OMC Test suite users guide

15/01/2013 : Draft 1 – Francesco Casella: specification
26/02/2013 : Draft 2 – Bruno Scaglioni: testcases howto
28/02/2013 : Draft 3 – Lennart Ochel: minor fixes (format, typos, …)

Specification of the OMC Test Suite Functionality

Goals of the testsuite

- ensure that all legal Modelica classes are handled correctly by OMC
- ensure that all illegal Modelica classes are reported as such by OMC
- ensure that a large set of simulation models (including examples from the MSL and from ModelicaTest) can be compiled and produce correct simulation results
- help the developers identify problems with OMC and solve them
- help the OSMC board understand the progress of the development work
- make the status and progress of the development of OMC clearly visible for the general audience

What needs to be tested?

Modelica classes undergo several stage of transformation, ultimately leading (only for simulation models) to simulation results. For the sake of convenience, several milestones in this process can be identified:

1. Flattening and static checking terminates as expected (success for correct Modelica classes, failure for intentionally wrong Modelica classes)
2. Simulation code generation terminates successfully
3. The simulation executable runs successfully until the end time is reached
4. The simulation results are correct

The first step applies to all Modelica classes, while the other ones are only applicable to simulation models (i.e., models having the StopTime annotation).

What needs to be reported?

In order to assist the developers and the managers, by providing meaningful feedback, it is important to report separate results for each milestone. Assuming, for example, that the current status is such that the first three milestones are successful and the fourth fail on a given model, it is important to see whether the effect of a change to the compiler source code leads to correct simulation results, leaves the situation unchanged, or possibly even hampers the successful generation of simulation code.

Ideally, the outcome of the testsuite run should be reported in a table like the following one:

Class name	Flattening and static checking as expected	Sim code generation successful	Simcode runs until stopTime	Correct sim results
PackageA.modelA	PASS	PASS	PASS	PASS
PackageA.modelB	PASS	PASS	PASS	FAIL
PackageA.modelC	PASS	PASS	FAIL	FAIL
PackageA.modelD	PASS	FAIL	FAIL	FAIL
PackageA.modelE	PASS	N/A	N/A	N/A
PackageA.modelF	FAIL	N/A	N/A	N/A
PackageA.functionA	PASS	N/A	N/A	N/A
PackageA.functionB	FAIL	N/A	N/A	N/A
PackageA.connectorA	PASS	N/A	N/A	N/A
PackageA.connector B	FAIL	N/A	N/A	N/A

In this case, the first four models are simulation models, while the fourth and fifth are not. Also, modelE might actually be wrong for some reason (e.g., because it is unbalanced), but if the test is contrived in order to check that the compiler actually detects this problem, the outcome can be PASS. The opposite case is also possible.

Note that, for a given class, in order to have a PASS result from the n-th milestone, all the previous milestones also necessarily give a PASS result. Based on this property, it is possible to devise efficient schemes in order to avoid repeating the early stages multiple times. For example, if the simulation code runs until stopTime, it is unnecessary to separately check the flattening and code generation steps, as they must necessarily give a positive outcome. Also, it is wise to first start testing the highest milestone that PASSed during the previous run of the testsuite, then try the next one if it PASSes, or the previous one if it FAILs.

It is also useful to report changes w.r.t. a previous run of the testsuite: this helps developers understand the effects of their work, and managers understand the progress in the code development. For example, one could build a table with the same text as above, changing the color code so that grey means no change, green means change from FAIL to PASS, and read means change from PASS to FAIL.

When are the tests to be run?

Ideally, every time a change is committed to the compiler code base, the compiler should be re-built and the entire test suite should be run. If the computing resources to do this are insufficient, an enquiry could be made among the members of the OSMC, if they are willing to provide access to workstation to perform this task.

How to decide that the simulation results are correct?

Ideally, the library developers should also provide reference results which they declare, under their responsibility, to be correct. As this is not feasible in practice, the following procedure is implemented instead.

Since Dymola has a good reputation as a solid simulation engine, reference simulations are produced by means of this tool, using default solver settings. Scripts have been developed that

allow to automatic producing simulation results for all the models in a Modelica package having the StopTime annotation. This is really convenient, as the developer effort when including new packages in the test suite is very low.

Simulations are then run using OMC – if the results are close enough (in terms of relative precision), then the simulation results are declared correct. This approach is fairly reasonable, as it is quite unlikely that both Dymola and OMC produce the same (or almost the same) wrong result.

For the few cases where this comparison fails, an expert modeler inspects Dymola and OMC results and tries to understand why that is the case, possibly adapting Dymola's and/or OMC's simulation settings in order to produce a more accurate numerical solution of the model equations. It might be the case that some difference remains, e.g. due to the triggering of events at slightly different time instants, that can generate very large discrepancies of the results for a few data points. The expert modeler can judge whether this is acceptable or not. Once a result that can be qualified as correct has been reached with OMC, OMC's results can be directly used as a reference.

