User Guide

OSMO Tester

MBT tool

v3.1

Teemu Kanstrén

NOTE: You may wish to check the tutorials for a bit more concrete and practical introductions.

Table of Contents

[Introduction 3](#_Toc365571263)

[Modeling notation 4](#_Toc365571264)

[Test generation 11](#_Toc365571265)

[Special model elements 12](#_Toc365571266)

[Report builders 16](#_Toc365571267)

[Customization of algorithms and other components 16](#_Toc365571268)

[Calendar Tutorial 16](#_Toc365571269)

[Optimizing the Test Suite 25](#_Toc365571270)

[OSMO Explorer 27](#_Toc365571271)

[Conclusions 28](#_Toc365571272)

[References 28](#_Toc365571273)

# Introduction

OSMO Tester is a model-based testing (MBT) tool. The general idea is to have the tool generate test cases as opposed to writing test cases manually. Of course, the tool does not generate the test cases magically out of nothing. To generate test cases, the tool requires the user to provide it with a *model program*. The model program represents the behavior of the system under test (SUT) at a suitably high abstraction level. It can be seen as representing the possible combinations of test steps for the SUT, and the test generator then combines these steps in different ways.

To avoid generating invalid test cases, the model program needs to define when each test step is allowed. To define this, a set of “guard” statements are added to the model program, which are associated to specified test steps to define at which point the test step is allowed. The model program is practically a Java program with specific annotations, and executed in specific ways by the test generator. This means the model program can keep track of what steps have been executed and what state the SUT should be in as a result. This allows writing the guard statements as expressions over the model state variables, and test oracles to check various properties of the model program against the SUT at different points in time.

An example of this is shown in Figure 1 for a simple vending machine. In this example, there are two basic test steps. One is for inserting money, and the other is for vending (which produces the bottle). In order to be able to get a bottle, one first has to insert the required amount of money, which in this case is 100 cents or more, with a bottle cost of 100 cents. In this case, the potential test steps are “insert money” and “vend”. These take different parameters and are allowed at different times. Insert money takes the parameter of how much money (coin type) is inserted. It is allowed always. Vending takes no parameters but is only allowed when 100 cents or more have been inserted. Similarly, we could disallow the “insert money” step if there are no bottles left in the machine.



Figure 1. Vending-machine example.

Inserting money increases the variable that defines how many cents have been inserted. Vending reduces the inserted money value by 100 cents and produces a bottle as output, also reducing the number of available bottles by one. The concrete test cases generated would then invoke the SUT with these inputs and check for the expected outputs (bottles, changes in inserted cents, …).

# Modeling notation

This section describes the OSMO Tester modeling notation in terms of an example of the vending-machine described before and as commonly used in MBT literature (e.g., [1]). This vending machine accepts three types of coins (10 cents, 20 cents, 50 cents) and when a total of 100 cents has been inserted the “vend” action is enabled and can be invoked to produce a bottle. Figure 2 illustrates this as modeled with OSMO Tester.



Figure 2. Vending machine more detailed example.

This model is an elaboration of the one shown in Figure 1. The small circles are actions (test steps) that can be performed on the SUT in a state and they can also modify the model program state. The big circle in the middle shows the state in terms of the model program variables (amount of inserted coins (cents) and number of available bottles). These are modified by the actions/test steps that are the smaller circles on the bottom row.

As a simplified view, consider that the bottom row defines the possible test steps and the variables (state) define when the steps are allowed to be taken. The dotted lines show how each step affects the variables, *coins* and *bottles*. For example, inserting 10 cents increases the amount of coins inserted in by 10. The solid lines show a step being taken in a given state, with possible guards associated with it. There is only one guard shown in this model, saying that the *vend* step can only be taken when there are 100 or more coins available and one or more bottles available (bottles > 0).

In this example the state is rather simple. It has two variables, coins and bottles. These can be used to define specific coverage requirements such as having generated tests for vending with bottles available to vend, no bottles available to vend, and enough or not enough coins for vending can be covered by different test generation algorithms and configurations. Defining coverage requirements will be discussed in later sections.

Listing 1 illustrates the OSMO Tester notation for the vending-machine example.

public class VendingExample {

private final Scripter scripter;

private int coins = 0;

private int bottles = 10;

@TestSuiteField

private TestSuite testSuite = null;

public VendingExample() {

scripter = new Scripter(System.out);

}

@Guard

public boolean gotBottles() {

return bottles > 0;

}

@BeforeTest

public void start() {

coins = 0;

bottles = 10;

int tests = testSuite.getTestCases().size()+1;

System.out.println(“Starting test:”+tests);

}

@AfterSuite

public void done() {

int tests = testSuite.getTestCases().size()+1;

System.out.println(“Total tests generated:”+tests);

}

@TestStep("10cents")

public void insert10cents() {

scripter.step("INSERT 10");

coins += 10;

}

@TestStep("20cents")

public void insert20cents() {

scripter.step("INSERT 20");

coins += 20;

}

@TestStep("50cents")

public void insert50cents() {

scripter.step("INSERT 50");

coins += 50;

}

@Guard("vend")

public boolean allowVend() {

return coins >= 100;

}

@TestStep("vend")

public void vend() {

scripter.step("VEND ("+bottles+")");

coins -= 100;

bottles--;

}

@EndCondition

public boolean end() {

return bottles <= 0;

}

@Post

public void checkState() {

scripter.step("CHECK(bottles == "+bottles+")");

scripter.step("CHECK(coins == "+coins+")");

}

public static void main(String[] args) {

OSMOTester tester = new OSMOTester(new VendingExample());

tester.generate(55);

}

}

Listing 1. Example vending machine in OSMO notation.

