MoUnit — A Framework for Automatic Modelica Model Testing

Roland Samlaus[1] Mareike Strach[1] Claudio Hillmann[1] Peter Fritzson[2]
Fraunhofer IWES, Turbine Simulation, Software Development, and Aerodynamics[1]
Department of Computer and Information Science Linköping University[2]

Abstract

A vital part in development of physical models, i.e., mathematical models of physical system behavior, is testing whether the simulation results match the developer's expectations and physical laws. Creation and automatic execution of tests need to be easy to be accepted by the user. Currently, testing is mostly performed manually by regression testing and investigation of result plots. Furthermore, comparisons between different tools can be cumbersome due to different output formats. In this paper, the test framework MoUnit is introduced for automatic testing of Modelica models through unit testing. MoUnit allows comparison of Modelica simulation results with reference data, where both reference data and simulation results can originate from different simulation tools and/or Modelica compilers. The presented test framework MoUnit brings the widespread approach of unit testing from software development into practice also for physical modeling. The testing strategy that is used within the Modelica IDE OneModelica from which the requirements for MoUnit arose, is introduced using an example of linear water wave models. The implementation and features of MoUnit are described and its flexibility is exhibited through two test cases. It is outlined, how MoUnit is integrated into OneModelica and how the tests can be automated within continuous build environments.

Keywords: MoUnit, OneModelica, Modelica, test framework, automatic testing, verification

1 Introduction

An important part of the process of developing physical models is model testing since it greatly increases the probability that the expected behavior is accurately modeled. Within software development it is convenient to create tests and test suites using test frameworks like JUnit[1]. The use of these test frameworks ultimately lead to a "test first" development approach, i.e., creating tests before actually developing the software parts that must conform to the tests. Also, engineers creating physical models can benefit from a test driven development. Immediate feedback about incorrect model changes can assist in keeping the models consistent during their lifecycle in the modeling process.

By using fine-grained tests for single components, errors in composed models can be traced back to the erroneous element. This also simplifies the development in teams as one modeler's changes may have a negative influence on another modeler's models. However, test-based development as known from software engineering is hard to maintain for physical model developed with Modelica. While the algorithmic parts of models could be checked in a similar way, the behavior expressed by equations cannot be tested directly since the usage of time as a parameter makes it necessary to perform simulations.

A reasonable solution for this is the use of regression tests where reference data is used as a base for comparison with results from subsequent simulations. When models are altered, the result can be compared to the reference files to check if the behaviour is still as expected. The two commonly used Modelica simulation tools OpenModelica[2] and Dymola[3] use regression tests for checking if changes made in the compilers still result in valid simulations as well as to allow users to create regression tests for model development.

Although these kind of tests already aid the development process of physical models, regression tests may not be sufficient to find all problems that may appear during the development and the tools provide a non-uniform way of defining test cases. With our test framework MoUnit we aim at facilitating test case and test suite definition for single model components as

[1] http://www.junit.org
[2] https://www.openmodelica.org/
[3] http://www.dymola.com

well as groups of models. Besides, the tests should be executable with various simulation tools as well as in different environments, e.g., directly inside the Modelica IDE OneModelica[4], or in continuous build environments like Hudson[5], which is currently in use for OneModelica and OpenModelica development. Moreover, we want to allow users to extend MoUnit in order to re-use their already implemented test functionality as well as making it possible to create more sophisticated tests then regression tests.

In this paper, we demonstrate how the user can create test cases for automatic testing. The user can use arbitrary simulation tools, compare the results to reference data and to the results of a second experiment. Thereby, it is possible to compare simulation tools to each other or investigate the impact of different solvers on the simulation results without the need to create additional reference files. Moreover, we demonstrate the extensibility of the test framework for example through the implementation of a test component for regression test of equidistant result files. As a second tester component we implemented the calculation of basic statistics. This allows us to validate stochastic properties of our Modelica irregular water wave models by comparing to results from the commercial software ASAS WAVE[6]. Here it is advantageous that several input formats are supported in order to use data from tools that are not Modelica-based. Thus, calculations from analytic formulae producing numerical results can also be used for testing in our framework.

