MASTER’S THESIS | LUND UNIVERSITY 2015

Developing a Test Authoring Tool for a Modeling Language

Victor Johnsson, Anders Tilly

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2015-12

Developing a Test Authoring Tool for a Modeling Language

Anders Tilly
ada09ati@student.lu.se

Victor Johnsson
ada10vjo@student.lu.se

June 7, 2015

Master's thesis work carried out at Lund University and Modelon AB.

Supervisors: Jesper Öqvist, jesper.oqvist@cs.lth.se
Jon Sten, jon.sten@modelon.com

Examiner: Görel Hedin, gorel.hedin@cs.lth.se

Abstract

For the modeling language Modelica, there is no standard for the testing of Modelica models. In this thesis we investigate a new way to create and run tests by developing an external test authoring tool. Our tool was developed by first examining existing testing practices, and then trying to improve upon those by iteratively developing our tool. This was done while continuously performing user studies with the users and our tool. The result is a functional tool capable of creating and running tests.

Keywords: Testing, Test Authoring, User Interface, User Studies, Modeling Language, Modelica

Acknowledgements

We would like to thank Jon Sten and Jesper Öqvist for being supervisors and Görel Hedin for being the examiner. We would also like to thank the modelers at Modelon for participating in our user studies and providing valuable feedback.

Contents

1 Introduction **9**
1.1 Contribution Statement 10

2 Background **11**
2.1 Modelica 12
2.1.1 Modelica Testing 12
2.1.2 JModelica.org and Optimica Compiler Toolkit 13
2.1.3 Regtest and Dymola 13
2.2 Testing Frameworks 13
2.2.1 Optimica Testing Toolkit 14
2.3 Eclipse 14
2.4 Usability 15

3 Implementation **17**
3.1 Framework and Language Selection 17
3.1.1 Python 18
3.1.2 Java 19
3.2 First Iteration 19
3.2.1 Task Analysis 20
3.2.2 Paper Prototype 22
3.3 Second Iteration 23
3.3.1 Mock-up Interface 23
3.4 Third Iteration 24
3.4.1 Feedback Response 25
3.4.2 Result Editor 26
3.4.3 OCT Integration 27
3.4.4 OTT Integration 27
3.4.5 Test Representation 28
3.4.6 Eclipse Run Configuration 29

3.5 Fourth Iteration 29
3.5.1 Feedback Response 30
3.6 Task Time Measurement 33

4 Evaluation **35**

5 Related Work **37**
5.1 UnitTesting 37
5.2 MoUnit 37
5.3 Eclipse Plugins 38

6 Conclusion **39**
6.1 Future Work 39

Bibliography **41**

Acronyms

API Application Programming Interface.

FMI Functional Mock-up Interface.

GUI Graphical User Interface.

HTML HyperText Markup Language.

IDE Integrated Development Environment.

JNI Java Native Interface.

OCT Optimica Compiler Toolkit.

OTT Optimica Testing Toolkit.

PDE Plugin Development Environment.

RCP Rich Client Platform.

XML Extensible Markup Language.

Chapter 1

Introduction

Modelica is a modeling language used for the modeling of physical systems, and Modelica models are used for simulating the systems. To test a Modelica model is to compare its simulation results to some desired behaviour. For some of the Modelica modelers at Modelon AB there was dissatisfaction with how they tested Modelica models. At the time of this project, there was no standard for the testing of a model, and the existing process at Modelon was considered unintuitive and unwieldy. There was a need for a more streamlined and user-friendly way of authoring and configuring tests. The tests were created programmatically which introduced certain errors, for example compile errors. Compile errors could be avoided with a user interface that does not allow the user to input incorrect values. Creating tests programmatically also makes test authoring more difficult than it needs to be, especially for a user without a programming background. To create a test authoring tool with this kind of user interface, there were a number of problems to consider:

- Current testing process: The current testing process should be studied to identify potential problems with it, so that improvements for them can be developed.
- Language and graphical framework: What programming language should the tool be developed in, and what graphical framework should it use?
- Usability: How can the usability of the program be ensured to be good?
- Model information: How will all necessary information from the models be obtained?
- Test authoring: How should the tests be created and saved?
- Test configuration: How should the tests be configured and run?

After examining which language and graphical framework to use, we decided to develop the tool in Java as an Eclipse plug-in. The Optimica Compiler Toolkit (OCT) was used to extract information from the models to be shown in our tool. The tests were saved as a combination of Extensible Markup Language (XML) files and Eclipse run configurations. The verification of simulation results was done by using Optimica Testing Toolkit (OTT).

Through iterative work and user studies, the tool was continuously improved upon in an effort to create a simple and user-friendly way of testing models. Every iteration started with obtaining feedback and then responding to that feedback by making changes in our tool. The feedback was collected using methods described by Lauesen [1]. What changes should be made was decided by reasoning about the usability gain contra time to implement. In the last part of the implementation we did a more extensive evaluation based on the six usability factors, as defined by Lauesen [1].

This master's thesis was carried out at Modelon in Lund. Modelon provides model- and system-based solutions centered around Modelica and Functional Mock-up Interface (FMI) [2]. Our tool is intended to be integrated into OTT and used in model development at Modelon.

In the background chapter we describe important concepts for this thesis. In the implementation chapter we follow the implementation process from beginning to end. In the evaluation chapter we do an extensive evaluation of the six usability factors. In the related work chapter we look at other similar projects and the difference between what we have done and they have done. Finally, in the conclusion chapter we summarize and reflect on our results based on what we set out to do.

1.1 Contribution Statement

In this project, we had collective code ownership so most of the code has been worked on by both authors. However, some specific things were given focus to one author. Anders Tilly focused on OTT extension, compiler integration and undo/redo features. Victor Johnsson focused on run configuration, HyperText Markup Language (HTML) report modification and test inheritance. Both authors worked on the report.

Chapter 2

Background

When building a complex physical system, there is usually a desire to ensure that the system will work as expected before it is released. One way to do this is to test the system using software, before the development of a physical prototype. Often a model of the system is developed in order to simulate the system.

The benefit of using software to test the system is that it is very easy to make changes in the model if the system is not behaving as expected. This decreases the time for correcting errors and thus decreases the time for the product to enter the market. To make the software testing as efficient as possible the tests are automated [3]. By having automated tests it is easier to continuously test the model when adding new functionality to it. This process is called regression testing. Regression testing is important since future development can create errors in unexpected parts of the program, and by having regression testing those errors can be found.

In software development, a test is usually run and checked towards an expected result [4]. However, for modeling languages, it can be different. When talking about testing a model there are three aspects to consider, if it: (a) can be translated and simulated without error, (b) delivers the expected results, and (c) represents reality adequately [5]. For aspects b and c, there is a reference value that is considered to be the "correct" value. The result of a test is checked to be within a specific tolerance of that value. If the modeler deems the new value to be better than the reference value, the modeler may choose to overwrite the old reference value and use the new value as future reference.

2.1 Modelica

Modelica is a modeling language targeted at the modeling of complex physical heterogeneous systems. This means that Modelica supports multiple physical domains and a mix of continuous and discrete components. Modelica is suited for a wide array of applications, such as automotive systems, power plants, thermo-fluid systems and robotics [6]. Development using Modelica can be done with a visual editor, such as Dymola. By being able to construct the model visually, the developer can focus on the design of the model and let the software translate it to Modelica.

A Modelica model contains variables and equations. A variable can have one of four levels of variability. Starting from the least variable, they are: constant, parameter, discrete-time and continuous-time [7]. The value of a variable with constant variability is considered constant during compilation and simulation. The value of a variable with parameter variability is considered unknown during compilation and constant during simulation. The value of a variable with discrete-time variability is changed based on events. The value of a variable with continuous-time variability is changed based on its time derivative [8]. In our tool, we eventually decided to call a variable with constant or parameter variability "modifier" while referring to a variable with continuous or discrete variability as "variable".

2.1.1 Modelica Testing

Testing consists of test authoring, test configuration and test execution. Test authoring refers to the creation of tests. Test configuration refers to choosing the appropriate settings for the test, such as which compiler to use. Test execution refers to running the tests.

Models can be created and represented both textually and graphically. Using a graphical user interface is sometimes more efficient than using a programmatic approach [9]. Some aspects of the test authoring on the other hand can only be done programmatically.

The Modelica language contains the concept of annotations for storing meta information about the model. Examples of such information are: graphics, documentation and versioning [10]. There are two types of annotations that are used for testing:

- `experiment`: The experiment annotation indicates that the model can be simulated and it also provides simulation settings, such as start or stop time [10]. See listing 2.1 to see an example of this annotation.
- `TestCase`: The TestCase annotation extends the experiment annotations and specifies additional information, such as whether the test should pass or fail [11]. See listing 2.1 to see an example of this annotation.

```
model SimpleDeclaration
  extends Icons.TestCase;
  Real x = 3;
  Real y = x;
  annotation(
    __ModelicaAssociation(TestCase(shouldPass=true)),
    experiment(StopTime=0.01),
    Documentation(
      info="<html>Tests simple component declarations.</html>"));
end SimpleDeclaration;
```

Listing 2.1: An example of TestCase and experiment annotations. This example is taken from the Modelica Compliance Library Guide [11].

