User Guide

OSMOTester

MBT tool

Table of Contents

[Introduction 3](#_Toc300060544)

[Modeling notation 3](#_Toc300060545)

[Test generation 9](#_Toc300060546)

[Special model elements 10](#_Toc300060547)

[Test optimization 12](#_Toc300060548)

[Customization of algorithms and other components 13](#_Toc300060549)

[Conclusions 13](#_Toc300060550)

[References 13](#_Toc300060551)

# Introduction

OSMOTester is a model-based testing (MBT) tool. It uses a state-machine notation to describe the system under test (SUT) from the testing perspective. A test model of the expected system behavior is provided by the user and used by the tool as a basis for automatically generating test cases for the SUT.

MBT is about modeling the behavior of a system in terms of states and transitions, and using a test generation tool to generate tests from this model instead of writing separate models by hand. In the simplest form, each transition can be seen as a test step, and the tool generates test cases by forming varying compositions of transitions. The state of the model then defines when a test step (transition) is allowed to occur, practically providing a “correct” ordering of available test steps. Of course, more advanced applications are possible and encouraged where useful.

# Modeling notation

This section describes the OSMOTester modeling notation in terms of an example of a vending-machine borrowed from [[1](#Utting2007)]. 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 can be activated to produce a bottle. Figure 1 illustrates this as a state-machine.



Figure 1. Vending machine as a state-machine.

Note that this state-machine description is different from what one would consider a standard “state-machine”. In this case the big circle in the middle shows the actual state of the system. This is modified by the transitions 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 state defines when they are allowed to be taken. The dotted lines show how each transition affects the global state that is composed of two variables, *coins* and *bottles*. The solid lines show a transition being taken from a given state, with possible guard statements shown with it. There is only one guard statement shown in this model, saying that the *vend* transition can only be taken when there are 100 or more coins available and one or more bottles available.

Thus this state-machine expression puts more focus on transitions than on state itself. In fact, in this representation the state itself is rather simple. It is a composition of two variables, coins and bottles. Other approaches include defining a specific set of state, such as having inserted some coins or having each number of bottles represented by a specific enumeration. The OSMOTester notation emphasizes transitions as, as noted, in MBT the transitions typically represent test steps while leaving the state representation open for the user. In this example, specific state conditions for test coverage such as having generated enough tests for covering requirements for vending with enough bottles, no bottles and enough or not enough coins can be covered by different test generation algorithms. Notations for defining requirements to be covered in test generation also exist and will be discussed later in this document.

The test models for OSMOTester are in practice executable programs written in the Java programming language. The specific model elements for the tool are identified based on a set of specific annotations. Listing 1 illustrates the OSMOTester 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;

//uncomment this for failure to continue with 0 available transitions

bottles = 10;

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

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

}

@AfterSuite

public void done() {

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

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

}

@Transition("10cents")

public void insert10cents() {

scripter.step("INSERT 10");

coins += 10;

}

@Transition("20cents")

public void insert20cents() {

scripter.step("INSERT 20");

coins += 20;

}

@Transition("50cents")

public void insert50cents() {

scripter.step("INSERT 50");

coins += 50;

}

@Guard("vend")

public boolean allowVend() {

return coins >= 100;

}

@Transition("vend")

public void vend() {

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

coins -= 100;

bottles--;

}

@EndState

public boolean end() {

return bottles <= 0;

}

@Post

public void checkState() {

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

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

assertTrue(coins <= 100);

assertTrue(coins >= 0);

assertTrue(bottles >= 0);

}

public static void main(String[] args) {

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

// tester.setDebug(true);

tester.generate();

}

}

Listing 1. Example vending machine in OSMO notation.

Here we see a number of the core OSMOTester model annotations being used. OSMOTester supports the following annotations:

* @Guard: Defines a check that needs to pass for an associated transition to be considered as enabled.
* @Transition: Defines a transition that can be taken by the test generation algorithm to execute a test step.
* @Pre: Defines a method that is executed before associated transitions.
* @Post: Defines a method that is executed after associated transitions.
* @EndState: 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.
* @EndCondition: Defines a check that needs to return true to allow for test generation to stop. Once a test generation algorithm has signaled test generation to stop, it continues until this returns true.
* @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.



Figure 2. Generation flow with annotation elements.

Figure 2 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 signature are as follows:

* @Guard: Method must return Boolean value. 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.
* @EndState: 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.
* @Variable: Field can be of any type. If it implements the VariableValue interface, the value returned by the value() method is stored.

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 tests and a test case refers to a set of test steps as separated by the given test generation algorithms to separate generated tests (with @BeforeTest and @AfterTest executed in between).

Before any 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 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. In this example, this resets the model state between generated tests.