The remainder of this paper is structured as follows: In Section 2 we describe the project structure of Modelica projects in OneModelica. Separation into projects allows to separate code for testing from models containing the physical behavior. The test language is described in Section 3 along with an explanation of how the test framework can be extended. The evaluation in Section 4 demonstrates how the water wave models developed at Fraunhofer Institute for Wind Energy and Energy System Technology (Fraunhofer IWES) are tested using our framework. Finally, Section 5 concludes our work and gives an outlook on how we plan to further enhance the test framework in the future.

[4]http://www.onewind.de/OneModelica.html

[5]http://hudson-ci.org/

[6]http://www.ansys.com/Products/Other+Products/ANSYS+ASAS

2 Testing Modelica Models Within OneModelica

Testing is a crucial part of model development. The following aspects of a model should be checked: whether it (a) can be simulated without errors, (b) delivers the expected results, and (c) represents the reality appropriately. Furthermore, when component models are developed and/or modified it also has to be checked whether the composed models deliver the expected results. That is, the robustness of a component model in various situations has to be ensured as — depending on the complexity of the model — even a small change might lead to unexpected results. Testing a model in different situations can be a tedious process as it involves a lot of “playing around with parameters” and waiting for simulations to finish. All these points are reasons for automated testing during model development.

The testing strategy which is presented herein and from which the requirements for the development of MoUnit arose, is closely tied to the Modelica IDE OneModelica. In OneModelica, models are organized in model projects [2]. This approach allows to have two model projects for each component model that has to be implemented. The first one contains all the physical system equations and model logic. However, the models are not yet simulatable because the parameters do not have values yet — as they should be modified by the user when the model is parameterized for various application scenarios. The second one contains the actual test models. The test models contain instances of the models taken from the original model project. The modifications of the original models are done with parameter settings that correspond to defined test case scenarios. This could also be a special set of parameters that has caused problems in the past and it should be ensured for future simulations that it will not occur again. This strategy is somewhat borrowed from the software development: JUnit test plug-ins will be separated from the actual Java plug-ins to divide program logic from the test. The use of instances of the models to be tested rather than extending it, is based on the fact that test models can also be seen as “best practice” examples: a look at the test models shows how the model to be tested should be used. However, if this aspect is not important, it is also possible to use a test model that extends the model to be tested within MoUnit. This would avoid possible inconsistencies between the physics and test model code that is duplicated such as parameter units. Using the ex-

DOI 10.3384/ECP14096549

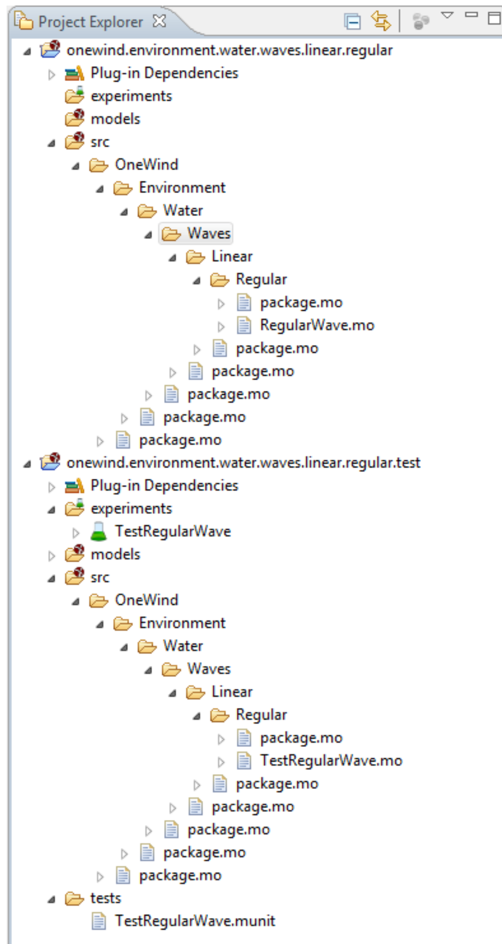

Figure 1: Model projects for both model logic and test models for regular water waves

```
model RegularWave
  "kinematics of a regular linear wave"
import SI = Modelica.SIunits;
import NonSI =
  Modelica.SIunits.Conversions.NonSIunits;

parameter SI.Height H "wave height";
parameter SI.Period T "wave period";
parameter NonSI.Angle_deg epsilon "phase";
// rest of parameters omitted

equation
  // model logic is placed here
end RegularWave;

model TestRegularWave
  "test model for ‘RegularWave‘"
import SI = Modelica.SIunits;
import NonSI =
  Modelica.SIunits.Conversions.NonSIunits;

constant SI.Height H = 5;
constant SI.Period T = 11;
constant NonSI.Angle_deg epsilon = 12;

RegularWave regularWave(H = H, T = T,
  epsilon = epsilon
  // rest of modifications omitted
  ) "instance to be tested";

equation
  // test logic like assert statements
  // is placed here
end TestRegularWave;
```