2.1.2 JModelica.org and Optimica Compiler Toolkit

JModelica.org is an open source platform for compilation, optimization, simulation and analysis of Modelica models. It was created in 2007 as part of a PhD thesis by Johan Åkesson [12]. At the time of writing it is being maintained by Modelon AB.

One part of the JModelica.org platform is the JModelica.org compiler. This project uses this compiler for compilation and simulation. The compiler has been developed using JastAdd, a meta-compilation system written in Java [13]. The compiler also has an Application Programming Interface (API) that can be used to extract information from Modelica code.

Optimica Compiler Toolkit (OCT) is the commercial version of JModelica.org. OCT has more features and better numerical algorithms.

2.1.3 Regtest and Dymola

Regtest is the current testing process used at Modelon. It is a combination of using Dymola and a custom Modelica library. Dymola is short for Dynamic Modeling Laboratory. It is a tool for modeling and simulating integrated and complex systems. It can be used to create Modelica models, and allows for graphical as well as textual editing of Modelica models.

2.2 Testing Frameworks

The testing process in sofware development is either automated or manual. Constructing an automated test is often more expensive than performing a single manual test. However, once the automated test is specified, running it is much more efficient than performing the test manually. Because of this, automated testing is well

suited for regression testing. Manual testing is often required for Graphical User Interface (GUI) applications where the look and feel of the application is of interest. Performing automated tests for this purpose can be difficult.

Automated tests can be built and run using testing frameworks. A testing framework provides a way to specify tests and can also run them. Some examples of established testing frameworks are:

- JUnit, a testing framework for the Java programming language [14].
- Nose, a testing framework for the Python programming language [15].

2.2.1 Optimica Testing Toolkit

Optimica Testing Toolkit (OTT) is a tool-agnostic framework for testing Modelica models. OTT uses OCT to extract models and model information and has been developed at Modelon AB. OTT is run through the command line, and supports a number of different operations. The operation most relevant to our tool is “verify”, which compiles the model(s), simulates them and then compares the simulation results with reference data. The comparison is done with an external program.

At the start of this project, OTT could only be run through the command line, which is sometimes not as user-friendly as having a graphical user interface. See listing 2.2 to see an example of how to run OTT using the command line.

```
mrtt.py verify --verbosity=DEBUG --hash-names --compiler=OCT --compiler-options="
    jvm_args=-Xmx4g" --target-type=FMU --simulator=OCT --artifact-dir=
    test_basedir --output-dir=test_output --output="HTML,Pickle,JUnit,Hash" --
    output-options="css=resources\style.css" --test-type=experiment --reference-
    result-paths="C:\Users\student\project\Miscellaneous\trunk\Jenkins\results\
    MSL\Base;C:\Users\student\project\Miscellaneous\trunk\Jenkins\results\MSL\OCT
    " --trajectory-verifier=mav --trajectory-verifier-options="binary_path=C:\
    Users\student\project\Miscellaneous\trunk\ComplianceErrorDiagnostics\
    ThirdParty\csv-compare\compare.exe" Modelica.Blocks.Examples
```

Listing 2.2: This will run OTT with the operation “verify” on the package `Modelica.Blocks.Examples`. It is worth noting that this command line is rather long, and could be considered a bit unwieldy to use.

2.3 Eclipse

Eclipse is an extensible Integrated Development Environment (IDE) for the Eclipse Rich Client Platform (RCP). An RCP is a toolset for developing an application on an existing platform. An extension of the Eclipse RCP is the Plugin Development Environment (PDE) which allows for creation of new plugins for the Eclipse RCP.

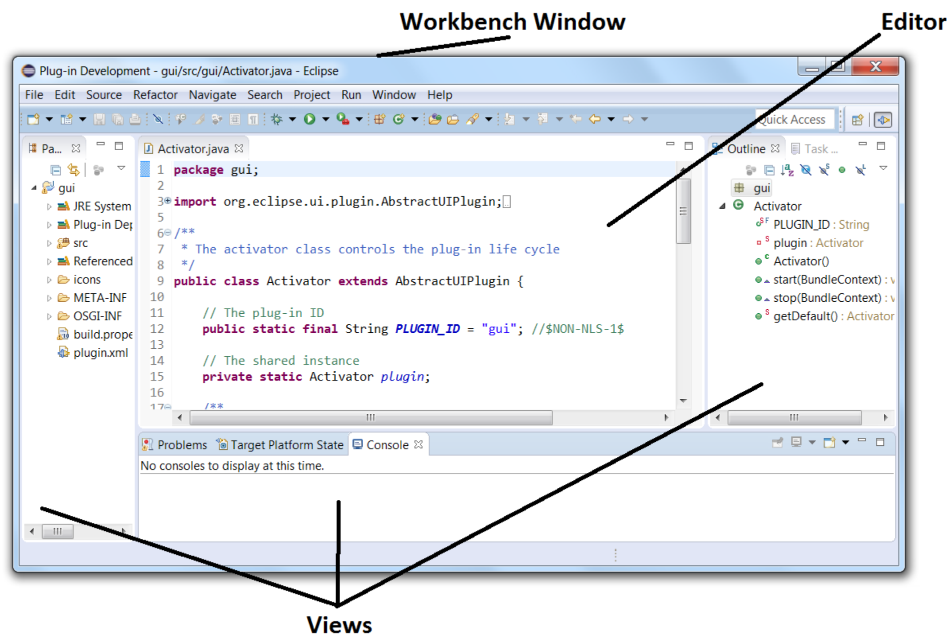


Figure 2.1: A visualization of some of the components used in Eclipse plugin development.

There are a number of different components defined in Eclipse plugin development, see figure 2.1. Some of these components are explained below.

- Workbench: A workbench contains one or more workbench windows.
- Workbench Window: A workbench window contains one or more perspectives.
- Perspective: A perspective is a set of different editors and views. Only one perspective is open at any time [16].
- Editor: An editor is typically used to edit or browse some resource.
- View: A view is a visual component that typically displays some hierarchical structure or properties of an editor. A view is similar to an editor but has no "dirty" state, meaning that changes to it are saved immediately.

2.4 Usability

When we talk about usability in this thesis, we mean the usability of software. Usability can be viewed as including a wide range of quality factors, for example maintainability. However, this thesis focuses on the aspects of daily operation as defined by Soren Lauesen [1]. Lauesen defines usability to consist of six usability factors:

- Fit for use: Does the software have the needed functionality?

- Ease of learning: Is it easy to learn?
- Task efficiency: Is it efficient for the frequent user?
- Ease of remembering: How easy is it to remember for the occasional user?
- Subjective satisfaction: Does the user feel satisfied when using the software?
- Understandability: Does the user understand what happens in the software?

The different types of user studies we used for this project were:

- Site visit: Site visit is part of the development process described by Lauesen [1]. It is usually done before the implementation begins [17]. It consists of visiting the users in their working environment and observing them using the current system, so that the existing work process can be documented.
- Task analysis: Task analysis is part of the development process described by Lauesen [1]. The usage of a program can be broken down into several tasks, after which a task analysis can be made. Task analysis is done to understand which tasks that work well, which need improvement, the frequency of performing the tasks and which are more time-consuming [1].
- Thinking aloud: Thinking aloud was used since it is a well-established method for determining usability [1, 17, 18]. Thinking aloud is the process of letting the user use the software and "think aloud" while doing so, by saying whatever the user is doing as the user is doing it [18]. Afterwards, conclusions about the usability of the software can be made based on the user's experience.
- Task time measurement: Task time measurement was used since the focus of our tool is to improve the efficiency for the regular user. Task time measurement is a suitable indicator for this factor [1]. Task time measurement (also called performance measurement) can be used to find out how long it takes to perform certain tasks [1].

Chapter 3

Implementation

The first step in the implementation of our tool was to decide which language and graphical framework to use. After that, user studies were conducted to identify the tasks that our tool needed to support, as well as what the problems had been with those tasks before. The implementation of features was done iteratively together with new user studies on the tool. This kind of implementation process was chosen because it is more fitting when focusing on usability. When determining if a program is usable or not, the feedback obtained by letting a potential user use the program is of great value. Getting this kind of feedback throughout the development process helps to keep it on the right track. Thus an iterative development process was used.

3.1 Framework and Language Selection

Early in development, we had to make a choice regarding which programming language our tool was going to be developed in. The choice was between Java and Python. Java was a candidate since OCT was written in Java. Python was considered because OTT was written in Python. There would be a need for our tool to be able to interface to both OCT and OTT.