OpenModelica Testcases Structure

OpenModelica test cases are all run by an automatic system named Hudson (http://hudson-ci.org/). When a developer commits a change in OM svn repository, Hudson builds the OM executable and tests it against all tests contained in „…/Openmodelica/trunk/testsuite“.

All testcases must be placed under „/testsuite“ directory.

- /simulation contains all the models that test the simulation capabilities of OMC. If you write a testcase that should produce simulation results, put it here.
- /openmodelica contains tests for internal components testing
- /metamodelica contains tests for MetaModelica language testing

All testcases have the same structure, divided on two separate files:

- Model
- TestFile

The model is contained in a .mo file, it may be part of a library (like MSL or ModelicaTest) or a standalone .mo file. If the model is part of a library provided with OpenModelica testers don’t have to provide the model in a separate file, and the test should be in /simulation/libraries otherwise a .mo file containing the model should be provided.

The testfile is an OpenModelica Script (.mos) containing three sections:

1. Header
2. Body
3. Expected results

The following is an example of testcase:

Header:

```
// name:     Modelica.Mechanics.MultiBody.Examples.Elementary.PointGravity
// keywords:simulation MSL Examples
// status: correct
//
//
// Modelica Standard Library
//
```

The header contains a description of the test, mainly the name of the model, a keyword for automatic search and the current status of the test.
Note that every line is commented out (in Modelica sense), this file is a Modelica script but it is also read by Hudson, this first part is ignored by OMC.

Body:

```
loadFile("./common/ModelTesting.mo");
loadModel(Modelica,{"3.2.1"});
modelTestingType := OpenModelicaModelTesting.Kind.VerifiedSimulation;
modelName := $TypeName(Modelica.Mechanics.MultiBody.Examples.Elementary.PointGravity);
referenceFile := "./ReferenceFiles/Modelica.Mechanics.MultiBody.Examples.Elementary.PointGravity.mat";
stopTime := 0.0; // For reading stopTime from annotation
relTol := 0.01;
absTol := 0.0001;
compareVars :=
{"body2.frame_a.r_0[1]","body2.frame_a.r_0[2]","body2.frame_a.r_0[3]","body2.v_0[1]","body2.v_0[2]","body2.v_0[3]","body2.w_a
[1]","body2.w_a[2]","body2.w_a[3]","body1.frame_a.r_0[1]","body1.frame_a.r_0[2]","body1.frame_a.r_0[3]","body1.v_0[1]","body1
.v_0[2]","body1.v_0[3]","body1.w_a[1]","body1.w_a[2]","body1.w_a[3]"};
runScript("./common/ModelTesting.mos");
getErrorString();
```

The body is the core of the testcase, it contains the statements read by OMC. The content of this part will be inspected in the following sections

Expected results:

```
// Result:
// true
// true
// OpenModelicaModelTesting.Kind.VerifiedSimulation
// Modelica.Mechanics.MultiBody.Examples.Elementary.PointGravity
// "./ReferenceFiles/Modelica.Mechanics.MultiBody.Examples.Elementary.PointGravity.mat"
// 0.0
// 0.01
// 0.0001
//
{"body2.frame_a.r_0[1]","body2.frame_a.r_0[2]","body2.frame_a.r_0[3]","body2.v_0[1]","body2.v_0[2]","body2.v_0[3]","body2.w_a
[1]","body2.w_a[2]","body2.w_a[3]","body1.frame_a.r_0[1]","body1.frame_a.r_0[2]","body1.frame_a.r_0[3]","body1.v_0[1]","body1
.v_0[2]","body1.v_0[3]","body1.w_a[1]","body1.w_a[2]","body1.w_a[3]"}
// Simulation options: startTime = 0.0, stopTime = 5.0, numberOfIntervals = 500, tolerance = 0.000001, method = 'dassl',
fileNamePrefix = 'Modelica.Mechanics.MultiBody.Examples.Elementary.PointGravity', storeInTemp = false, noClean = false,
options = '', outputFormat = 'mat', variableFilter = '.*', measureTime = false, cflags = '', simflags = ''
// Result file: Modelica.Mechanics.MultiBody.Examples.Elementary.PointGravity_res.mat
// Files Equal!
// "true
// "
// ""
// endResult
```

The expected results section contains the output that the modeler expects to get from OMC in order to declare the test PASSED. If the result of OMC is different from this section Hudson will declare the test as failing.
This section is completely commented out (in Modelica sense) because it's parsed only by Hudson, the section starts with the statement: `// Result:` and ends with the stamenent `// endresult` both on a new line.
Everything contained in between these statements is verified by Hudson to be equal to the output of the preceding section.