Listing 1: Example for model and corresponding test model

ample of Modelica models for regular water waves, this is illustrated in Figure 1: The test model project has the same name as the original model project, but with an appended `.test`. This makes it clearly distinct as a test project for all the developers, who work on the models. As can also be seen in Figure 1, the test model project contains experiments that simulate the test models. The structure of the model to be tested and the testing model is shown in Listing 1: the model `RegularWave` has the parameter instances and equations, while the model `TestRegularWave` has an instance of type `RegularWave` and modifies the corresponding parameters. The experiments that are used to simulate the test models are set up manually. They can also be simulated manually during the model development. However, once the number of models and functions has exceeded some threshold been created, manual execution of all the experiments becomes too cumbersome. This is the point where MoUnit should be used. The necessary files for automated execution of the experiments are stored in the test model project as shown in Figure 1. These files enable checking that all the models that have been implemented in the original model project are fulfilling the three properties mentioned above: it can be easily seen if a simulation crashes and if the implementation of the model works correctly by comparing it to reference data that can either come from other simulation tools, measurements, or analytical solutions.

3 Implementation of the Modelica Test Framework

For the development of valid physical models with Modelica it is vital to define test cases to check that model changes do not negatively affect simulation results. A common way to assure this is using regression tests. This is used by the OpenModelica development group using their framework to test compiler changes by comparing results of simulations to reference data. In the same way Dymola's model management library allows performing regression tests with

Modelica models.

The goal of MoUnit is to provide test-driven physical model development with Modelica. For this purpose we implemented a test framework that is integrated into the Modelica IDE OneModelica and is flexible to be also used in automatic build environments like Hudson. A special test language has been defined that can be parameterized with custom test components and test result listeners which are responsible for processing test results appropriately.

In the following, the test language is described that has been developed with Xtext [3]. It allows defining test cases and test suites for automatic execution. Additionally, the extension mechanism is explained to show how custom test components can be registered allowing users to write test cases for special purposes — for example, basic statistics analysis. Subsequently, the test runner that performs the test execution is described and the registration of listeners is demonstrated. The listener mechanism allows to use MoUnit in different environments, e.g., by displaying the test results visually inside OneModelica or by generating text files for the integration of tests in continuous build environments like Hudson. We have implemented an ANT task that is executed by Hudson. The task adds a listener to the test runner creating a XML file for the documentation of the test results. The result file can then be used by Hudson for error reports.

3.1 Test Definition Language

Xtext is a tool for textual language development. By providing a grammar definition of a language, editors and views can automatically be generated with helpful functionality like syntax highlighting and syntax checking. The Modelica editor being the basis for OneModelica has been developed with Xtext. Parsed documents are represented as Ecore-based [4] parse trees and can be used along with other Ecore-based languages (e.g., cross-references between elements of different languages are supported). Since mandatory components for automatic test execution in our IDE are defined as Ecore-models, those components can easily be re-used for MoUnit.

Two components can be defined with the test language that are comparable to concepts used by the test framework JUnit for Java code: Test cases and test suites. Test cases (see Listing 2) define the basic elements that are needed for a test to run while test suites (see Listing 3) combine test cases and other test suites. Hence, a set of test cases can be accumulated to test the correct behavior of interacting Modelica models.

```
name TestCase1
experiment ToTest
file "pathToFile.mat"
compare {
  qualifiedname.*
  test.qn -> reference.qn

  TestComponent {
    attribute = 1.e-5
  }
  error "Error message"
}
```

Listing 2: Example of a Test Case Definition

```
suite AllTests:

  TestSuite1,
  TestCase1
```

