Optimica Testing Toolkit: a Tool-Agnostic Testing Framework for Modelica Models

Anders Tilly[1] Victor Johnsson[1] Jon Sten[2] Alexander Perlman[2] Johan Åkesson[2]

[1]Lund University, Sweden, {ada09ati,ada10vjo}@student.lu.se
[2]Modelon AB, Sweden, {jon.sten,alexander.perlman,johan.akesson}@modelon.com

Abstract

The need for regression testing increases as the size and complexity of software projects grow. The same is true for Modelica libraries and Modelica tools. Large Modelica projects often involves several Modelica tools and libraries which are under development. In those situations, with several orthogonal code bases, the need for systematic regression testing is needed.

In this paper we investigate a new way to create and run tests by developing a tool-agnostic testing framework. Additionally a graphical user interface for test authoring and management was created.

Keywords: Cross Testing, Testing Framework, Test Authoring, Regression Testing, User Interface, Modelica, FMI

1 Introduction

Optimica Testing Toolkit (OTT) is a tool-independent framework for performing automatic testing on Modelica models. It supports both static and script-based testing. Static testing is used to perform predefined tests on a subset of models in a library, where the user provides the specific library as well as a criteria for selecting which models to test. Each model is automatically compiled and simulated and the resulting trajectories are compared to reference trajectories. Script-based testing enables the test author to write finely tuned tests that interact with the compilation and simulation process and to test individual models with specific compiler and simulator scenarios. OTT supports cross-tool testing with several different Modelica compilers and simulation environments using FMI.

The purpose of OTT is not only to provide a framework for testing Modelica models, but also to provide a testing pipeline that is tool agnostic. OTT provides the same testing pipeline regardless of what compiler and simulator performs the actual model compilation and simulation. Tool agnosticism is provided by means of an abstraction layer between OTT and the actual tools. Each tool is hooked into the abstraction layer via a plugin tailored specifically to that tool.

As part of the development cycle a Graphical User Interface (GUI) was developed (Tilly and Johnsson, 2015). The GUI can be used for test authoring, test configuration and test execution. One important aspect considered during development was to ensure that the GUI had good usability. We used a number of different user studies together with the users in order to discover usability problems, and then used iterative development to address and fix those issues.

OTT was initialy developed by Modelon as an in-house tool for performing library testing and verification using several Modelica tools. It has since been extended with the GUI and other features and is now provided and maintained as a commercial product by Modelon [1].

2 Background

In software development, a test is usually run and checked towards an expected result (Burnstein, 2004). Testing Modelica models are tested using the same concept. More specifically, testing a Modelica model means testing if it: (a) can be translated and simulated without error, (b) delivers the expected results, and (c) represents reality adequately (Samlaus et al., 2014). For aspect b, there are reference values that are considered to be the "correct" values. 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.

Testing consists of test authoring, test configuration and test execution. Authoring a test for a model means to select some variables to compare against references, and also changing some parameters in the model. Test configuration refers to choosing the appropriate settings for the test, such as which compiler to use, and test execution refers to running the tests.

In Modelica, models can be created and represented both textually and graphically. Using a graphical user interface is sometimes more efficient than using a program-