Here we see a number of the core model annotations being used. The following annotations are supported:

* @Guard: Defines a check that needs to pass for an associated transition to be considered as enabled.
* @Transition: What is now @TestStep used to be called a transition. Same thing, different name.
* @TestStep: Defines a possible test step to take, and the logic to execute (generate) it.
* @Pre: Defines a method that is executed before associated test steps.
* @Post: Defines a method that is executed after associated test steps.
* @EndCondition: Defines a check that when it returns true it causes the current test generation to stop and the generation to continue with the next test in the suite.
* @BeforeTest: Called before a test case is generated.
* @AfterTest: Called after a test case generation has finished.
* @BeforeSuite: Called before any test cases in the test suite are generated.
* @AfterSuite: Called after all test cases in the test suite have been generated.
* @RequirementsField: Defines a field that holds a test requirements object.
* @TestSuiteField: Defines a field that is used to hold test generation history information.
* @Variable: Identifies an important variable value o capture for analysis purposes.
* @LastStep: A final step (transition) to be executed at the end of each test.
* @StateName: You can give more generic names for, and group several variable values as, states via methods tagged with this. Used in some visualizations and coverage calculations.
* @ExplorationEnabler: If online exploration is used, called before running a model program instance in exploration mode. Should disable side-effects.
* @GenerationEnabler: If online exploration is used, called before running a model program instance in generation mode. Should enable side-effects.

Another important part to note here is the setting of the “seed” values. This defines what values are generated for any randomization elements. Since the version 3.1 the seed is given as a parameter to the generator to allow parallel execution of optimizers with different seeds. This helps ensure the user understands if they are doing deterministic test generation or not. If you want non-deterministic generation, use a random source such as the system clock to generate the seed itself.



Figure 3. Generation flow with annotation elements.

Figure 3 shows a high-level overview of how the different annotations are processed by the test generation engine. Note that only the annotations matter, the naming of the methods themselves does not matter. However, some constraints on the method and field signature are as follows:

* @Guard: Method must return Boolean value. Method must not take any parameters.
* @Transition: Method must not take any parameters.
* @TestStep: Method must not take any parameters.
* @LastStep: Method must not take any parameters.
* @Transition: Method must not take any parameters.
* @BeforeTest: Method must not take any parameters.
* @AfterTest: Method must not take any parameters.
* @BeforeSuite: Method must not take any parameters.
* @AfterSuite: Method must not take any parameters.
* @Pre: Method must not take any parameters or take a single parameter of type Map<String, Object>.
* @Post: Method must not take any parameters or take a single parameter of type Map<String, Object>.
* @EndCondition: Method must return Boolean value. Method must not take any parameters.
* @RequirementsField: Field type must be Requirements. Value must be non-null.
* @TestSuiteField: Field type must be TestSuite. Value must be null (will be set by the generator).
* @Variable: Field can be of any type. If it implements the VariableValue interface, the value returned by the value() method is stored.
* @StateName: Method takes no parameters and returns a String.
* @ExplorationEnabler: Method takes no parameters. Return value is ignored.
* @GenerationEnabler: Method takes no parameters. Return value is ignored.

**@BeforeSuite**

During test generation, first any methods annotated with the @BeforeSuite annotation are executed on the model. These are executed only once during test generation before any tests are generated. Practically here, suite refers to all generated test cases and a test case refers to a set of test steps. This is useful for setting up stuff for the overall test generation process.

**@BeforeTest**

Before a test case is generated, all methods annotated with @BeforeTest are executed. New test case generation is considered to start at suite start, and when a test end condition returns true signaling the end of a test case, while at the same time the suite end condition returns false, signaling that more test cases should be generated. For example, in Listing 1 there is one method called start() with this annotation, which resets the model state between generated tests.

**@Guard**

When test generation is progressing, the test generation engine calls all @Guard annotated methods to identify the current set of enabled steps. Any guard method that returns false is considered to disable all associated steps, even if other guard methods for that step return true. If no guard is associated to a step, the step is considered to be enabled always. A guard is associated to a specific step based on their identifiers. An identifier is associated to a guard annotation as @Guard(“stepname”), where stepname is a String matching the name of a step to which it is associated. If @Guard is present with no name given, it is associated to all steps in the model. It is also possible to associate a single guard to several steps using the notation of @Guard({“name1”, “name2”}) where the associated steps are given as a list of strings. Every guard method is always executed to identify all enabled steps between test steps (transitions executed). Note that defining a guard named @Guard(“all”) equals defining it as @Guard, as this matches the unnamed @Guard annotation in the internal OSMO Tester representation. A special way to express a guard is by using ! in the beginning of the name it is associated to perform a negation of the name. For example @Guard(“!login”) would associate a guard to all test steps but the login step.

Besides direct association via the name of the test step, it is also possible to associate guards to groups of steps. In this case, the test steps must have been defined with the group name as explained below. The name given for the guard is matched to either a step name or a group name, whichever the generator finds the match for.

**@TestStep**

The actual test steps to be generated are represented by the @TestStep annotated methods in the model. The enabled steps are identified by their associated guard statements as described above. Steps are named similar to guards, i.e. @TestStep, @TestStep(“name”), @TestStep({“name1”, “name2”}). Similar to guards, step named @TestStep(“all”) are forbidden. The set of enabled steps are identified by the associated guard statements that return true at a given time. From this set of enabled steps the chosen test generation algorithm then picks one to be executed as the next test step.

It is also possible to give weights to steps, using the notation of @TestStep(name=”name”, weight=2). In this case, name is the name of the step and the weight is used with different weighted algorithms to define how many times the step should be taken in relation to other available steps. The default value for a weight is 10, so any step with undefined weight has a weight of 10. When multiple steps are enabled, a test generation algorithm that takes weights into account will favor the ones with more weight more often. For example, if step A has a weight of 1 and step B a weight of 2, and both are always enabled at the same time, step B will be taken twice as often as step A. Examples of a weighted algorithms are the WeightedRandomAlgorithm and WeightedBalancingAlgorithm implementations. Note that the choice is still based on several factors with only a higher probability given to the ones with higher weights (and with the balancing algorithm, the number of times the step has been taken also impacts the choice).

