester foundation level syllabus, *Journal of International Software Testing Qualifications Board*.

Myers, G. J., C. Sandler and T. Badgett (2011), *The art of software testing*, John Wiley & Sons, ISBN 9781118031964.

Nägele, T. (2020a), *20sim modeling and simulation*, https://cohla.nl/docs/ (accessed July 28, 2020).

Nägele, T. C. (2020b), *CoHLA: Rapid Co-simulation Construction*, Ph.D. thesis, Radboud University Nijmegen.

Nguyen, V. H., Y. Besanger, Q. T. Tran, T. L. Nguyen et al. (2017), On conceptual structuration and coupling methods of co-simulation frameworks in cyber-physical energy system validation, *Energies*.

Parr, T. (1989), *Antlr*, https://www.antlr.org (accessed May 13, 2020).

SM, R. (2016), *Seven Principles of Software Testing, Software Testing Material*, https://www.softwaretestingmaterial.com/principles-of-software-testing-2/ (accessed August 10, 2020).

Zandberg, K. (2020), *Dataflow-Based Model-Driven Engineering of Control Systems*, Master's thesis, University of Twente, 003RaM2020.