[1]http://www.modelon.com/

```
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 1. An example of TestCase and experiment annotations. This example is taken from the Modelica Compliance Library Guide (Open Source Modelica Consortium, 2013).

matic approach (Chen and Zhang, 2007). Test authoring on the other hand is usually 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 (Modelica Association, 2014). There are two types of annotations that are relevant 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 (Modelica Association, 2014). See listing 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 (Open Source Modelica Consortium, 2013). See listing 1 to see an example of this annotation.

2.1 Testing Frameworks

The testing process in software 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 has been specified, running it is much more efficient than performing the test manually. Because of this, automated testing is well suited for regression testing. Regression testing means performing tests continuously throughout the development process. This is done to discover possible introduced errors when making changes in the software. Manual testing on the other hand is done by a human. Manual testing is often required for GUI applications where how things look and feel 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 for specifying and executing tests. Some examples of established testing frameworks are:

- JUnit, a testing framework for the Java programming language (Gamma and Beck, 1999).
- Nose, a testing framework for the Python programming language (Arbuckle, 2010).

2.2 Usability

When we talk about usability in this paper, we mean the usability of software. Usability can be viewed as including a wide range of quality factors, for example maintainability. However, this paper focuses on the aspects of daily operation as defined by Soren Lauesen (2005). 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?

2.3 Related Work

Testing and automatic testing is nothing new to Modelica and FMI. Two examples of such implementations are: UnitTesting (Tiller and Kittirungsi, 2006) and MoUnit (Samlaus et al., 2014).

UnitTesting is a Modelica based library targeted at unit testing of Modelica models. Tests are created by defining Modelica models which extends the UnitTesting library. One big aspect of the testing library is to provide a wide range of metrics for the tested models. Example of supported metrics are component-, condition- and static-coverage.

MoUnit is a framework for automatic Modelica model testing. Tests are written in a language defined by MoUnit. MoUnit is integrated into the Modelica IDE OneModelica which supports the user during test authoring and test execution. However MoUnit can also be used standalone when integrated into automatic build environments such as Jenkins and Hudson. MoUnit provides result reporting and comparison against reference results.

The solution presented in this paper differs from these implementations. It is tool-agnostic, plugin-based and supports enhanced cross-testing. This enables library developers to verify their library with different Modelica and FMI tools. Additionally it allows for cross-testing between different compilation and simulation tools.