We decided to develop our tool in Java as an Eclipse plugin. The reason was partly because of the useful built-in functions that come with development on the Eclipse RCP. There was also some consideration for the fact that there had been several projects about Modelica and the Eclipse RCP done before that this project could potentially make use of, see section 5.3 for more about this.

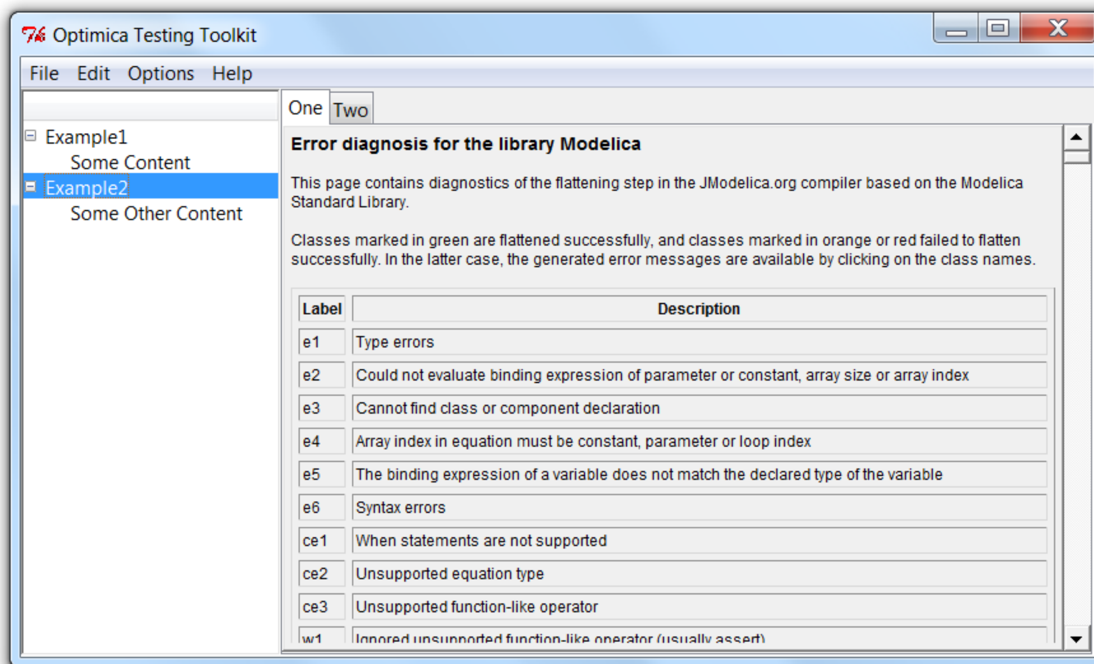


Figure 3.1: An example interface developed with TkInter.

3.1.1 Python

While deciding between which framework to use, we developed two example interfaces for the Python language, one in TkInter, see figure 3.1, and one in wxPython, see figure 3.2. We decided to try Tkinter because it is available by default in Python and because it has a low learning curve [19].

WxPython was considered because it has more advanced features than Tkinter at the expense of being slightly harder to use [20]. In particular, it has built-in functionality for displaying HTML which is a potential bonus since some results from the testing are generated in HTML. In order for Tkinter to be able to display HTML, an external module called tkhtml has to be used.

Another thing to keep in mind was that even if Python was going to be used, there would still be a need to make use of the compiler written in Java. Thus, it would require some form of Python-to-Java interface. We looked briefly at some alternatives to do this in Python, but did not try any implementation of it. The alternatives were: JCC, JPype and Java Native Interface (JNI). JPype was disregarded as it did not work on Python 2.7, which OTT is developed in. JCC is a C++ code generator for calling Java from Python. Using JCC requiers less work than using JNI, but using JNI increases control and reduces dependencies on third party software.

The examples developed showed some basic functionality, such as a menu bar, tabs and a tree view. Both views also displayed an example HTML page.

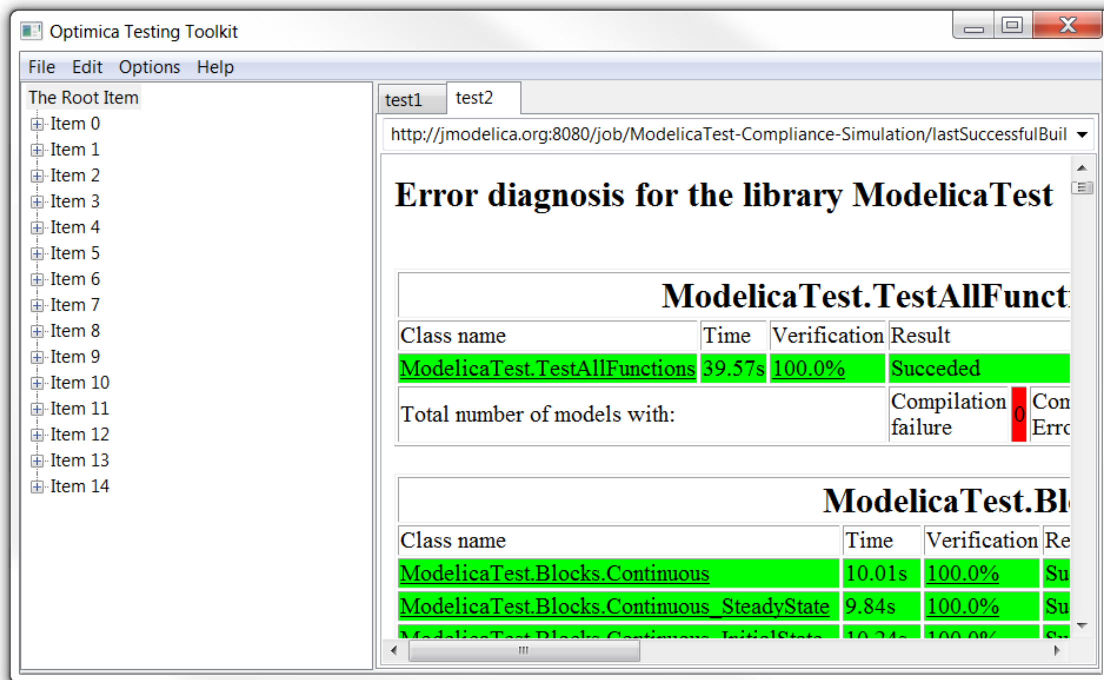


Figure 3.2: An example interface developed with wxPython.

3.1.2 Java

For Java, Swing and and developing it as an Eclipse plugin was considered. Swing is the standard GUI framework that is included with Java and is therefore a natural consideration. Developing it as an Eclipse plugin would mean it could be easily integrated with other Eclipse RCP projects. We did not try to make a mock-up for Swing or Eclipse as we did with Python, but we did try to use Eclipse plugin development for a short time to get a feel for it.

For Java, there would be a requirement for some Java-to-Python interfacing to communicate with OTT. Some alternatives for this exist, but the easiest would be to simply use the command line to run Python from within Java, and that is what we ended up using.

3.2 First Iteration

Our first user study was a site visit [17]. Each user was instructed to carry out a number of tasks. The tasks were:

1. Create a new test
2. Run the test
3. Run multiple tests

The modelers at Modelon had a specific way of testing models that they referred to as "Regression Testing", not to be confused with the general term used in software testing. In this thesis we refer to this process as Regtest. To create such a test, a test model is created that extends a specific test interface and the model to be tested. To specify the modifiers and variables of interest the user first simulates the model and then copies the names of the modifiers and variables to the test model. In addition to modifiers and variables, the test also specifies certain test parameters. A test parameter is a setting specific for the test, for example start and stop time for the simulation.

Running a Modelica test means simulating a model and comparing the results to a reference. While the test was running, the user could not use the program. The feedback while the test was running was written to a console window. When finished, the result could be viewed in a generated web page.

To run multiple tests, there were two ways. The first was to just run a test package in Dymola, which would run all the tests in the package. The second was to use an external program on a separate computer, that after being configured for a specific set of tests, would run all tests and mail the results to the user. This second approach allowed the user to continue working while the tests were being run, which is useful since sometimes running tests could take more than two hours. The feedback the user received while running the tests was Dymola displaying all the operations it carried out, as if it was done by hand.

3.2.1 Task Analysis

After the site visit, a task analysis was made. The analysis is based on one by Xavier Ferré, Natalia Juristo, Helmut Windl and Larry Constantine [17]. The different columns are:

- Task: The task that was analyzed.
- Start: When the task is started.
- End: When the task is finished.
- Frequency: How often the task is done.
- Difficult: What is difficult or complicated about the task.

We broke down the testing process into a number of different tasks, which were described by the users. These tasks were then the basis for the development of the first prototype, which needed to have support for all the listed tasks. In the task analysis, we also tried to identify problems with the different tasks in order to find out potential improvements that could be made. Also, the frequency of the tasks is helpful to determine the priority of the tasks as the more frequent ones are typically more important. Three modelers took part in the task analysis. See table 3.1 to see the results of the task analysis.