Since OSMO Tester 3.0 version, it is also possible to define a group name for a test step. This is done with a group fields such as @TestStep(name=”name”, group=”groupname”). Note that it is not allowed to have any step and group with the same name in the given model objects. You can have several steps belonging to the same group (that is the point to have groups) but, for example, it is not allowed to have @TestStep(name=”my\_name”, group=”my\_name”).

**Example for @Guard+@TestStep**

For example, in Listing 1, there is a test step and a guard named *vend*. The guard checks that this step can only be taken if there are 100 coins or more inserted. After this becomes true, the *vend* step is enabled. There is also a general guard statement in the form of the method gotBottles(). This makes sure that any step can only be taken (coins inserted, vending applied) when there are some bottles in the machine. Notice that in this case this is necessarily needed as there is an end condition in the model making sure this state is never achieved. However, it is included here to illustrate the concept. We could also add a new step called *return deposit* that would return all inserted coins and this could be enabled even when no bottles are present. In this case the association for the gotBottles() guard method would become @Guard(“!return deposit”). That is, it would not include the “return deposit” step, which would be the only step allowed when no bottles are left.

**@LastStep**

After a test case is destined to end, before @AfterTest is handled, all @LastStep annotated methods are called. Difference with the @AfterTest is that this one is executed as a part of the test case and thus failed assertions and other such exceptions will not stop whole test generation but fail the test case. Can be useful, for example, if generating a large input for one-shot execution as the result of the test model (e.g. transformation testing). In such a test case we might wish to run the test input and evaluate the test oracle once at the end, while the test steps would build the overall input over time.

**@Pre**

Before a test step is taken, all associated @Pre annotated methods are executed. These are named and associated to test steps similar to guard statements. If the method has the Map parameter, a single Map instance is passed for each pre-method. This is the same instance each time for the same test step invocation. Thus any values set in the pre-methods are accessible in all post-methods associated to that test step using the Map. This can be useful for writing a test oracle that compares the state before and after the test step.

**@Post**

After a test step is taken, all associated @Post annotated methods are executed. These are named and associated to test steps similar to guard statements. If the Map parameter is provided, it will be the same as was provided for any @Pre method for that step. Thus any values set in the pre-methods are accessible in all post-methods associated to that step. Any global variables in the model class tagged as @Variable will be stored into this Map as well (automatically by the generator).

**@EndCondition**

When all @Post-methods associated to the currently executed test step have been executed, the current test case is evaluated for stopping. At this point any method annotated with @EndCondition is executed. If any one of them return true, the current test generation is stopped. If no end condition returns true, the test generation end conditions as defined for the generator are executed to evaluate if the test case generation should be continued. If test generation is evaluated to continue, the test generation engine will re-iterate with evaluating all guards for enabled steps and continue again from there, picking another step and so on. In Listing 1 there is one end condition that makes sure test generation is ended when there are no bottles present as otherwise the generator would throw an exception as there would be no enabled step left in this state.

**@LastStep**

When test generation for a test case is finished but before @AfterTest is called, the methods tagged with @LastStep are executed. These are intended to do final checks and executions, such as running a transformation on the generated input and checking the results. The reason one might want to use this annotation instead of @AfterTest is that this is executed as part of the test generation flow, and any failures are reported as test failures (marking the test failed), not as errors in test generation (which stop the whole process). There is no guarantee in the order multiple methods annotated with this are executed in.

**@AfterTest**

When test generation for a single test case is finished, all methods annotated with @AfterTest are executed. Once this has been executed, the generator test suite end condition is evaluated to define whether to continue with generating the next test case or to stop all test generation. If test generation is continued, the generator continues with the next test case and the @BeforeTest annotated methods.

**@AfterSuite**

Once all test generation is finished (test suite end condition tells the engine to stop), all @AfterSuite annotated methods are executed.

**@ExplorationEnabler and @GenerationEnabler**

The exploration algorithm creates new model program instances and calls these methods to initialize each instance into the exploration or concrete test generation/execution mode before the instance is executed.

# Test generation

As described, the purpose of OSMO Tester is to generate test cases from the given model program based on the annotations described above. Note that the specific order of executing several available guards or other annotated methods is not explicitly defined. It should be deterministic (at least when using a specific generator version) but beyond that you should make no assumptions that they would be executed in specific order (such as the ordering in your text editor for the source code from top to bottom).

Besides the annotations, it is also possible to configure OSMO Tester using a set of configuration methods. This includes defining a set of the test suite end conditions, a set of test case end conditions, a test generation algorithm, and enabling or disabling debug logging. As noted before, it is also possible to add specific elements to the model that are relevant for the different test generation algorithms, such as giving weights to test steps and defining requirements to be covered in the model.

In Listing 1 the test generation is initiated with the following the following fragment:

public static void main(String[] args) {

OSMOTester tester = new OSMOTester(new VendingExample());

tester.generate(100);

}

However, we can also define a set of additional attributes such as

public static void main(String[] args) {

1 OSMOTester osmo = new OSMOTester();

2 osmo.setDebug(true);

3 osmo.addModelObject(new VendingExample());

4 osmo.addTestEndCondition(new Length(3));

5 osmo.addSuiteCondition(new Length(2));

6 osmo.setAlgorithm(new BalancingAlgorithm());

7 osmo.generate(100);

}

}

The lines here are the following:

1. Creates the OSMO Tester test generation engine.
2. This enables more verbose debug printing in System.out and in a log file. Best done right after creating the generator.
3. Adds a new model object to be parsed for generation. You can add as many as you like, and they will be combined together where transitions, oracles, guards, etc. are matched across the provided objects. Some people call the result a “flattened” model because they are treated as if written inside the same class.
4. Sets the end condition object for ending generation of single test cases. The condition used here causes each generated test case to have 3 steps (that is, after executing three transitions, the generator stops). The default condition used, if no end condition is by the user, is to end after generating a minimum of 1 test step, with 5% probability after each test case.
5. Sets the end condition object for ending generation of all tests (the test suite). The condition used here causes the generator to generate 2 tests. The default condition is set to end after generating a minimum of 1 test cases, with 10% probability at each point.
6. Sets the test generation algorithm. By default this is set to a RandomAlgorithm that randomly takes one of the available steps. The BalancingAlgorithm used here takes previously uncovered steps if available, and balances further choices by always taking one of the available steps and step-pairs with the least coverage so far.
7. This invokes the test generation engine to generate tests from the given model objects with the defined configuration.