DOI 10.3384/ecp15118687

3 Test Methodology and Requirements

3.1 Static

Static testing is primarily used to test a large set of models that share one or more properties, e.g. package container, base class, annotation etc. A static test session begins with a set of Modelica packages containing test models as input, as seen in figure 1. The packages are traversed and models with properties matching a given set of criteria are selected. When a selection is made the test cycle; translation, compilation, simulation and verification begins. After verification is complete, information collected during test execution is passed to a set of output modules responsible for rendering the results, this completes the test cycle. The test session terminates when no more models are found.

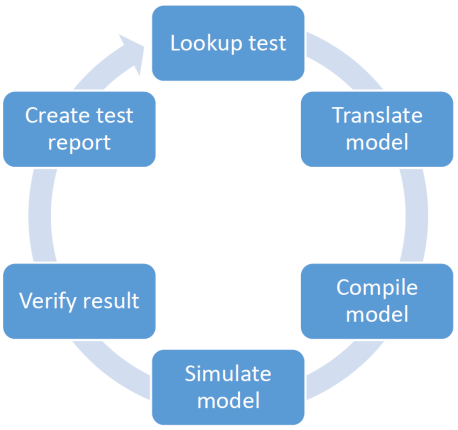

Figure 1. OTT static test cycle

The following types of static tests are currently supported:

- Experiments, tests models containing the `experiment()` annotation
- Test cases, tests models containing the `__ModelicaAssociation(TestCase())` annotation

A test session may be explicitly setup to run its test cycles up to and including a certain operation and still be considered completed (this is true of all static tests except for the Test cases tests where a test may be specifically designed to fail during one of the operations).

Due to the limitations of the experiment and test-case annotations, a new test specification format is under development. The format facilitates additional input and output for the test, such as modifiers for parameters, reference variables, tool specific options and much more. The current implementation of this test specification stores most of the information in an Extensible Markup Language (XML) file, but in some cases uses other formats, such as tool-specific scripts and reference results. This paper will not explore this specification format further due to its current state.

3.2 Scripted

Script-based testing is primarily used to perform fine-grained and diversified testing of models which, unlike the models used for static testing, share none or very few properties. The OTT script-based testing pipeline gives the user total control over the testing process.

Much like static testing, OTT automatically retrieves relevant tests based on the sieve provided by the user. But that is where the similarities to static testing ends. It is the user defined test that is the driving factor during execution of scripted tests. Instead, OTT provide convenient and uniformed interfaces to the different Modelica tools and result reports. This gives the user full control of the test execution and less worry about tool specific interfaces. Additionally OTT provides mechanisms for populating and producing test reports.

Scripted tests are written in Python and resembles tests written for the Nose testing framework. However, unlike Nose, OTT provides interfaces to common Modelica and FMI tools.

3.3 GUI

The basic workflow when using the GUI is as follows: the modeller (a) creates a test for a specific model, (b) selects variables and parameters to include in the test, (c) runs the test and (d) examines the results.