Task	Start	End	Frequency	Difficult
Add Variable	After a test has been created, sometimes to an old test	After the variable has been added	Roughly once a month	Extracting the full path name of the variable
Remove Variable	If a model has been modified	When the variable has been removed	Almost never	Nothing
Add Parameter	After a test has been created	After the parameter has been added	Roughly once a month	Nothing
Set Parameter	After a test has been created	After the parameter has been set	Roughly once a month	Nothing
Remove Parameter	If a model has been modified	When the parameter has been removed	Almost never	Nothing
Update reference	After a new test has been created, also after a new variable has been added	When the new reference has been saved	Roughly once a week	Would prefer to be able to overwrite old values easier if the new values are outside the tolerance
Create test	After the creation of a new component or a new example	When the test model has been created and is extending the original model and the test interface	Roughly once a month	Nothing
Delete test	Whenever a component is removed	After the test file is removed	Roughly once every 6 months	Might cause missing numbers in the reference file
Run test	If a change has been made to a component	After the result has been generated	Roughly once a month	Nothing
View result	After a test is run	After the result has been viewed	Roughly once a month	Results should be displayed in the current program

Table 3.1: The results from the task analysis

3.2.2 Paper Prototype

A paper prototype of our tool was developed early in the project. The first design was inspired by the design of Eclipse and Dymola. A view to find and select models in a library by using a tree structure was positioned to the left in the workbench window. A result editor displaying the OTT web page results was positioned to the right. The test editor, used for editing individual tests, was positioned in the center. To the left inside the test editor, included variables, modifiers and test parameters were displayed in a list. However, in this stage of the development modifiers and test parameters were jointly referred to as parameters. To the right inside the test editor, variables not in use were displayed in a filtered list. This first prototype was shown to one of the users, who commented that there should be an additional view for displaying all parameters, as there was for the variables.

The second prototype, as seen in figure 3.3, incorporated the feedback from the first and provided a parameter explorer as well as a variable explorer. The reasoning behind positioning everything to be able to view it at the same time was that we wanted to focus on efficiency. This kind of positioning allows for easy overview and not having to switch between different windows or tabs also increases efficiency. The reasoning behind the horizontal positioning is that represents the workflow from left to right: first the user creates a test, secondly the user sets parameters and chooses variables and thirdly views the result. The Parameters and Variables views were split horizontally because of lack of space.

As a result of discussing the second prototype with a user, we developed alternatives to this prototype. The third and fourth prototype provided two alternate ways of displaying variables and parameters: One with tab-completion fields that would only display parameters or variables when a query is entered, and one with “add” buttons that would open up a new window with a menu allowing for the adding of either variables or parameters.

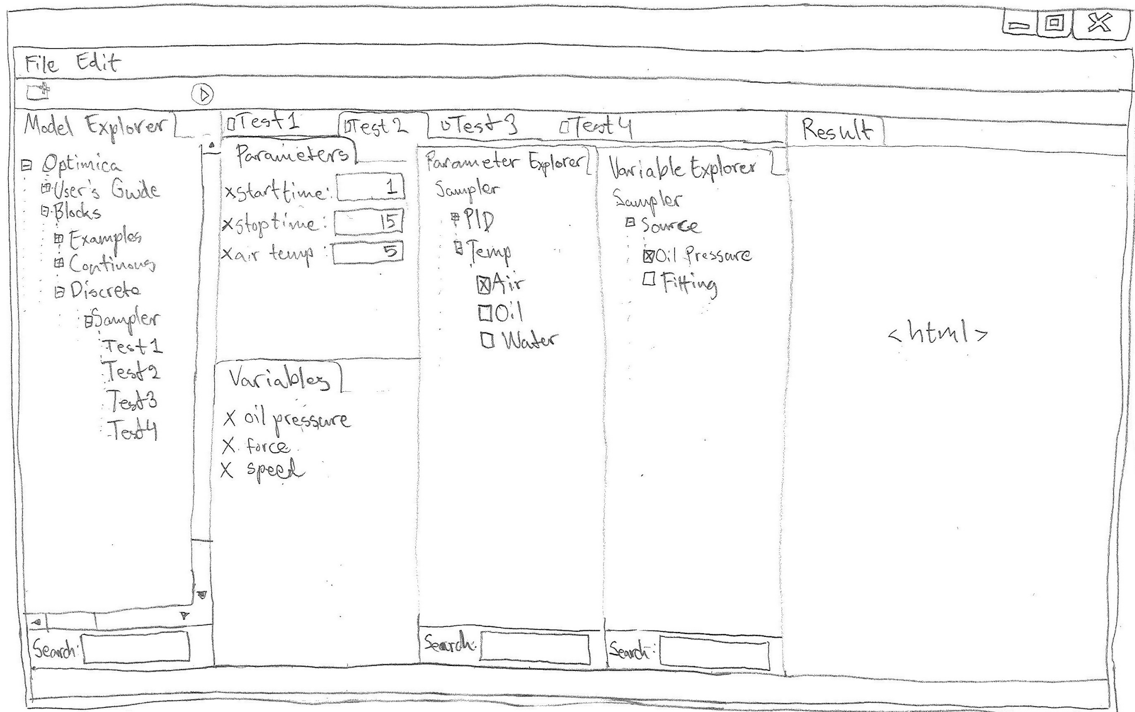


Figure 3.3: The sketch that the mock-up interface is based on.

3.3 Second Iteration

The second, third and fourth prototypes were now shown to two users. The first user thought that although the tab-completion fields seemed good, they had the disadvantage of not being able to display all variables or parameters. This would be a problem in case some user did not know the name of a specific variable that needed to be added. Also, this user did not like the idea of a new window opening up. Therefore, this user preferred the second prototype. The second user also preferred the second prototype, since the second prototype always displayed all variables and parameters which to the user was as important as seeing the added variables or parameters. One important feedback was that the parameter list should be split in two. Some parameters are specific for the test. These always need to be set, such as start and stop time. The other kind of parameters changes something in the model, such as dimensions of components. They have default values which means they do not need to be set. We decided to base the development of the mock-up interface on sketch two, as seen in figure 3.3.

3.3.1 Mock-up Interface

After discussing the mock-up with a user, it became evident that the window was too crowded. A parameter and a variable contains a lot of information which would

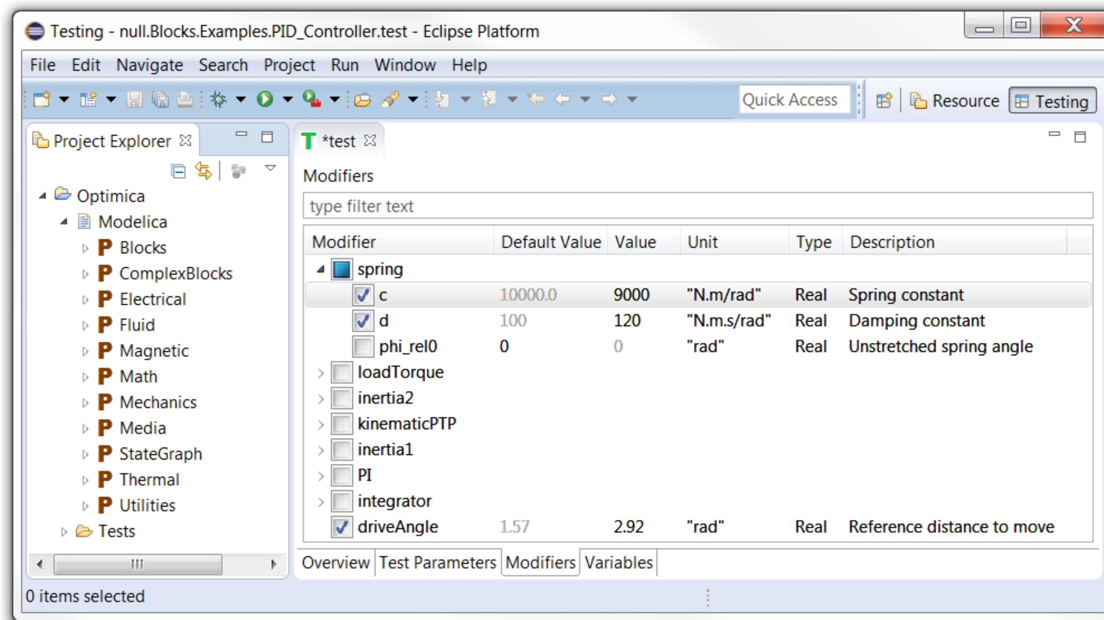


Figure 3.4: The modifiers tab of the second version of the mock-up

not fit in the lists.

Another feedback was that the mapping between the explorers and the lists for included variables and parameters were unclear. The lists for included parameters and variables were positioned vertically while the explorers were positioned horizontally. Parameters had to be split into the two subcategories: test parameters and modifiers because they are very different. Modifiers are extracted from the model while the test parameters are statically defined.