Note that you must always specify the randomization seed before generation as shown in these listings. This is to make it explicit that if you do not specify the same one every time, different runs will produce different results. Most of the times this is not the desired behavior as it makes repeating results difficult. You can still randomize the seed yourself if so desired, for example, using system clock.

# Special model elements

As noted before, there are some special model elements that only make sense when combined with specific test generation algorithms. These include weighted transitions, requirements and state variable definitions.

**Weighted Transitions**

For example, we could emphasize the test step for 10 cents by changing it in the following way:

@TestStep(name="10cents", weight=20)

public void insert10cents() {

scripter.step("INSERT 10");

coins += 10;

}

With this change, the 10 cents step would now be taken twice as often where the others are taken once (assuming the step is enabled). The default weight if none is specified is 10. To make the step useful, we would also need to use a matching weighted test generation algorithm:

osmo.setAlgorithm(new WeightedRandomAlgorithm());

**Requirements**

We could also use requirement tags in the model to ensure that one of the generated test cases will cover a case where all bottles have been emptied in the vending machine:

private int bottles = 10;

@RequirementsField

private Requirements req = new Requirements();

...

public VendingExample() {

scripter = new Scripter(System.out);

req.add(“all bottles vended”);

}

...

@TestStep("vend")

public void vend() {

scripter.step("VEND ("+bottles+")");

coins -= 100;

bottles--;

if (bottles == 0) {

req.covered(“all bottles vended”);

}

}

In this case, we add coverage requirements that should be covered in the constructor with the add() method. We mark them covered in test generation with the covered() method. The given parameters (requirement name) must match in order for the requirements to be considered as covered. New requirements for coverage can be added at any time before the generation is started with the osmo.generate() method. In most cases, they can even be added during generation but this is not guaranteed to work.

In order for test generation to continue until this requirement is covered, we can apply a matching test generation algorithm:

CoverageRequirement req = new CoverageRequirement(0, 0, 1);

osmo.addSuiteEndCondition(new CoverageEndCondition(req));

With this, the test suite generation will go on until the requirement is covered. Note that this can mean it going on forever if the combinations are incorrect (such as length of 3 for a test case, which will never reach this state). In practice, such goals are best achieved with customized model configuration, modularized models, and a diverse set of simple model objects.

**Composition of End Conditions**

In addition to specifying specific end conditions, it is also possible to compose several end conditions into one. This can be done using the And and Or end condition classes. Alone these have no meaning but together with others can combine several end conditions into one. For example, the following creates an end condition that specifies that test cases need to have a minimum length of 5 steps (transitions taken in the model), maximum of ten steps (transitions) and the probability of 25% for having a length of 6-9 transitions taken (ending before max is reached):

EndCondition min = new Length(5);

EndCondition max = new Or(new Length(10), new Probability(0.25d));

EndCondition minMax = new And(min, max);

osmo.addTestEndCondition(minMax);

To put it different, the test must have a minimum of 5 steps AND (maximum of 10 steps OR randomized probability of 25% becomes true in any step after 5). OK, practically the probability to end is 25% at each step and not overall in all the 5 steps (it is actually higher since each step adds to the probability) but anyway...

**Modelling Data**

The items described above mainly relate to defining different aspects related to the control-flow of the model (the test sequence generation). However, it is also important to note the need to generate data for parameter values of the test sequences. Support for this is provided by elements in the osmo.tester.model.data package. Each component in this package can generate values fitting the data values it represents, as well as to check if the given value is part of the given set. These are performed in each case by the next() and evaluate(Object) methods of each class. They also share common Input and Output interface implementations, allowing each to be replaced by other when references through the interfaces. These objects include the following elements:

* Boundary: Data at the boundaries of given range.
* CharSet: Set of defined characters, such as ASCII, valid XML content, and others.
* Text: Sequences of CharSet.
* ValueSet: Set of values containing any Java objects.
* ValueRange: Range of numerical values with a minimum and maximum.
* ValueRangeSet: A set of numerical ranges, possibly overlapping or not.

These objects are based on general test automation concepts (such as category-partitioning). You can also provide your own as you see best fit by implementing the Input and Output interfaces. If you want the OSMO Tester to make better use of them in optimization and test tracing, you can also extend the SearchableInput class in your own implementations, which will cause OSMO Tester to capture generated values in the test steps.

**BaseModel Class**

The code base also includes a BaseModel class, which can be extended to provide a test model. This gives access to the current test case being generated, the previous step taken, and the test case identifier.

**State Variable Identification**

The @Variable annotation can be used to identify specific state variables in the model. If any model variables are described with this annotation, they will be taken by the generator to describe the model state at different points of time. The generator will not act on them but will store each variable value to every TestStep object stored in the TestSuite object (accessible in the model with @TestSuiteField annotation and directly in the test generation algorithms). Each TestStep object will then contain values for all these variables both before the transition that created the test step was taken and after the transition was taken. These values can be accessed using corresponding getter methods in the TestStep objects.