When running a test in the GUI, the included variables and parameters and their values are extracted from the test and run using OTT. OTT then produces the results in the form specified, and if the results contain a HTML report, the report is displayed in the GUI.

The requirements for the GUI were that it should be user-friendly and it should provide all the necessary features. The workflow and features were discovered using user studies with the users, see 4.2.1 for more about this.

4 Implementation

OTT is a plugin-based tool written in Python and Java. It is able to interface to FMI and Modelica compliant tools either through Python interfaces or sub-process calls.

4.1 OTT Core

OTT Core contains functionality for collecting and performing static and scripted testing. It also contains abstraction layers for the different test steps, compilation, simulation and verification.

4.1.1 Overview

For static testing OTT uses the Optimica Compiler Toolkit (OCT)[2] to traverse Modelica libraries. During the traversal the user configurable sieve is used to collect tests by filtering the target library. Depending on the user's command OTT will then take the tests through the different test steps. Each test step provides one or more tool implementations. Current version of OTT supports OCT and Dymola both for compilation and simulation. Result verification is currently done using CSV Result Compare tool (CSV compare) developed by ITI GmbH, see figure 2.

Each test produces a test report containing information collected during execution. The test report is in an intermediate format which can be converted into any type of presentation format. OTT has a presentation layer which, like the tool abstraction layer, relies on plugins. OTT has the following set of default output plugins:

- HTML, produces reports in human readable form, see figure 3.
- JUnit, produces reports in machine readable form, suitable for build servers.
- Pickle (Python), produces serialized Python test report objects.
- Hash, produces a file mapping model names to hashed filenames.

The main entry point to OTT is through the MRTT command line program. It allows the user to specify a wide range of settings, such as: target library, what tools to use for the different steps, output type and tool specific settings. An overview of the OTT Core can be found in figure 4.

Scripted testing works in a similar fashion as static. However, unlike static tests, scripted tests control the execution flow. OTT only facilitates integration to supported Modelica and FMI tools. OTT also simplifies the generation of various report artifacts by providing access to the presentation layer. This allows the user to focus on authoring tests instead of writing report files, such as HTML and JUnit reports.

4.1.2 Jenkins Integration and JUnit

OTT can easily be integrated into common Continuous Integration (CI) frameworks such as Jenkins and Hudson. This is done by configuring OTT to output a JUnit test report. This report is then parsed by the CI framework. The JUnit report contains status information for the different test steps, each with its own pass/fail flag. This enables the framework to detect changes in tests that changes between two failing states, i.e. if a model goes from compilation failure to simulation failure.