To solve these problems a second version of the mock-up was made, see figure 3.4. Instead of having everything displayed at the same time, the content is spread across four tabs: Overview, Test Parameters, Modifiers and Variables. Instead of being lists, the explorers are now checked trees. If a modifier is checked, it overwrites the default value in the model with the value specified in our tool.

3.4 Third Iteration

To examine the usability of the mock-up interface as seen in figure 3.4, the interface was tested using the thinking aloud method. The user was asked to author a test and run it for a model called `PID_Controller`. Two users took part in this user study. The discovered usability problems and the user feedback that we received are described below.

1. It is confusing whether to expand the Modelica folder or the Test folder to start creating tests.

2. It is too hard to find the model in the project explorer.
3. There is no feedback when trying to modify the overview.
4. The tabs at the bottom of the editor are not noticed at first.
5. The included variables list in the overview tab should include a description for the variables.
6. A more suitable name for the test parameters tab might be “Simulation settings”.
7. It should be more convenient to produce many similar tests.
8. It should be possible to view the bottom tabs for one test simultaneously.
9. There should be more keyboard shortcuts.
10. It should be possible to choose parameters and variables through a graphical representation of the model.
11. It should be possible to edit the test file directly using our tool.

3.4.1 Feedback Response

This feedback was taken into consideration, and a third version of the mock-up was developed, see figure 3.5. The following list is the response to the feedback given in the previous list.

1. The test folder was removed entirely as it was only confusing.
2. The package explorer was replaced by a filtered tree view, as seen in figure 3.5, to provide the possibility of searching for models.
3. The value of modifiers was made to be editable in the overview tab to make our tool more flexible.
4. This was not addressed since it is something that is learned after the first use and our tool is intended for more frequent users.
5. The overview tab was extended to contain the same information for modifiers and variables as shown in their respective tabs to increase flexibility of our tool.
6. The “test parameters” tab was not renamed. The name “simulation settings” was considered, but there were conflicting opinions among the users so no change was made.
7. This was solved by introducing test inheritance. Tests can be created to extend another test and if they do, they inherit all modifiers and their values as well as all included variables. The modifiers in the new test can still be modified, in which case the new values will overwrite the values of the parent test, as

seen in figure 3.5. Inherited modifiers and variables cannot be removed, only changed.

8. Viewing the bottom tabs for one test simultaneously would be a feature that is too expensive to implement compared to what usability is gained so we left that one out.

9. Several keyboard shortcuts were added to improve task efficiency, such as shortcuts to copy and paste tests and shortcuts to delete a test, modifier or variable.

10. Selecting parameters and variables through a graphical representation of the model is a very powerful feature. Since the graphical representation of a model is how a modeler thinks about the model it would decrease the risk of making errors when using our tool. But including this feature is outside the scope of this thesis as it is a fairly large feature to try to implement.

11. There is now an XML tab to see the XML representation of the test, but it was left as read-only as it (a) was likely to cause problems if the user could edit it and introduce errors and (b) it would be too costly to implement proper editing of the XML file, as it was not a feature with high priority.

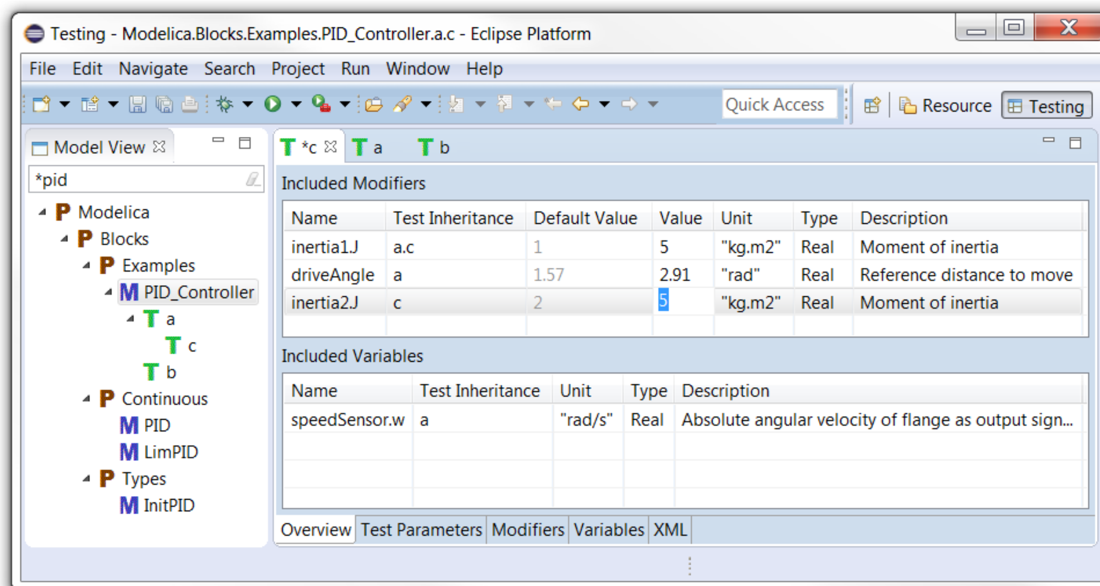


Figure 3.5: The test c extends a. C overwrites the value of driveAngle and introduces a new value for inertia2.J, but everything else is inherited from a.

3.4.2 Result Editor

We also added some features to the result editor. One of the new features was to have a console printing out the relevant information from OTT as the program was

running the tests. In addition to this, a time stamp was added to the result editor showing the time the tests were run as well as the name of the launch configuration that was run. There is now also an “update reference” button next to each result that can be used to update either a missing reference or to replace an old reference with a new set of values. The final version of the result editor can be seen in figure 3.8.

3.4.3 OCT Integration

After the mockup was finished, integration with the OCT compiler was started. We used the OCT compiler since it had been developed at Modelon and it was easy for us to get access to it. Since our tool and the compiler were both written in Java, this could be easily done by instantiating the compiler at the startup of our tool and then letting the compiler load the Modelica Standard Library. The model view was populated using the compiler and relevant information could then be extracted from each model. All the modifiers of the components in the model were extracted to populate the modifier tab, as well as the variables for the variables tab. Relevant information for each variable and modifier was extracted using the compiler.

3.4.4 OTT Integration

OTT is run through the command line. The full command line that is run is based on the run configuration from our tool. Before this project, OTT did not support running XML files as tests, so OTT itself had to be extended in order to support this functionality. Previously, OTT was run with “targets” as one option in the command line, with targets being a list of Modelica models and/or packages. If a package is included in targets, then all models inside that package are intended to be tested. We also extended OTT to be able to handle any number of XML files included in targets. OTT then parses those XML files and treats them as tests in addition to the other tests extracted from the remaining targets.