Writing a testcase

Every model can be tested for several functionalities (as explained in the previous chapter):

- Instantiation
- Translation
- Compilation
- Simulation with suppressed warning
- Simple Simulation
- Simulation with results verified against a known source

The testcases should be built in such a way to report if the test fails (namely: gives a different output then expected) but it should also report if the expected milestone is outdated and the test now passed the following milestone. This is done in OM by some scripts designed to help the test writing, particularly a script called „ModelTesting.mos“ is capable of checking the behavior of OMC with respect to the milestones described in the previous chapter and to report if the test has passed the next milestone. This procedure is explained in next section.

The script is divided in two files:

- **ModelTesting.mo**: contains the enumeration describing the milestones, the code is reported:

```
type Kind = enumeration(Instantiation "Like instantiateModel()",
                  Translation "Like translateModel",
                  Compilation "Like buildModel",
                  SuppressedSimulation "Like simulate(), but suppresses output (only checks exit status)",
                  SimpleSimulation "Like simulate()",
                  VerifiedSimulation "Like simulate(), but also verifies the results against a known good source");
```

- **ModelTesting.mos**: contains the script that tests the current and subsequent milestone

The best way to understand how to write a test is an example, below is reported a MSL testcase (body section) and every statement is explained:

```
loadFile("./common/ModelTesting.mo");
```

This statement loads a modelica enumeration that lists the possibile testing procedures

```
loadModel(Modelica,{"3.2.1"});
```

This particular testcase is testing a MSL model so we must load MSL, it could be necessary to load other files, for example the „.mo“ file containing the test

```
modelTestingType := OpenModelicaModelTesting.Kind.VerifiedSimulation;
```

This variable must have this name, it describes the kind of test tob e performed.

```
modelName := $TypeName(Modelica.Mechanics.MultiBody.Examples.Elementary.DoublePendulum);
```

This variable must have this name, it contains the model name

```
referenceFile := "./ReferenceFiles/Modelica.Mechanics.MultiBody.Examples.Elementary.DoublePendulum.mat";
```

…reference file for result comparison…must be present if the testcase is simulation

```
stopTime := 0.0; // For reading stopTime from annotation
```

stopTime must be always present, if the value is equal to 0.0 the stopTime annotation will be used

```
relTol := 0.01;
absTol := 0.0001;
compareVars := {"revolute1.phi","revolute1.w","revolute2.phi","revolute2.w"};
```

statements for variables comparison, describe the tolerances and the variables tob e compared. Usually state variables are compared.

```
runScript("./common/ModelTesting.mos");
```

run the script for testing

```
getErrorString();
```

read errors buffer

The scripts mentioned above are in „./testsuite/simulation/libraries/common“.

This directory contains also some helpful python scripts for automatic testcases generation that will be explained in some following sections

Breaking a testcase could mean: getting better

As described above, the „ModelTesting.mos“ script tests the model for the chosen operation, but it tests also the next milestone in order. This means that a test passing the milestone **Translation** is also tested for **Compilation.** If Compilation is passed the script reports a warning in the output. This will formally break the testcase in Hudson, but inside the output log users can read that testcase has reached next step. It is then a modeler duty to update the test.
In this context a Hudson failing testcase could mean getting better in terms of simulation process. The testcase writer has the responsibility of distinguish between good and bad broken testcases.

Adding testcases to the suite

There are two possible scenarios when adding new testcases:

- Add a single testcase in an existing directory:
  - Add the mos file in the selected directory
  - Add the reference file to the appropriate directory
  - Open the Makefile in the directory where you put the mos testfile
  - Add your testcase mos file to @TESTFILES list

- Add a group of testcases in a separate directory
  - Create your folder.
  - Copy the file Makefile_sample.txt (located in testsuite directory) to your directory. Rename it to Makefile.
  - Add your test files (*.mo and *.mos) -> @TESTFILES
  - Add any other files that are needed (e.g. C files with external functions ...) at->@DEPENDENCIES
  - Add the folder

Automatic testcases generation

TODO – not complete
The „common“ directory contains a python script designed for automatic testcases generation, for the time being it has been tested only on MSL models so some changes could be necessary in order to use it with other models.
The scripts generate results files with dymola and use OMPython for checking the OMC results when trying to simulate a model, so in order to use this script you need to have
-Dymola, if you want to generate results file automatically
-OMPython
-There is an additional function in the script, that automatically builds a list of testcases from a library, but it requires an additional capability of Dymola named: ModelManagement, which is not provided with the basic license.

Additional commands

These additional commands can be inserted in the header of the mos file if special configurations

are necessary.

```
cflags: +d=xyz
```

Will insert the text as arguments to omc.
Useful if you e.g. want to disable compiling functions with gcc while you flatten code.
You can also set the environment variable RTEST_OMCFLAGS if you want to insert these flags for all commands you run.

```
setup_command: gcc ...
```

Will execute the provided command before running omc.

```
teardown_command: rm -f ...
```

Will execute the provided command after running omc.