[2] http://www.modelon.com/

4.2 GUI

The OTT GUI allows the user to create, modify and execute tests. It was developed with usability in mind to ensure that it would be user-friendly.

4.2.1 User Feedback

We continuously evaluated the GUI by using the methods described by Lauesen (2005). Every iteration began with an evaluation of the GUI in the form of a user study followed by a response in the form of implementation in the GUI. The features that were implemented often directly addressed some usability concern.

Here are some important usability concerns we addressed:

- How to find the names of variables and parameters that will be included in the test.
- How to update the reference value of a test.
- How to view the results of the test.
- How to create many similar tests, and how to update them.

4.2.2 Features

In the GUI, as seen in figure 5, variables and modifiers (parameters) are displayed in tree views. Every tree view has a filter to make it more flexible to navigate the view.

The GUI has support for test inheritance, primarily to make it easier to create many similar tests. Test inheritance means that a subtest can be created to an already existing test. The subtest inherits all of the included variables and modifiers of the parent. The subtest can then change the value of those modifiers and add additional modifiers or variables. If the parent tests is updated all subtests will be updated. For example, as seen in figure 5, test c is a subtest of test a. Test a includes the modifiers `driveAngle` and `inertia1.J`. Test c inherits these two modifiers and also changes the value of `inertia1.J`. Test c also includes its own modifier `inertia2.J`.

After a test or suite of tests are run, the results will be displayed in the GUI. Reference results can be updated by pressing a button in the displayed results file. This allows the modeler to overwrite the old reference value if the new value is deemed more appropriate. When updating the reference, all variables specified in the test are updated.

Some basic features included in the GUI are: undo/redo operations, keyboard shortcuts and a run configuration.

DOI 10.3384/ecp15118687

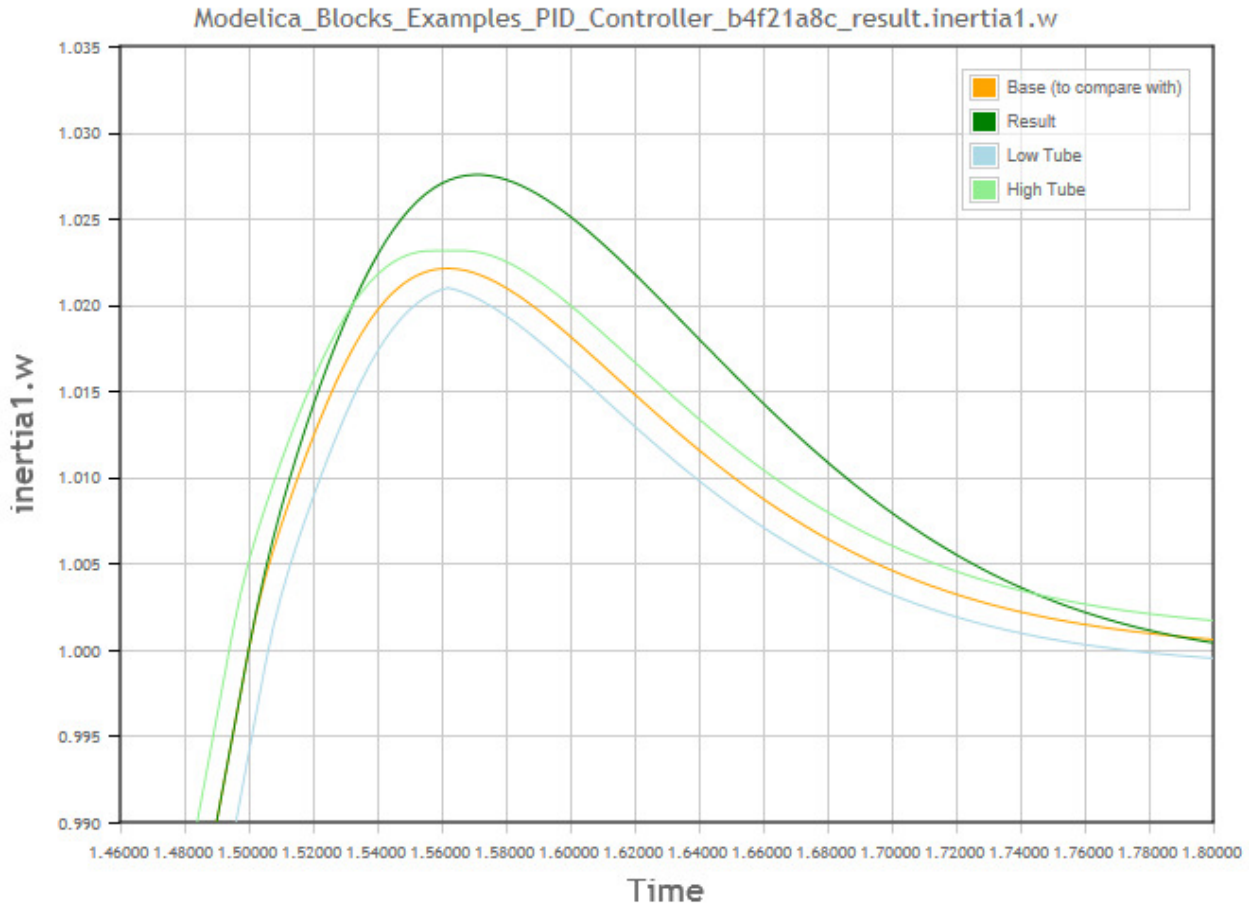

Figure 2. One variable compared to the reference value in the result file.

Verification Report

Modelica.Blocks

Models	Compilation	[s]	Simulation	Time [s]	n	s]	Rate
Modelica.Blocks.Examples.PID_Controller	pass	4.46	pass	0.41	pass	1.91	100%
Modelica.Blocks.Examples.Filter	pass	5.81	pass	0.92	fail	n/a	Either the result file or the reference file does not exist
Modelica.Blocks.Examples.FilterWithDifferentiation	pass	5.53	pass	0.48	pass	0.96	100%
Modelica.Blocks.Examples.FilterWithRiseTime	pass	4.69	pass	0.36	pass	1.11	100%
Modelica.Blocks.Examples.InverseModel	pass	3.29	pass	0.46	pass	0.72	100%
Modelica.Blocks.Examples.ShowLogicalSources	pass	3.34	pass	0.27	pass	0.42	100%
Modelica.Blocks.Examples.LogicalNetwork1	pass	3.37	pass	0.35	pass	0.48	100%
Modelica.Blocks.Examples.RealNetwork1	pass	3.40	pass	0.41	pass	0.91	100%
Modelica.Blocks.Examples.IntegerNetwork1	pass	3.64	pass	0.41	pass	0.87	100%