3.4.5 Test Representation

We decided to store tests as XML files. The XML file of a test contains the selection of variables and modifiers to be used in a specific simulation, as well as what values to use for the included modifiers. Additionally, the XML file also contains test parameters such as start time, stop time, tolerance and other options. See listing 3.1 to see an example XML file.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Test extends="" model="Modelica.Blocks.Examples.PID_Controller">
  <TestParameters>
    <TestParameter name="Tolerance" value="1e-6"/>
    <TestParameter name="Start Time" value="0"/>
    <TestParameter name="Stop Time" value="5"/>
    <TestParameter name="Fixed Step Size" value="0"/>
    <TestParameter name="Number of Control Points" value="2000"/>
  </TestParameters>
  <Modifiers>
    <Component name="inertia1">
      <Modifier name="J" value="1"/>
    </Component>
    <Modifier name="driveAngle" value="1.57"/>
  </Modifiers>
  <Variables>
    <Component name="inertia1">
      <Variable name="w"/>
      <Variable name="a"/>
    </Component>
  </Variables>
</Test>
```

Listing 3.1: An example of the XML file of a test.

3.4.6 Eclipse Run Configuration

The run configuration contains what operation to run, as well as a set of XML files that are to be used for the operation, and any options adhering to that operation. An option might be, for instance, what compiler to use. See figure 3.6 to see the run configuration of our tool.

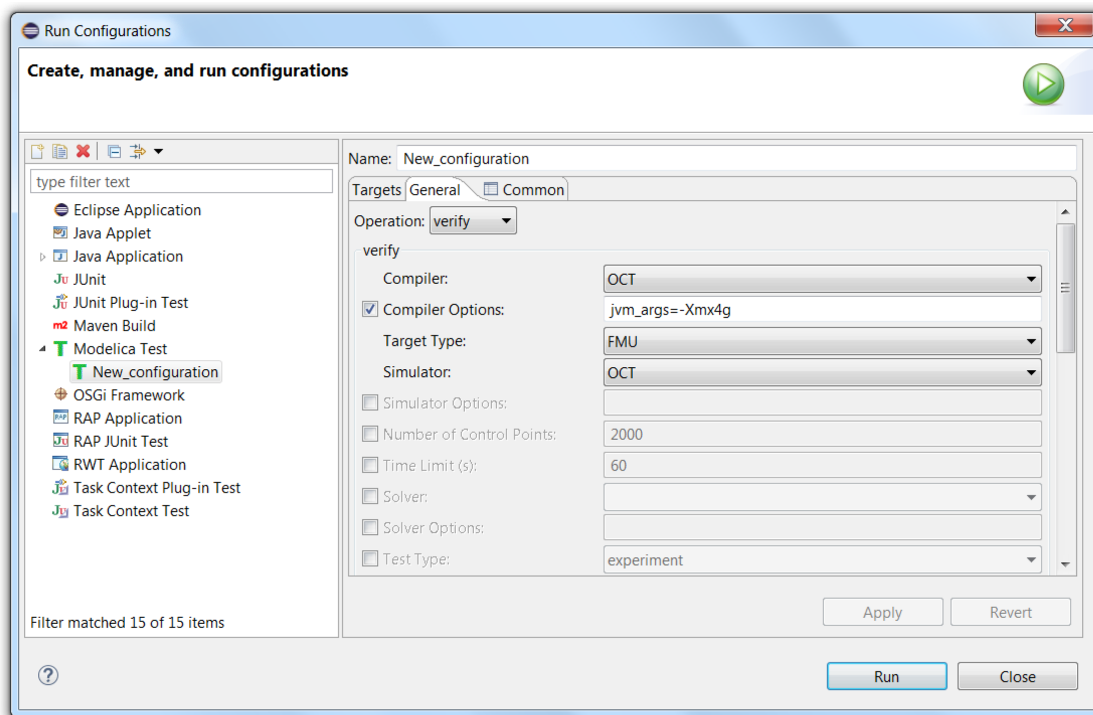


Figure 3.6: The "general" tab of the run configuration.

3.5 Fourth Iteration

A set of tasks were prepared for a thinking aloud study, with the idea being that the user would attempt to use our tool to complete the tasks. The tasks were:

1. Navigate to any model that is feasible to test.
2. Create a test for the model with at least one modifier and at least one variable included.
3. Create two tests with the same modifiers and variables as the first test, but change the value of one of the modifiers.
4. Run the tests and update the reference values.
5. Create a run configuration and change the operation to 'simulate'.

The results were varied. One user was unable to at first find the different tabs of the test editor, and was stuck at the overview tab trying to add modifiers there. However, once the user found the correct tabs the tasks could be completed without much difficulty.

Another user was able to add modifiers and variables without problems, but was then unable to find a way to save the test. After it had been pointed out to the user where the save button was, the user had no subsequent problems with saving tests.

In summary, after analysing what the users said and did during the user study, the following usability problems were discovered:

1. When navigating the modifier and variable browsers, icons should be used to tell components from modifiers and variables.
2. If a test has not been saved and a subtest is created for it, the subtest will not inherit the changes and there is no feedback about this. Also, if there are unsaved tests there is no feedback about this when pressing "run".
3. How a test result is identified is currently done by the model name, what modifiers the model have, and the values of those modifiers. How tests are identified should correspond to how they are created in our tool.
4. Tests should have their own run configuration. For example, two tests should be able to be run in the same batch where one test is of type verify and one is of type simulate.
5. When showing the context menu for something in the model view there should be an option to run all tests below the selected element in the tree.
6. In the targets tab in the run configurations, it should not be possible to select a model that has no associated tests.
7. If we are in a test it should be clearer if the used value for a modifier is defined in this test or in a parent test.
8. The result editor should have some clickable button, in addition to backspace, to return to the main view after you have clicked on a link.
9. In the result editor, it is unclear which table row corresponds to which test, only what model is tested is shown.
10. That a reference has been updated is not clear in the result editor.

3.5.1 Feedback Response

Some of the above points were addressed.

1. This was fixed by adding some placeholder icons, displaying a "c" for components, a "v" for variables and a "m" for modifiers.

2. This was fixed by adding a confirmation message if a user attempts to add a subtest to an unsaved test, or if the user presses “run” when there are unsaved tests.
3. This was fixed by modifying OTT so that the hash in the result file is derived from the path of the test file rather than the model name. Doing this means that changing modifiers for a test does not change the path to the reference file for that test. Consequently, the user can change the modifiers and compare the results to the previous results from that test.
4. This was not taken care of, see chapter 4 for an explanation of why.
5. This was fixed by adding a “run all” option to all packages, models and tests with subtests. Also, pressing “launch” while a model is selected executes “run all” by default.
6. This was not taken care of, see chapter 4 for an explanation of why.
7. This was fixed by adding color coding, with a blue value indicating that the value is defined in a parent and a green value indicating that the value is defined in the current test.
8. This was addressed by adding a “back” and a “forward” button to the browser.
9. The output was changed to represent the full path of the test rather than just the model. So if there is a test named “filtertest” for the model “Modelica.Blocks.Examples.Filter”, the results will now show “Modelica.Blocks.Examples.Filter.filtertest” instead of just the model name.
10. This was fixed by changing the button to a “reference updated” message if the user updates a reference.

See figures 3.7 and 3.8 to see these changes in our tool.

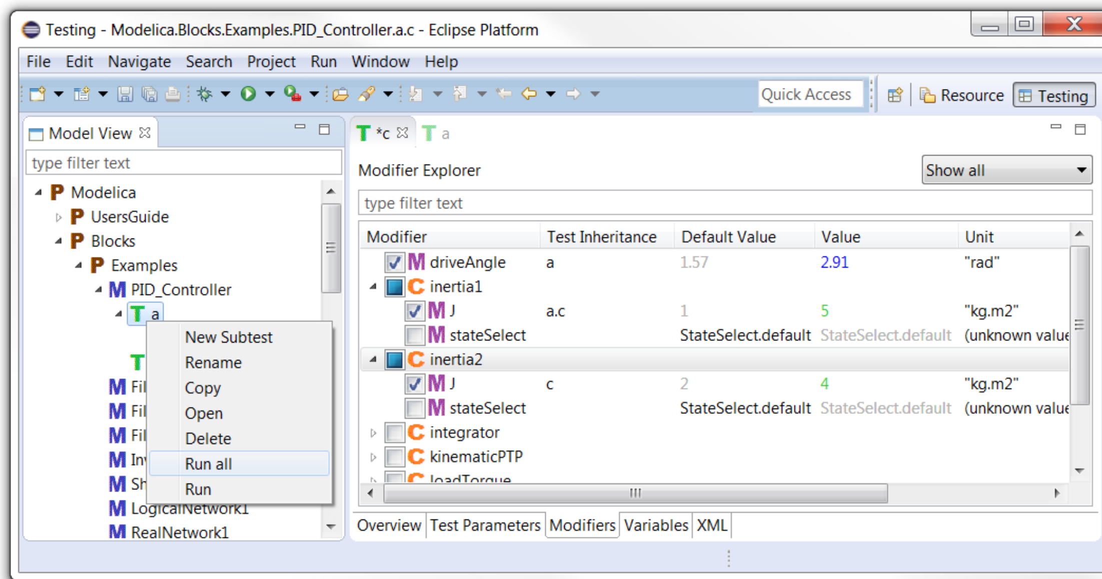


Figure 3.7: Added features shown here are: placeholder icons for components and variables, color coding for values and a “run all” option in the context menu when selecting a model, a package or a test with subtests

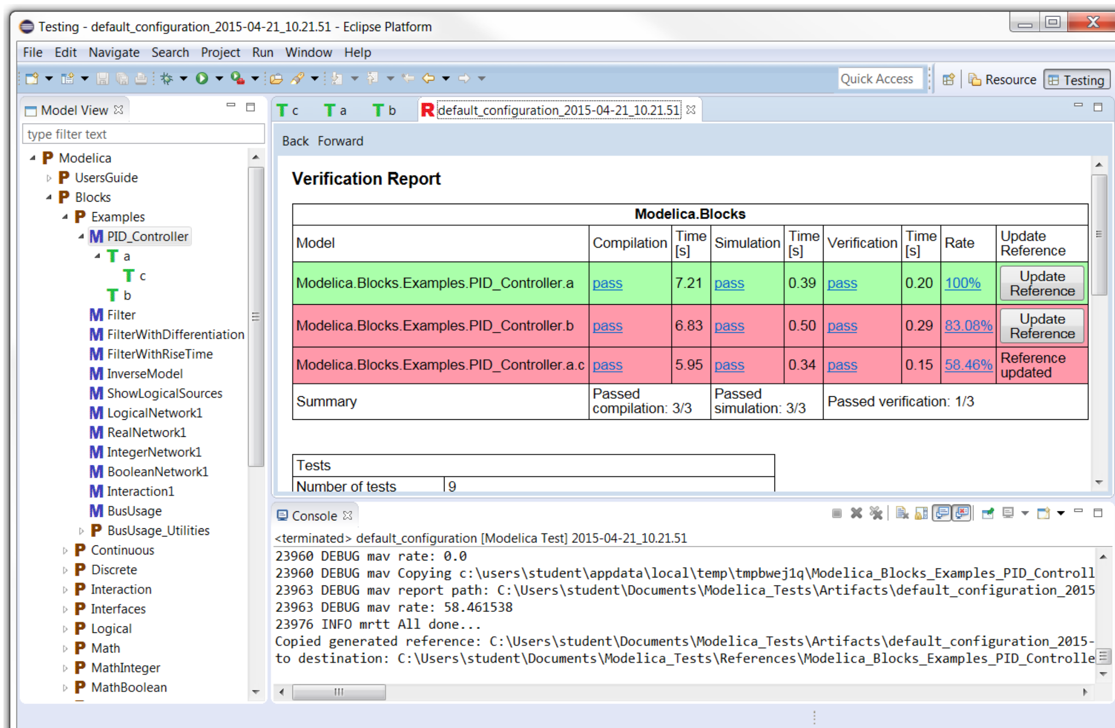


Figure 3.8: The result editor displays the proper names of each test rather than just the model name. There is also a “back” and “forward” button for the browser. Also, there is a more clear indication of when a reference has been updated.

3.6 Task Time Measurement

To evaluate task efficiency, a task time measurement was done. The users were asked to perform a set of tasks using both Regtest and our tool. Normally, a task time measurement has a target time, but our target time was simply that our tool should be at least as good as when using Regtests. We also decided to measure the total time it took to perform the tasks rather than to measure each task individually, since we were interested in the overall efficiency of our tool. The tasks were:

1. Navigate to any model that is feasible to test.
2. Create a test for the model with at least one modifier and at least one variable included.
3. Create two tests with the same modifiers and variables as the first test, but change the value of one of the modifiers.
4. Run the tests
5. Change one modifier in one test, run that test and update the reference values.

See table 3.2 to see the results of the task time measurement. Most of the users used Regtest first and then our tool, and it is possible that performing the tasks in Regtest first created a bias towards our tool, since they were then familiar with the tasks. However, user 1 and user 4 used our tool before using Regtest and we could not discern a noticeable difference in the results. All users were familiar with Regtest and had used it before. User 1, 2 and 3 had previously participated in user studies so they had a basic grasp of our tool. The other three users were given a short introduction on how to use our tool just before using it.

User	Time with Regtest	Time with our tool
User 1	6:33	2:43
User 2	6:47	4:07
User 3	6:52	3:14
User 4	6:33	3:34
User 5	4:15	2:38
User 6	8:40	2:56
Average	**6:37**	**3:12**

Table 3.2: The results from the task time measurement

Chapter 4

Evaluation