Listing 3: Example of a Test Suite Definition

Test cases define names that make them referenceable and thus usable in test suites. The `experiment` keyword states that a Modelica experiment is used in the test. Modelica experiments in OneModelica can use several tools for simulation, including OpenModelica and Dymola. For regression tests a path to a reference file is provided following the keyword `file`. Several checks were implemented to display problems to the user, e.g., when the provided file cannot be found or the experiment is set to leave the Dymola window open after simulation for post processing. This would cause the test execution to pause until the user closes the window manually. It is also possible to directly compare the test results of two experiments by using a second experiment instead of the reference file. This can help to ensure that the results of simulations with OpenModelica are the same as those obtained from simulations with Dymola. Another possibility is to check if the results are equal for different kinds of solvers.

It is also possible to compare two result files with each other, e.g., when it is desired to compare results from other modeling tools and languages. The result files can be in .mat format as provided by OpenModelica and Dymola (textual or binary format) as well as in comma separated values format. Moreover, the user can also create results based on some analytical formulae, if it is not desired to implement the analytical result as a test component to register it to the test framework, and save them in one of the supported formats instead.

Simulation result files can be referenced by abso-

DOI
10.3384/ECP14096549

lute or relative paths. Using relative paths, files can be put into a project inside OneModelica's workspace and can be synchronized by multiple users with source code management tools like SVN. Since simulation result files might be large, it may be necessary to use network drives for storage. This can be done by using absolute paths that can be prepended with the prefix $. The prefix will be replaced by a user defined path. This way tests are re-usable for other users, and the prefix only has to be configured once per workspace.

Now that the data for comparison is available, the user needs to provide the qualified names of model elements that shall be compared, following the `compare` keyword. If no element is defined then all time series from the results are compared to each other. The user can use the asterisk character * to select groups of elements.

Next, test components are referenced that perform the comparison of results. The components are registered via an Eclipse extension point [5, 6]. The test components must implement a predefined interface and need to be modeled with Ecore. Attributes can be defined which must be parameterized by the user. In the provided example a tolerance for the result comparison is defined which will allow a small difference in the result data of the compared time series.

Possible values for the parameterization are Doubles, Integers and arrays of Doubles and Integers. The attributes are looked up by introspection [9] which is used to validate whether the user provides correct input. When the validation is performed a new instance of the test component is created and the parameters are set. Finally the user defines an error message that will be displayed when a test run fails.

Test suites start with the keyword `suite` followed by a name. They contain a list of test cases and additional test suites. All test cases contained in a test suite will be performed when the suite is executed.

3.2 Test Execution and Result Processing

The test definitions are processed by a Modelica test runner. The runner is implemented without dependencies to user interface components to allow usage in build environments that do not have graphical user interface installed (e.g., Linux operating systems on dedicated build servers). The test runner executes the experiments used in the test cases and compares the results to the referenced experiments or files. Listeners can be registered to the test runner to be informed about the executed tests and test results. The duration of each test is measured for performance analysis. The

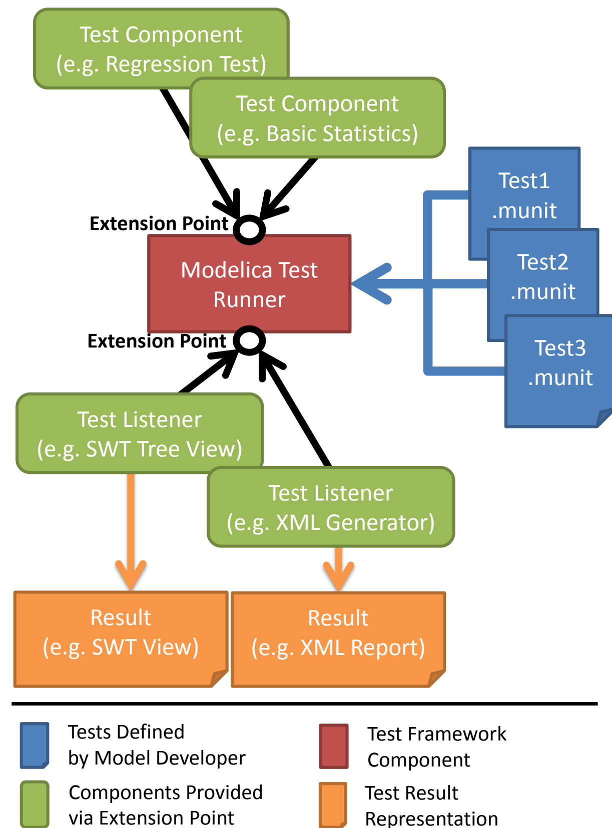

Figure 2: Structure of the Modelica Test Framework

test components must return a status object that provides information about whether the test succeeded or failed. When a test failed, a message must be provided and an optional index (e.g., the time step) can be used to tell the user about the erroneous location in the result file.