Similar to the values being stored to the TestStep object, they are also provided to all the @Pre and @Post methods if they are defined to have the Map<String, Object> parameters. In this case, the @Pre methods will get the values of the variables before the associated transition is taken and the @Post methods will get the values after the transition has been taken. The key in the map will be the variable name and the value will be the value of the object stored in the field. In case of primitives or Objects not implementing the VariableValue interface, the value stored is the raw value of the field. In case the field contains an Object implementing the VariableValue interface, the value() method of the interface will be called to provide a value to be stored.

The @Variable annotation is intended to be used for tagging primitive (as is) and custom values (with the VariableValue interface) in the model. It captures the variable name either from as the name defined in the source code for the variable or as a parameter to the annotation. The dataflow modeling objects provided with the OSMO Tester library are of type SearchableInput, which means the generator will actually capture all values they produce when their next() method is invoked even when not annotated as @Variable. In fact, annotating them so has little useful effect, except storing the last value they generated before and after a transition.

**JUnit integration**

It is also possible to run tests using JUnit integration. Check the JUnit online testing with OSMO Tester tutorial for more information.

# Report builders

There are some components in the osmo.tester.reporting package that can generate various types of reports for you. These include coverage and test data reporters in HTML and CSV format. HTML reports can be handy for browser based sharing of results, and CSV reports for Excel based sharing. These reporters work in different ways, either studying the information stored in the test suite object or by giving you and interface to store data while the tests are executed.

Coverage reports are based on coverage data stored in the TestSuite object. Here is an example:

CSV csv = new CSV(test, testCoverage, fsm);

String report = csv.getTransitionCounts();

This will now have created the report that is stored in the “report” variable. You can store this on disk, publish it someplace, or do anything you wish as it is just a piece of text. There is a more detailed example in the OSMO Tester examples package.

# Customization of algorithms and other components

In addition to the existing test generation algorithms, suite end conditions and other components provided with OSMO Tester, customized versions of these components can also be created.

To create a test generation algorithm that traverses the model object, you must implement the FSMTraversalAlgorithm interface. To define your own test suite or test case generation end conditions, you must implement the EndCondition interface. More information on these can be found in their Javadocs and in the project source code.

# Calendar Tutorial

The OSMO Tester source code includes several smaller examples and one bigger example. The bigger example is an example for generating tests for a calendar application. The calendar used as a test subject is a simple example of a calendar component implemented in Java and also included in the OSMO Tester calendar example source code. The example can be found under the examples source code in package osmo.tester.examples.calendar.

This has several subdirectories/packages, which are the following

* osmo.tester.examples.calendar.scripter. Examples of online and offline scripters for the calendar example.
* osmo.tester.examples.calendar.testapp. The test subject in this example.
* osmo.tester.examples.calendar.testmodel. The test models for this example.

The tested application provides the following calendar functionality and constraints

* A calendar can be created, with a user associated to it.
* One user can have one calendar associated.
* Tasks can be added to the calendar, a task has a description and a date.
* Meetings can be added to the calendar, a meeting has a description, a location, a start date, and an end date.
* Existing meetings can be linked to new users, making the added users participants in the meeting, while the user for whom it was originally created is the organizer of the meeting.
* Tasks can be deleted.
* Meetings can be deleted. If the meeting is deleted from a participant, it has no effect on other users and their calendars. If the meeting is deleted from an organizer, the same meeting is also deleted from all linked participants.
* Task and meeting descriptions and locations are String data types.
* Task and meeting related times are described using Java Date data type.

In order to demonstrate the different OSMO Tester features, the following model elements are provided:

* adding and removing meetings for organizer.
* adding and removing tasks.
* creating meetings that overlap other meetings and tasks that overlap meetings.
* attaching meetings to an organizers calendar, creating participants for the meetings.
* error handling of invalid input.
* test oracles to assert correct tasks and meetings are present in the calendar.

**Base model for creating meetings**

The base model for the calendar includes the creation and removal of meetings. It looks like this:

public class CalendarMeetingModel {

/\*\* The global model state, shared across test models. \*/

private final ModelState state;

/\*\* The scripter for creating/executing the test cases. \*/

private final CalendarScripter scripter;

private final PrintStream out;

private final Tags reqs = new Tags();

public CalendarMeetingModel(ModelState state, CalendarScripter scripter) {

this.state = state;

this.scripter = scripter;

this.out = System.out;

}

public CalendarMeetingModel(ModelState state, CalendarScripter scripter,

PrintStream out) {

this.state = state;

this.scripter = scripter;

this.out = out;

}

@BeforeTest

public void setup() {

state.reset();

scripter.reset();

out.println("-NEW TEST");

}

@AfterSuite

public void dump() {

scripter.write();

}

@TestStep("Add Meeting")

public void addEvent() {

String uid = state.randomUID();

Date start = state.randomStartTime();

Date end = state.randomEndTime(start);

ModelEvent event = state.createEvent(uid, start, end);

out.println("--ADDMEETING:" + event);

scripter.addEvent(event);

}

@Guard("Remove Meeting")

public boolean guardRemoveOrganizerEvent() {

return state.hasEvents();

}

@TestStep("Remove Meeting")

public void removeOrganizerEvent() {

ModelEvent event = state.getAndRemoveOwnerEvent();

out.println("--REMOVEMEETING:" + event);

scripter.removeEvent(event.getUid(), event.getEventId());

}

}

Here we see a number of basic elements for the model as expressed in the OSMO Tester annotations:

@BeforeTest: This method resets the model state between generated tests, and also resets the scripter to initiate the creation of a new test case.

@TestStep(“Add Meeting”): This method creates a new meeting with a set of generated data, and scripts the its creation with the help of the defined scripter. This transition has no guard associated, meaning that it can be taken by the generator at any time.

@Guard(“Remove Meeting”): This method ensures that a step to remove a meeting is only generated when meetings exist (have been previously added).

@TestStep(“Remove Meeting”): This method adds a test step that removes a meeting. More precisely, it removes a meeting from the organizer. This is because in other partial models, we add an option to link already created meetings to new users through their calendars. Thus, this will remove the meeting from the organizers calendar. In practice, if the meeting also has participants, the meeting will also be deleted from their calendars since this simulates the full cancellation of the meeting.