Modelica.Blocks.Examples.BooleanNetwork1	pass	4.25	pass	0.40	pass	1.11	100%
Modelica.Blocks.Examples.Interaction1	pass	4.25	pass	0.41	pass	0.95	100%
Modelica.Blocks.Examples.BusUsage	pass	3.32	pass	0.35	pass	0.57	100%
Summary	Passed compilation: 12/12		Passed simulation: 12/12		Passed verification: 11/12		

Figure 3. The results from testing a package with OTT.

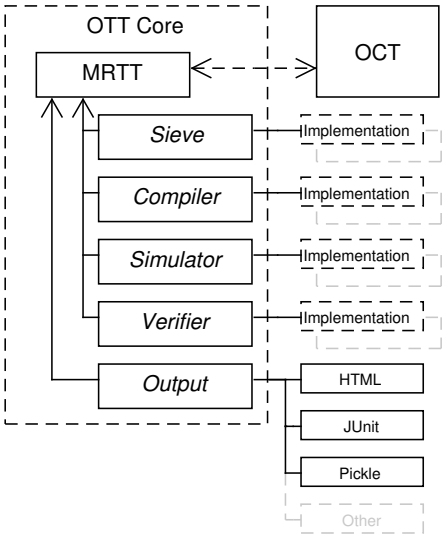

Figure 4. Overview of OTT Core

5 Conclusion and Future Work

In this paper, we presented a solution for tool agnostic regression testing in Modelica. By using a plugin like structure we have shown that it is possible to achieve a clear separation between the different testing steps. This allows us to use one Modelica tool to compile the model and another FMI tool to simulate the model, thus providing true cross-testing of Modelica libraries and tools. We have also shown why both scripted and static testing is necessary in Modelica development and how it can be implemented in a testing framework. The plugin structure also facilitates extendable and customizable test reports. One demonstration of this extendability was exemplified by showing the integration to automatic build systems such as Jenkins/Hudson by creating a output module that writes JUnit reports.

Additionally we present a GUI which enables the user to do test authoring, test execution and viewing of test results. We show how the GUI can improve the efficiency of test authoring by providing tools for efficient selection of test variables and parameters in the test model. We also show how the usability of the GUI was improved using iterative user studies and development.

In the future we plan to extend the number of supported Modelica and FMI tools, which will further strengthen the cross-testing capabilities. In order to improve the usability of the GUI we plan to integrate a graphical model viewer that has previously been implemented (Sten, 2012). Likewise we plan to render model icons correctly by integrating a previously developed icon rendering framework (Olsson and Moraeus, 2011).

References

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

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

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

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.

Open Source Modelica Consortium. *Modelica Compliance Library Guide*. 2013.

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

ITI GmbH. Csv result compare tool. `https://github.com/modelica-tools/csv-compare`. Accessed: 2015-05-19.

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

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

Roland Samlaus, Mareike Strach, Claudio Hillmann, and Peter Fritzson. *MoUnit - A Framework for Automatic Modelica Model Testing*. Proceedings of the 10th International Modelica Conference, 2014. doi:10.3384/ecp14096549.

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

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

Anders Tilly and Victor Johnsson. Developing a test authoring tool for a modeling language. Master's thesis, Lund University, 2015.

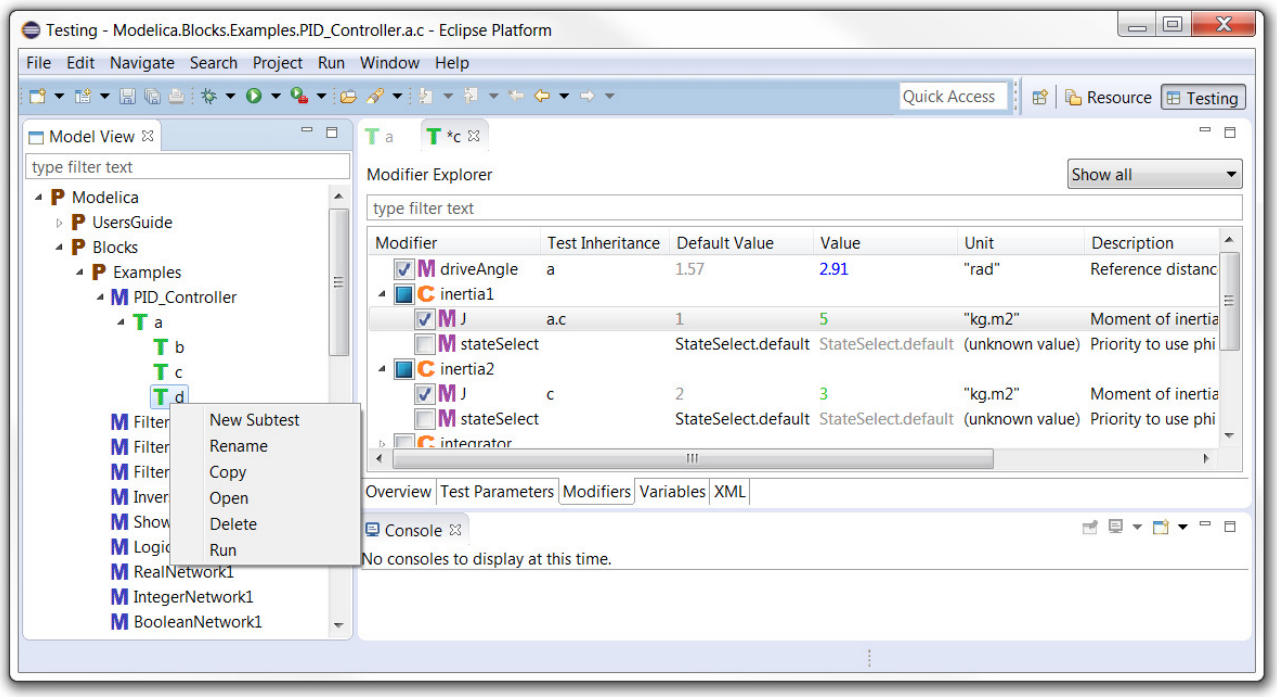

Figure 5. The OTT GUI