Besides the basic information described above, a test component can return arbitrary objects with additional information about an error. This can, for example, be an SWT[7] composite object for visualization inside OneModelica or a file path to a serialized plot which can be used for a test report. The test runner listener is responsible for checking whether it can handle the object or not.

For the visualization of test results in OneModelica we implemented a view (see Section 4) that is comparable to JUnit's test view. Test suites and test cases are displayed as a tree and the nodes are marked according to successful or erroneous execution. While JUnit test cases are based on assert statements that check whether particular parts of code run correctly, we need to compare result files to check whether the desired behavior is matched. This results in a different kind

[7] http://www.eclipse.org/swt/

of test status visualization where the qualified names of erroneous models is provided and optional graphs can visualize deviations between expected and present values. A JUnit like behavior could be implemented for algorithmic code, e.g. by interpreting the algorithms [1], allowing to check code very fast. However, this is currently not yet supported by MoUnit.

For the integration with automatic build environments we developed an Eclipse ANT task[8]. The task is performed on an Eclipse workspace that contains the Modelica projects as described in Section 2 with experiments and test definitions. The user provides information about which test cases and test suites shall be executed. For test logging we implemented a test runner listener that creates XML files according to the format used by the JUnit ANT task. Hence, failing Modelica tests can be handled in the same way as JUnit tests, e.g. by sending out notification mails to the responsible model developers. The ANT task enables the use of the same test definitions in continuous build environments as well as for manually triggered testing inside OneModelica.

4 Evaluation

In the following, the example of linear water wave models is used for the evaluation of MoUnit. These models were developed at Fraunhofer IWES to display both regular and irregular waves according to Airy wave theory with corresponding stretching methods. The underlying theory for these models can be found in standard offshore engineering text books like [7] or [8].

For evaluation purposes, the variable `eta` corresponding to the wave elevation, i.e. the instantaneous wave height at a given position, is used. For a regular linear wave, `eta` is deterministic as it can be displayed by a cosine function. That is, it can be compared to reference data through a regression test. Accordingly, this is used as a first test case denoted as `TestRegularWave`.

The second test case is called `TestIrregularWave` and tests the variable `eta` for an irregular linear wave. In this case, `eta` is stochastic, i.e., a direct comparison of time series to reference data is meaningless unless they are generated with the exact same input. However, this is not always possible, e.g. when the simulated time series are compared to reference data from a different tool or measurements. Instead, the mean

[8]http://ant.apache.org/

```
name TestRegularWave

experiment
TestRegularWheelerWave
file
"$/WAVE_WheelerElevation.csv"
compare {

  regularWaveKinematics[1].eta
  -> asas.elevation.eta

  EquidistantTSValidator {
    tolerance=10e-11
  }
  error "Wave elevation does not match."
}
```

Listing 4: Test Definition for Regular Waves

value and the standard deviation of the time series are compared to those of reference data.

For the test case `TestRegularWave` (see Listing 4), we implemented the test component `EquidistantTSValidator`. It can be parameterized with a `tolerance` that represents the acceptable absolute difference between the two compared time series. The input files for the component must have the same length as well as equidistant values. If the input does not match the reference data, i.e., if the absolute difference between input and reference data exceeds the value of `tolerance`, an error will be displayed. The resulting view for our example is shown in Figure 3: the simulation time plus the time used for validating the values is displayed as a sum in the test case tree view. When the user selects the erroneous test, basic information about the location of the first assert failure is provided together with a plot of the two compared curves.