The two variables of the model shown here are:

ModelState: This incorporates the state of the test model. The elements of the model state could also be directly embedded into this single class file but here we prefer to encapsulate it in its own object to provide for more cohesive structure and also to be able to share the state across several partial models.

CalendarScripter: This defines an interface for scripting the results. There are two different implementations of a scripter in this example. One is an online scripter, another one is an offline scripter. The role of these scripters is to relay the test step commands to the test framework and through there to the test target.

Now we are ready to run the generator to generate test cases from the model. Here is how we can do this:

/\*\*

\* This is used to execute the calendar example.

\*

\* @param args command line arguments, ignored.

\*/

public static void main(String[] args) {

OSMOTester osmo = new OSMOTester();

ModelState state = new ModelState();

CalendarScripter scripter = new OnlineScripter();

osmo.addModelObject(new CalendarMeetingModel(state, scripter));

osmo.generate(111);

scripter.write();

}

Note that the last part of scripter.write() is only needed for offline scripting. In any case, this should execute the model with OSMO Tester using the online scripter. As a result the calendar application is created in memory and all input data is provided and output data checked. The sequences should be visible on the system console.

**Scripter**

You may wonder what happens in the scripter to make the magic of actually executing the test cases happen. You can find the full source code for this class in the examples package. Here is the listing from the OnlineScripter for the relevant parts:

public class OnlineScripter implements CalendarScripter {

/\*\* Maps user identifiers to CalendarUser objects. \*/

private Map<String, CalendarUser> users = new HashMap<String, CalendarUser>();

public OnlineScripter() {

}

/\*\*

\* Used to reset the scripter between generated tests.

\*/

@Override

public void reset() {

users.clear();

}

/\*\*

\* Helper to get the calendar object for the given user.

\*

\* @param uid User whose calendar we want.

\* @return The calendar.

\*/

private CalendarApplication getCalendarFor(String uid) {

CalendarUser user = users.get(uid);

if (user == null) {

user = new CalendarUser();

users.put(uid, user);

}

return user.getCalendar();

}

@Override

public void addEvent(ModelEvent event) {

CalendarApplication calendar = getCalendarFor(event.getUid());

CalendarEvent calendarEvent = calendar.addEvent(event.getStart(),

event.getEnd(), event.getDescription(), event.getLocation());

event.setEventId(calendarEvent.getId());

}

@Override

public void removeEvent(String uid, ModelEvent event) {

CalendarApplication calendar = getCalendarFor(uid);

calendar.removeEvent(event.getEventId(), false);

}

This is an online scripter, meaning that it will practically create the test calendars for requested users, create meetings (events in the code) for these calendar objects and so on as needed.

For the remainder of this tutorial we will describe the online scripter, but for offline scripting, there is also an example in the osmo.tester.examples.calendar.scripter.offline package for scripting the calendar example through the Robot Framework. This is based on the generic Robot Framework scripter available in the osmo.tester.scripter.robotframework package. Running with the offline scripter produces the script in HTML format into the directory where started with the given filename.

**Model state object**

The model state is stored in the ModelState object. Here is the relevant part for the meeting creation:

public class ModelState {

/\*\* Test users with calendars. \*/

private Collection<String> uids = new ArrayList<>();

/\*\* Events for each user. \*/

private Map<String, Collection<ModelEvent>> userEvents = new HashMap<<>>();

/\*\* Used to generate unique identifiers for events. \*/

private AtomicInteger eventCount = new AtomicInteger(0);

/\*\* Used to generate start times between January 2000 and December 2010. \*/

private ValueRange<Long> startTime;

public ModelState() {

}

/\*\*

\* Used to reset the state between test generation.

\*/

public void reset() {

uids.clear();

userEvents.clear();

eventCount = new AtomicInteger(0);

int users = cInt(1, 5);

Text names = new Text(4,7);

names.setName("name");

for (int i = 1; i <= users; i++) {

this.users.add(names.next());

}

Calendar start = Calendar.getInstance();

start.set(2000, 0, 1);

Calendar end = Calendar.getInstance();

end.set(2010, 11, 31);

startTime = new ValueRange<Long>(start.getTimeInMillis(), end.getTimeInMillis());

}

public String randomUID() {

return oneOf(uids);

}

public ModelEvent createEvent(String uid, Date start, Date end) {

int count = eventCount.incrementAndGet();

String description = "event" + count;

String location = "location" + count;

Collection<ModelEvent> events = getOrCreateEvents(uid);

ModelEvent event =

new ModelEvent(uid, start, end, description, location);

events.add(event);

return event;

}

private Collection<ModelEvent> getOrCreateEvents(String uid) {

Collection<ModelEvent> events = userEvents.get(uid);

if (events == null) {

events = new HashSet<ModelEvent>();

userEvents.put(uid, events);

}

return events;

}

public boolean hasEvents() {

return userEvents.size() > 0;

}

public ModelEvent getAndRemoveOrganizerEvent() {

Collection<ModelEvent> uniqueEvents = getUniqueEvents();

ModelEvent event = oneOf(uniqueEvents);

Collection<String> toRemove = new HashSet<String>();

for (String uid : userEvents.keySet()) {

Collection<ModelEvent> events = userEvents.get(uid);

events.remove(event);

if (events.size() == 0) {

toRemove.add(uid);

}

}

for (String uid : toRemove) {

userEvents.remove(uid);

}

return event;

}

Here we see how the state is reset between test runs, how input is given in terms of picking a valid user identifier for test generation, how meetings are created for testing, and how meetings are removed. Notice that when actions are performed that modify the state on the test target, also the model state needs to be modified similarly. For this reason, the removal of a meeting needs to remove the meeting from the state but also to return it so that the scripter can be used to remove the same meeting from the actual test target. This keeps the model state in a consistent state where it can be assumed to match the test target at all times.