We developed a functional tool in the allocated time. The first user studies revealed some of the problems with the previous testing method, with the most important ones being:

1. The names of the variables were too difficult to find
2. The results should be displayed in the same program that is used for creating and running tests
3. The reference values should be easier to overwrite

Item 1 was addressed by extracting the names of all variables and modifiers with the OCT compiler and structuring them in trees. Thus, the user does not need to know the full path of a variable, but can simply find it in the tree structure. Item 2 was addressed by opening a new editor displaying the results after the tests had been run. Item 3 was addressed by adding the “update reference” button to the result editor.

Our tool addresses the usability factors defined by Lauesen [1] in the following way:

- Fit for use: this is mostly covered, as all the tasks discovered in the task analysis are supported. There are still other features that can be added, see section 6.1 for more on this.
- Ease of learning: The user studies did show some problems due to not being familiar with it. However, As the tool is intended to be used for longer periods of time, the initial intuitive use of the tool is less important than the efficiency when using it regularly.

- Task Efficiency: The result from the task time measurement was that our tool was twice as fast as Regtests for the selected combination of tasks.
- Ease of remembering: The users who used our tool after having previously used it usually had no problems with it, except that it was a bit difficult to run properly. For instance, it was unclear that pressing the "run" shortcut while selecting a test would run only that test and not the subtests belonging to that test. It is possible that this is something that should be changed.
- Subjective satisfaction: The user response to using the program was generally positive, especially when compared to Regtest. The main complaint was that the modelers felt lost since they could not view the graphical representation of the model. The modeler's understanding of the model is based on the graphical representation, since that is how they develop it. A modeler might not know the name of a variable or a parameter but instead the location of it in the graphical representation of the model. Also, by viewing the model graphically, it can be easier to discover variables and parameters of importance.
- Understandability: The feedback to the user when running a test is the debug output from OTT, which includes redundant things like which code is being executed by OTT. This output was not well understood and should probably be changed. Also, When we asked the modelers to create two similar tests, most of the modelers expected to be able to create a test that inherits the modifiers and variables of the specified test. However, how test inheritance was represented was not as clear and required some explanation. Also, for the color coding that was added to distinguish whether a modifier defines a value or inherits it, it was unclear what the different colors did and it is possible there should be a more clear indication of this somewhere.

There were a few other problems as well. For instance, the run configuration is not only unintuitive, but it also creates ambiguity since it has "targets" that it selects that are different from the selection in the model view. It is therefore possible for a user to select a test and run a run configuration and end up running other tests than the test selected. It is questionable whether or not the run configuration is necessary at all. It might be better to migrate all settings in the run configuration to the tests individually instead, which would remove ambiguity and make the process of running tests a lot smoother.

Chapter 5

Related Work

5.1 UnitTesting

UnitTesting is a library for Modelica Unit Testing [21]. The tests are defined programmatically by extending a `TestCase` model. Another model is used to display the results, different models are used for different ways to generate the results. This model is also used to select variables of interest.

UnitTesting can generate a wide range of testing metrics. Component coverage is a metric that measures how many components are covered. State coverage analyses the code to find variables of interest and compares that to what variables that are chosen in the test. Condition coverage checks which ways through an if-statement has been tested. Pinpoint metric identifies in which components of a model there might be errors.

UnitTesting is a programmatic way to define tests compared to our tool which is more graphical way. UnitTesting focuses on providing testing metrics while our tool focuses on the test authoring process.

5.2 MoUnit

MoUnit is a framework for automatic Modelica model testing [5]. It can be used for testing Modelica models with unit tests. It supports usage of multiple simulation tools as well as input files from other tools for result comparison. MoUnit is

integrated into the Modelica IDE OneModelica and can also be used for automated build environments like Hudson.

Modifiers (Parameters) can be modified in the test and variables of interest can be selected in the test. Tests can be run in test suites, a test suite contains tests and/or other test suites to be run. MoUnit can compare simulation results with references similar to OTT. MoUnit also has support for stochastic models. Stochastic properties of a model can be validated by looking at the difference between the mean and standard deviation of the reference and the simulation.

5.3 Eclipse Plugins

There have been several other works done on Eclipse and Modelica before. A Modelica IDE was started by Jesper Mattson [22] in 2009 as part of his master's thesis. It had support for text based editing of Modelica models including the features: syntax highlighting, auto indentation, brace matching, code folding and error checking. It was improved upon in 2010 by adding support for auto-completion among other things which was done as a part of Philip Nilsson's master's thesis [23]. Kristina Olsson and Lennart Moraeus worked on the rendering of graphical icons for Modelica classes [24] in 2011.

The two most recent additions were done in 2012 by Jon Sten and Jonas Rosenqvist. Jon Sten worked on a graphical representation of Modelica models and editing of graphical models [25]. Jonas Rosenqvist worked on handling documentation of Modelica code in Eclipse [26]. He implemented a new plugin featuring a documentation editor, which was capable of displaying and editing the documentation of Modelica models. In addition to this, the plugin also had support for exporting all documentation to the file system.

Chapter 6

Conclusion

In this thesis we developed a functional test authoring tool for Modelica models. The basic workflow using our tool is as follows: the modeller (a) creates a test for a specified model, (b) selects variables and modifiers to include in the test in tree views, (c) runs the test and (d) examines the results. Features included in our tool are: test inheritance, undo/redo operations, keyboard shortcuts, run configuration and reference updating. Information about models, modifiers and variables are extracted using the OCT compiler. The tests are saved as XML files and run using Optimica Testing Toolkit.

We continuously evaluated our tool by using the methods described by Lauesen [1]. We provided solutions for the three main problems discovered in the evaluation of Regtest. Every iteration began with an evaluation followed by a response in the form of implementation in our tool. As a result, by using our tool in the testing process instead of the process used with Regtest, we were able to decrease the time spent for modellers on a combination of common tasks by 52%.

We do not consider our tool to be a completely finished product and as such it does not have to provide all the desired functionality. It does, however, provide examples of how certain things can be done better. In particular, a lot of the flaws of the original testing process were found and improved upon in this project.

6.1 Future Work

A feature missing from our tool was a graphical representation of the model. This was omitted mainly due to lack of time and for being outside the scope of this thesis,

but it was a feature often requested by the modelers. This feature has been done for the Eclipse RCP and Modelica before [25], but not including the testing aspect.