For the test case `TestIrregularWave` (see Listing 5) a test component named `BasicStatistics` has been implemented to compare mean value and standard deviation as described above. The user can provide the parameters `startindex` and `endindex` to define the range of values that shall be checked for the provided input values.

```
name TestIrregularWave

file
"$/ir_periodical_elevation_dymola.csv"
file
"$/ir_elevation_asas.csv"
compare {

  modelica.elevation.eta
  -> asas.elevation.eta
```


DOI
10.3384/ECP14096549

```
  BasicStatistics {
    startindex=0,
    endindex=800,
    meandeviation=0.01,
    stddeviation=0.001
  }
  error "Basic statistics of wave
      elevation do not match."
}
```

Listing 5: Test Definition for Irregular Waves

Additionally, the user can define the allowed variance of the calculated mean values and standard deviations. Figure 4 shows the resulting view displaying information about the assertions that were violated. While the mean value of the wave elevation matches the reference data, the standard deviation is not acceptable. As for the test case `TestRegularWave`, a plot is provided to visualize the compared values to the user.

5 Conclusion and Outlook

In this paper, we demonstrated that the test framework MoUnit developed at Fraunhofer IWES can effectively be used for testing Modelica models by unit testing. We showed that the framework is flexible as it supports multiple simulation tools as well as input files from other tools for result comparison. This allows to check the different behavior of models when using different simulation tools as well as different solvers.

We also illustrated that it is possible to extend MoUnit with custom test components that provide functionality different from regression testing. Using the example of simulating water waves, we compared simulation results of our Modelica models to simulation results of the reference tool ASAS WAVE. The extensibility also allows to add test components that use results from analytical calculations for result validation.

For error reporting, a flexible interface has been defined. We implemented a plot for wave model tests displaying the time series of the data that is being compared together with the corresponding mean values and standard deviations. Thereby, the user can directly investigate the cause for a problem visually. Furthermore, an ANT task has been implemented that executes test suites and registers a listener that serializes a test report in an XML file compatible to JUnit. Hence, the tests can be integrated into a build system for continuous testing.

In the future, we plan to create additional test functionality that allows to test single values of result files. The implementation of test components for analytical solutions are planned for the models of wind turbine components — as far as they are available. For fast testing of algorithms, it is furthermore desirable and planned to interpret the algorithmic code, e.g. of Modelica functions to allow direct calls and comparison to expected result values.

Acknowledgements

This work is financially supported by the Federal Ministry for the Environment, Nature Conservation and Nuclear Safety based on a decision of the Parliament of the Federal Republic of Germany, by the Swedish Government in the ELLIIT project, by the Swedish Strategic Research Foundation in the EDOp projects, and by Vinnova in the RTSIM and ITEA2 MODRIO projects.

References

[1] Höger C. Modelica on the Java Virtual Machine. In: Proceedings of the 5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, April 19 2013, University of Nottingham, Nottingham, UK.

[2] Samlaus R, Hillmann C, Demuth B, Krebs M. Towards a Model Driven Modelica IDE. In: Proceedings of the 8th International Modelica Conference 2011, Dresden, Germany, Modelica Association, 20-22 March 2011.

[3] Köhnlein J., Efftinge S.,Xtext 2.1 Documentation, Itemis GmbH, October 31, 2011.

[4] Budinsky F, Brodsky S. A., Merks E. Eclipse Modeling Framework, Pearson Education, 2003.

[5] Clayberg E, Rubel D. Eclipse Plug-ins, Addison Wesley Professional, 2009.

[6] McAffer J, van der Lei P, Archer S. OSGi and Equinox: Creating Highly Modular Java Systems, Addison-Wesley Professional, 2010.

[7] Faltinsen O M. Sea loads on ships and offshore structures, Cambridge University Press, 1990.

[8] Chakrabarti S K. Ocean Environment. In: Chakrabarti S K (ed.), Handbook of Offshore Engineering, Elsevier, 2005.

[9] Eckel B., Thinking in Java, Prentice Hall, 978-0-13-187248-6, 2006.