This example also illustrates the use of one of the input data model objects provided with OSMO Tester. This is the ValueRange object. As time in computers is typically expressed in terms of milliseconds from a given data as a long valued integer, this value range has minimum value as the number of milliseconds to create the date 1st of January, 2000. As the upper range it has the number of milliseconds to create the date 31st of December, 2010. Depending on the configured data generation strategy for this model object, the .next() method can then be used to create new values in this range, effectively providing us with dates between January 1st, 2000 and December 31st, 2010. Similar data model objects can be found in the OSMO Tester in the osmo.tester.model.dataflow package.

**Adding test oracles**

A following question is, what are the models good for, as currently they do not check anything. A test case is not very useful if it cannot make a verdict about correctness of what happened. We can add a new partial model for test oracles to provide an example:

public class CalendarOracleModel {

/\*\* The global model state, shared across test models. \*/

private final ModelState state;

/\*\* The scripter for creating/executing the test cases. \*/

private final CalendarScripter scripter;

public CalendarOracleModel(ModelState state, CalendarScripter scripter) {

this.state = state;

this.scripter = scripter;

}

@Post

public void genericOracle() {

for (String uid : state.getUids()) {

Collection<ModelEvent> events = state.getEventsFor(uid);

scripter.assertUserEvents(uid, events);

}

}

}

Now, what have here is the same model state and the same scripter. There is only one annotation here:

@Post: This method now loops through the calendars for all users, checking that the meetings for those users as stored in the actual calendar object match those generated in the model. Since the @Post annotation has no name defined it equals to @Post(“all”), meaning that it will be executed after each @TestStep tagged methods. This way the generated tests will keep checking the results all the time, after each test step.

The matching part of the online scripter is:

@Override

public void assertUserEvents(String uid, Collection<ModelEvent> events) {

CalendarApplication calendar = getCalendarFor(uid);

Collection<CalendarEvent> calendarEvents = calendar.getEvents();

if (events == null) {

if (calendarEvents != null && calendarEvents.size() > 0) {

fail("Events are null in model, should be empty also in calendar");

}

return;

}

assertEquals("Number of events in model vs calendar", events.size(),

calendarEvents.size());

for (ModelEvent modelEvent : events) {

String description = modelEvent.getDescription();

Date start = modelEvent.getStart();

Date end = modelEvent.getEnd();

boolean found = false;

for (CalendarEvent calendarEvent : calendarEvents) {

if (!calendarEvent.getDescription().equals(description)) {

continue;

}

if (!calendarEvent.getStart().equals(start)) {

continue;

}

if (!calendarEvent.getEnd().equals(end)) {

continue;

}

found = true;

}

assertTrue("ModelEvent not found on calendar:"+modelEvent, found);

}

}

Here we use the JUnit test framework to provide us with some online assertion capabilities, illustrating how any Java libraries can be used with the models. The assertEquals() and assertTrue() methods come from the JUnit framework and evaluate that the given reference value matches the actual observed value. Similar assertions can be built for any values observed or expected to be observed at any point in the model code. This includes any return values, any accessible global state value or any other values available. In the end it is just Java code with special features added by the OSMO Tester (kind of like a framework).

To execute OSMO Tester with the new partial model added, we do the following:

/\*\*

\* This is used to execute the calendar example.

\*

\* @param args command line arguments, ignored.

\*/

public static void main(String[] args) {

OSMOTester osmo = new OSMOTester();

ModelState state = new ModelState();

CalendarScripter scripter = new OnlineScripter();

osmo.addModelObject(new CalendarMeetingModel(state, scripter));

osmo.addModelObject(new CalendarOracleModel(state, scripter));

osmo.generate(444);

scripter.write();

}

**Testing error handling behavior**

The above examples test the expected correct behavior of the system. In addition, it is commonly of interest to use to test that the error handling of the test target works as expected. Here is an example of a partial model for testing the error handling functionality of the calendar example:

public class CalendarErrorModel {

/\*\* The global model state, shared across test models. \*/

private final ModelState state;

/\*\* The scripter for creating/executing the test cases. \*/

private final CalendarScripter scripter;

public CalendarErrorModel(ModelState state, CalendarScripter scripter) {

this.state = state;

this.scripter = scripter;

}

@Transition("Remove Meeting That Does Not Exist")

public void removeMeetingThatDoesNotExist() {

System.out.println("--REMOVEMEETINGHATDOESNOTEXIST:");

scripter.removeMeetingThatDoesNotExist(state.randomUID());

}

}

Here we have again the shared model state and scripter. Additionally, we have the transition:

@TestStep(“Remove Meeting That Does Not Exist”): This tries to remove an event with an invalid identifier but with a valid user id. The corresponding code from the online scripter is:

@Override

public void removeMeetingThatDoesNotExist(String uid) {

CalendarApplication calendar = getCalendarFor(uid);

try {

calendar.removeMeeting("no such meeting", false);

fail("Removing a meeting that does not exist should fail.");

} catch (Exception e) {

//expected

}

}

Notice that here the event id given is “no such meeting” and since the meeting identifiers are always of form “meeting”+number, we know this will fail. Thus we expect to get an Exception, and fail the test if one is not observed. We could further assert that the exception includes the correct error message and other details as necessary.

Again, to execute OSMO Tester with the oracle and error handling partial models added, we do the following:

/\*\*

\* This is used to execute the calendar example.

\*

\* @param args command line arguments, ignored.

\*/

public static void main(String[] args) {

OSMOTester osmo = new OSMOTester();

ModelState state = new ModelState();

CalendarScripter scripter = new OnlineScripter();

osmo.addModelObject(new CalendarMeetingModel(state, scripter));

osmo.addModelObject(new CalendarOracleModel(state, scripter));

osmo.addModelObject(new CalendarErrorModel(state, scripter));

osmo.generate(764);

scripter.write();

}

**Adding tasks, event participants, and more**

In order to add support for tasks, meeting participants, and other elements, we create similar new partial models. Full source code for these can be found with the calendar model source code in the OSMO Tester package.