Editing the model and testing the model should preferably be done in the same program since they are closely related.

Also, proper icons for the models should be used. In our tool, placeholder icons were used but ideally there are icons that can be generated from the Modelica code. By using those icons, models and packages would be more recognizable thus increasing efficiency and decreasing the possibility of a mix-up. Kristina Olsson and Lennart Moraeus worked on this previously [24].

In addition to these, there were various small features that were desired but did not make it into our tool. Some of those were:

- It should be possible to edit the tests directly in the XML tab, and the other views should be updated when this is done.
- Our tool currently only supports working with the Modelica standard library. To make our tool more useful it should support creating tests for any Modelica library.
- Our tool is currently developed as an Eclipse plug-in. This provides a lot of functionality that is left unused by our tool, which for a modeler that is not familiar with Eclipse is only confusing. To avoid all the unnecessary functionality, the tool should instead be built on the Eclipse RCP as a stand-alone application.
- There should be better support for choosing values for modifiers. Currently, only boolean values are actually restricted to correct values. For other types, there is no restriction on the input. Also, for certain types, the allowed values should be available for the user. For example, if a filter can be of type “high-pass” or “lowpass” then it should be obvious to the user that those are the valid values. Also, our tool works with an array when choosing values but not for a matrix of any size.
- The output of our tool as tests are running should more clearly indicate what is happening.

Bibliography

[1] Soren Lauesen. *User Interface Design - A Software Engineering Perspective.* Addison-Wesley, 2005.

[2] Modelon AB. About modelon. `http://www.modelon.com/about-modelon/`. Accessed: 2015-06-05.

[3] Kerry Zallar. *Practical Experience in Automated Testing.* Methods and Tools, 2000.

[4] Ilene Burnstein. *Practical Software Testing : A Process-Oriented Approach.* Springer, 2004.

[5] Roland Samlaus, Mareike Strach, Claudio Hillmann, and Peter Fritzson. *MoUnit - A Framework for Automatic Modelica Model Testing.* Proceedings of the 10th International ModelicaConference, 2014.

[6] Martin Otter and Hilding Elmqvist. *Modelica - Language, Libraries, Tools, Workshop and EU-Project RealSim.* German Aerospace Center and Dynasim AB, 2001.

[7] Modelica Association. Modelica - a unified object-oriented language for systems modeling, language specification version 3.3 revision 1. page 31, 2014.

[8] Jonathan Kämpe. Applying constant propagation in a modelica compiler. Master's thesis, Lund University, 2013.

[9] Jung-Wei Chen and Jiajie Zhang. Comparing text-based and graphic user interfaces for novice and expert users. In *AMIA Annual Symposium Proceedings*, volume 2007, pages 125–129. American Medical Informatics Association, 2007.

[10] Modelica Association. Modelica - a unified object-oriented language for systems modeling, language specification version 3.3 revision 1. pages 221–225, 2014.

[11] Open Source Modelica Consortium. *Modelica Compliance Library Guide.* 2013.

[12] Johan Åkesson. *Languages and Tools for Optimization of Large-Scale Systems.* PhD thesis, Lund University, 2007.

[13] Görel Hedin and Eva Magnusson. Jastadd—an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003.

[14] Erich Gamma and Kent Beck. Junit: A cook's tour. *Java Report*, 4(5):27–38, 1999.

[15] Daniel Arbuckle. *Python Testing: Beginner's Guide.* Packt Publishing Ltd, 2010.

[16] Dave Springgay. *Using Perspectives in the Eclipse UI.* Object Technology International, Inc, 2001.

[17] Xavier Ferré, Natalia Juristo, Helmut Windl, and Larry Constantine. Usability basics for software developers. *IEEE software*, 18(1):22–29, 2001.

[18] Jakob Nielsen. *Usability Engineering.* Morgan Kaufmann, 1993.

[19] John W. Shipman. *Tkinter 8.5 reference: a GUI for Python.* New Mexico Tech Computer Center, 2013.

[20] Hugues Talbot. wxpython, a gui toolkit. *Linux journal*, #74, June 2000.

[21] Michael M Tiller and Burit Kittirungsi. *UnitTesting: A Library for Modelica Unit Testing.* 2006.

[22] Jesper Mattsson. The jmodelica ide: Developing an ide by reusing a jastadd compiler. Master's thesis, Lund University, 2009.

[23] Philip Nilsson. Compiler extensions for semantic editing using jastadd. Master's thesis, Lund University, 2010.

[24] Kristina Olsson and Lennart Moraeus. Eclipse-based graphical rendering and editing of modelica code. Bachelor's Thesis, Lund University, 2011.

[25] Jon Sten. Graphical editing in jmodelica.org. Master's thesis, Lund University, 2012.

[26] Jonas Rosenqvist. Handling documentation of modelica code in eclipse. Master's thesis, Lund University, 2012.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTATIONSDAG 2015-06-04

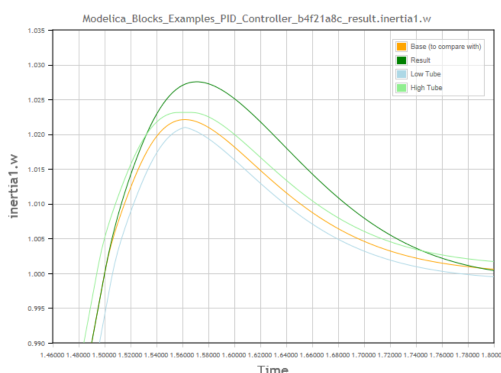
EXAMENSARBETE Developing a Test Authoring Tool for a Modeling Language
STUDENTER Victor Johnsson, Anders Tilly
HANDLEDARE Jesper Öqvist (LTH), Jon Sten (Modelon)
EXAMINATOR Görel Hedin (LTH)

Testverktyg för modelleringsspråk

POPULÄRVETENSKAPLIG SAMMANFATTNING **Victor Johnsson, Anders Tilly**

Många företag idag vill göra simuleringar i en dator innan de börjar bygga fysiska prototyper. Fördelen med detta är att det är billigare att simulera produkter än att bygga prototyper, och det går snabbare att göra ändringar.

För att göra simuleringar så kan man göra en modell av verkligheten i ett speciellt modelleringsspråk anpassat till detta. När man gör sådana modeller vill man också testa modellerna för att kontrollera att de fungerar som de ska. Vi gjorde ett verktyg för att skapa test till modeller skrivna i modelleringsspråket Modelica. I en sådan modell så finns det ett antal olika variabler för de olika delarna, till exempel så kan det finnas en variabel för temperaturen och en variabel för trycket. Vissa variabler kan sättas till att alltid ha samma värde under en simulering, och kallas då för parametrar. En parameter kan till exempel bestämma storleken på en cylinder i en motor. Ett test av en modell innebär att man ställer in värden på vissa parametrar, och därefter simulerar modellen och ser vad man får för värden på de andra variablerna. Slutligen jämförs variablernas värde med ett sedan tidigare känt referensresultat. Det finns också en viss tolerans för hur långt utanför referensvärdet man får lov att komma. Testet går igenom om värdet man får är innanför toleransen. Se figur 1 för att se hur ett resultat kan se ut.



Figur 1. Resultatet från en simulering jämfört med referensvärdet

Implementation

Vårt verktyg är skrivet i programmeringsspråket Java. Vi jobbade dessutom med modellerare som kunde testa vårt program och ge oss feedback medan vi jobbade på det. Vi använde oss av ett flertal olika användarstudier, som är metoder för att upptäcka hur användarvänligt ett program är. Med hjälp av detta kunde vi se till att vårt verktyg blev användarvänligt och vi kunde upptäcka alla funktioner som vårt verktyg var tvunget att stödja.

Vårt verktyg låter användaren skapa test till modeller, och visar därefter en lista på alla tillgängliga variabler och parametrar. Användaren kan välja variabler att undersöka och ändra parametrar. Se figur 2 för att se hur vårt verktyg visar parametrar. Utöver detta så kan man även starta simuleringar från vårt verktyg, och även jämföra resultatet från simuleringen med referensvärden.

Modifier	Value	Unit	Description
C ambient1			
C ambient2			
C heatCapacitor			
M C	0.2	"J/K"	Heat capacity of element...
C heatFlow			
M k	13	""	Constant output value
C medium			
M cp	0.95	"J/(kg.K)"	Specific heat capacity at ...
M cv	1.15	"J/(kg.K)"	Specific heat capacity at ...
M lamda	1	"W/(m.K)"	Thermal conductivity
M nue	0.87	"m2/s"	Kinematic viscosity
M rho	1	"kg/m3"	Density
C pipe			
C prescribedHeat			

Figur 2. Vårt verktyg när den visar parametrar. De kallas här för “modifiers”.

Resultat

Det fanns redan ett system för att testa modeller, men många av modellerarna var missnöjda med det. När vi jämförde vårt verktyg med det gamla som fanns så gick det dubbelt så fort att skapa test med vårt verktyg, vilket är ett resultat vi är nöjda med.