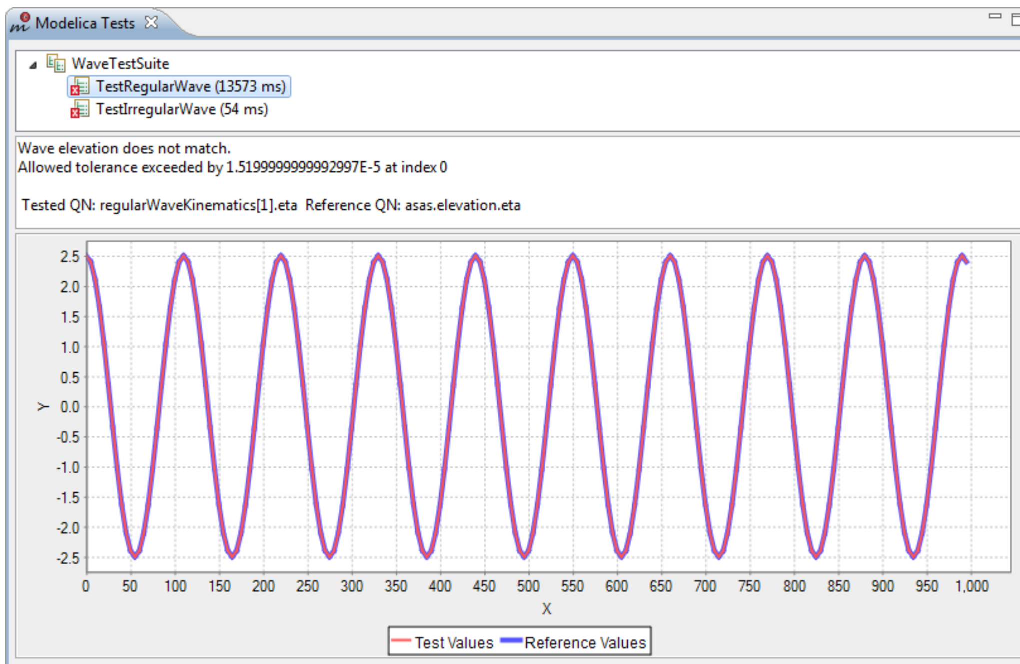

Figure 3: Result for test case `TestRegularWave`

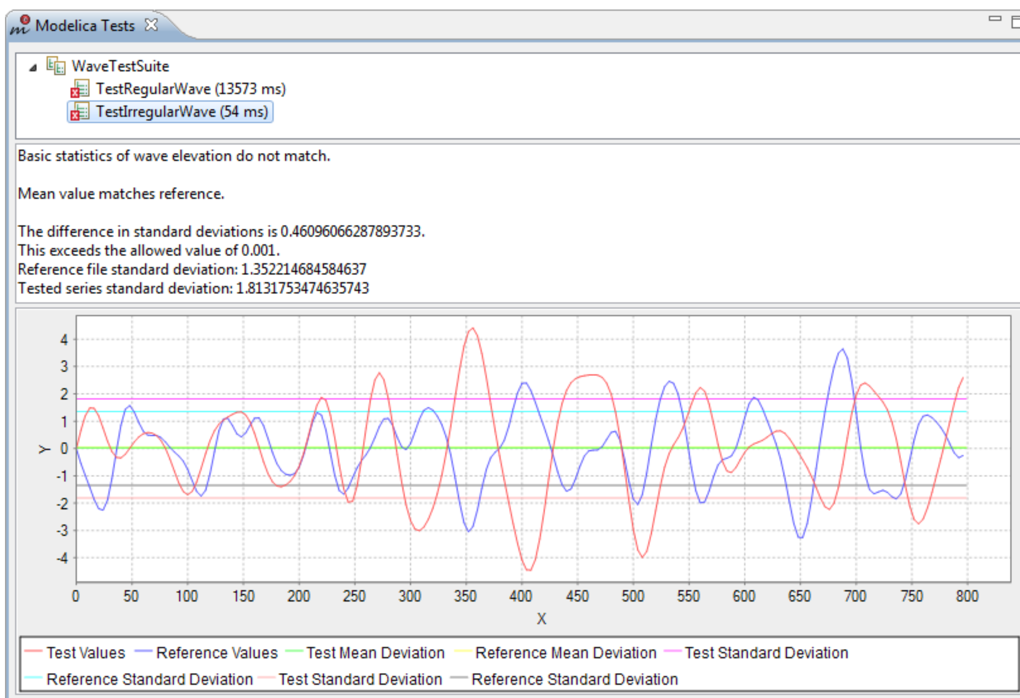

Figure 4: Result for test case `TestIrregularWave`

DOI 10.3384/ECP14096549