# Optimizing the Test Suite

Since version 3.0, a “guided” test generation approach is also supported. In this case, the tool generates a set of test cases as guided by a set of user defined coverage criteria. These criteria can cover both structural aspects of the model as well as values over the internal state of the model.

The way the offline optimizer works is by

1. generating a number of test cases (by default 1000),
2. evaluating them according to user defined coverage criteria,
3. merging these with any existing tests in the suite
4. choosing a subset that best fulfills these criteria
5. running another iteration from step 1
6. repeating 1-5 until a new iteration has not produced any added coverage over a threshold score

Once this is done, a test set is provided as a result that in this set of generated test cases provided the best results.

To define your coverage criteria of interest, you must create a ScoreConfiguration object. This allows you to define weights for different coverage criteria:

* Length: Each test step gives N points of coverage score. Can be used to prefer longer (positive weight), neutral (zero weight) or shorter (negative weight) test cases.
* Variable count: Covering new unique variables for the first time scores N points.
* Variable values: Unique values for specific variable give N points of score.
* Step-pairs: Covering new pairs of test steps gives N points. A pair means two steps occurring in a sequence inside a test case.
* Steps: Covering a new unique step gives N points.
* Requirements: Covering a new requirement gives N points. Requirements are the elements tagged with Requirements.covered() method.
* Custom states: Covering a new value for the @StateName annotation gives N points.
* Ranges: You can split your numerical model variables to partitions and score the coverage of those ranges separately. For example, covering a value between 1-10 could score N points for that range, while covering a value for 11-100 would score another N points for that range and so on. These are separate from the ValueRange objects, the type does not matter as long as it produces integers. Currently only integers are supported.
* Variable combinations: For a unique combination of given variable values inside a test step, the test scores N points. For example, if we have variable “teemu” and “muumi”, if they have values teemu=1 and muumi=2 in step1 and teemu=2, muumi=5 in step2, then the first step will cover the value 1&2, and the second the value 2&5.

Notice the all the combinations are calculated at the suite level for the optimizer. Thus, if one test case/step already covers a specific criteria, following test cases/steps will score nothing for it. For the state/variables that is, if we talk about structural values such as steps, pairs, and length that is another matter.

/\*\* An example of running the greedy optimizer. \*/

public static void main(String[] args) {

ScoreConfiguration sc = new ScoreConfiguration();

//define neutral length, high requirements score and user custom state score

sc.setLengthWeight(0);

sc.setRequirementWeight(100);

sc.setRequirementWeight(50);

GreedyOptimizer optimizer = new GreedyOptimizer(sc,

new LengthProbability(1, 10, 0.1d), 234);

optimizer.addModelClass(CalculatorModel.class);

//stop if a new test case cannot get minimum of 50 score

optimizer.setThreshold(50);

List<TestCase> tests = optimizer.search();

//..now do what you wish with your tests

}

Now we get the results as a list of test cases. This also means that as new test cases are generated and thrown away if they do not make it to the final set, some considerations need to apply. If we just write the test script to disk while generating each test case, we will end up with a large set of scripts and difficulty in picking the scripts that match the chosen test cases. The simplest way to address this is to generate the test script and store it using the TestSuite.getCurrentTest().setAttribute(String) method.

The simplest way to get access to these is to use the BaseModel class which comes with the current test pre-set. You can then iterate the given test set and write the test scripts to external storage. Alternatively, you can of course build any solution you prefer.

Finally, the GreedyOptimizer implementation runs the algorithm described above using a single core on the system. If you want to make better use of your multi-core system, another implementation exists in the MultiGreedy optimizer. This one takes the same configuration as the single-core version and uses it to instanciate a given number of single-core instances to run in parallel. Finally, it merges all of their results to provide a single optimized set.

# OSMO Explorer

Since version 3.1, the OSMO Explorer extension is also included with the distribution. This is a special test generator algorithm implementation that runs several versions of the model program concurrently, each exploring different possible paths forward and picking one of the best scoring to take next. Exploration takes time, so in our experiments this has worked best with depths of 1 and 2 when generating test cases for systems where executing the step concretely takes some time (e.g., GUI testing).

More detailed documentation and tutorials may be added later but the basic form of use is to first initialize a coverage score configuration (as discussed above for guided generation). Once this is defined, use it to initialize the explorer itself and run test generation:

public static void main(String[] args) {

OSMOExplorer osmo = new OSMOExplorer();

ExplorationConfiguration config = createConfiguration();

osmo.explore(config);

}

private static ExplorationConfiguration createConfiguration() {

PaperModel1Factory factory = new PaperModel1Factory(System.out);

ExplorationConfiguration config = new ExplorationConfiguration(factory, 5, 55);

config.setFallbackProbability(1d);

config.setMinTestLength(10);

config.setLengthWeight(0);

config.setRequirementWeight(0);

config.setDefaultValueWeight(1);

config.setStepWeight(0);

config.setStepPairWeight(10);

config.setVariableCountWeight(0);

config.setMinTestScore(60);

config.setMinSuiteLength(5);

config.setTestPlateauThreshold(1);

return config;

}

This (above) is an example from the OSMO Tester core repository test cases (check there for more details if interested).

# Conclusions

OSMO Tester provides means to create test models and to generate test cases from these models. This document covered the basic concepts and notation of the tool. For more details, the reader is encouraged to check the OSMO Tester Javadocs. Tutorials for specific features are also available on the project website. There are also more examples in the OSMO Tester source code under the osmo.tester.examples package and in the source code test directory in form of JUnit tests.

Contact information can be found on the project web site for questions.

# References

OSMO Tester home page, discussion forums & source code: <http://code.google.com/p/osmo/>