When test generation is progressing, the test generation engine calls all @Guard annotated methods to identify the current set of enabled transitions. Any guard method that returns false is considered to disable all associated transitions. If no guard is associated to a transition, the transition is considered to be enabled always. A guard is associated to a specific transition based on their identifiers. An identifier is associated to a guard annotation as @Guard(“transitionname”), where transitionname is a String matching the name of a transition to which it is associated. If @Guard is present with no name given, it is associated to all transitions in the model. It is also possible to associate a single guard to several transitions using the notation of @Guard({“name1”, “name2”}) where the associated transitions are given as a list of strings. Every guard method is always executed to identify all enabled transitions 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 OSMOTester internal model representation.

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

It is also possible to give weights to transitions, using the notation of @Transition(name=”name”, weight=2). In this case, name is the name of the transition and the weight defines how many times the transition should be taken in relation to other available transitions. The default value for a weight is 1, so any transition with undefined weight has a weight of 1. When multiple transitions are enabled, a test generation algorithm that takes weights into account will favor the ones with more weight more often. For example, if transitions A has a weight of 1 and transition B a weight of 2, and both are always enabled at the same time, transition B will be taken twice as often as transition A. An example of a weighted algorithm is the WeightedRandomAlgorithm.

For example, in Listing 1, there is a transition and a guard named “vend”. The guard checks that this transition can only be taken if there are 100 coins or more inserted. After this becomes true, the vend transition is enabled. There is also a general guard statement in the form of the method gotBottles(). This makes sure that any transition can only be taken (coins inserted, vending applied) when there are some bottles in the machine. Notice that this is in this case practically a pointless guard since there is an end condition making sure this state is never achieved. But it serves to illustrate the concept. We could also add a new transition called “change” that would return all inserted coins and this could be enabled even when no bottles are present. In this case the guard would become @Guard({“10cents”, “20cents”, “50cents”, “vend”}).

Before a transition is taken, all associated @Pre annotated methods are executed. These are named and associated to transitions similar to guard statements. If the method has the Map parameter, a single Map is passed for each pre-method. This is the same instance each time for the same transition. Thus any values set in the pre-methods are accessible in all post-methods associated to that transition.

After a transition is taken, all associated @Post annotated methods are executed. These are named and associated to transitions similar to guard statements. If the Map parameter is provided, it will be the same as was provided for any @Pre method for that transition. Thus any values set in the pre-methods are accessible in all post-methods associated to that transition.

When all test oracles associated to the currently executed test step (transition) 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 strategy is 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 transitions and continue again from there. In Listing 1 there is one end condition that makes sure test generation is ended when there are no bottles present as otherwise this state machine would throw an exception as there would be no enabled state left.

When a test generation for a single test case is evaluated as finished, all methods annotated with @AfterTest are executed. Once this has been executed, the test suite strategy is evaluated to check whether the generation engine should continue with the next test case generation or to stop all test generation. If test generation is continued, the test generation continues with the next test case and the test generation continues from the @Before annotated methods.

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

# Test generation

OSMOTester generates test cases from the given test model. The test generation is based on the annotations described above. Note that if there are several methods annotated with a specific annotation that are available at a given time, the order in which they are executed is unspecified. Only for transitions the choice is taken by the chosen algorithm. For the rest, the ordering should be considered unspecified and possibly random.

Besides the annotations, it is also possible to configure OSMOTester 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 transitions 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();

}

}

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

public static void main(String[] args) {

1 OSMOTester osmo = new OSMOTester();

2 osmo.addModelObject(new VendingExample());

3 osmo.addTestEndCondition(new LengthCondition(3));

4 osmo.addSuiteCondition(new LengthCondition(2));

5 osmo.setAlgorithm(new OptimizedRandomAlgorithm());

6 osmo.setDebug(true);

7 osmo.generate();

}

}

The lines here are the following:

1. Creates the OSMOTester test generation engine.
2. 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.
3. 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. The default condition is set to end with 5% probability.
4. 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 with 5% probability.
5. Sets the test generation algorithm. By default this is set to a RandomAlgorithm that randomly takes one of the available transitions. The theOptimizedRandomAlgorithm used here takes previously uncovered transitions if available.
6. This enables more verbose debug printing in System.out.
7. This invokes the test generation engine to generate tests from the given model objects with the defined configuration.

# 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:

@Transition(name="10cents", weight=2)

public void insert10cents() {

scripter.step("INSERT 10");

coins += 10;

}

With this change, the 10 cents transition would now be taken twice where the others are taken once (assuming it is enabled). To make the transition useful, we would also need to use a matching test generation algorithm:

osmo.setAlgorithm(new WeightedRandomAlgorithm());

**Requirements**

We could also use requirements 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”);

}

...

@Transition("vend")

public void vend() {

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

coins -= 100;

bottles--;

if (bottles == 0) {

req.covered(“all bottles vended”);

}

}

In this case, we add 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 String) must match in order for the requirement 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 must again apply a matching test generation algorithm:

osmo.addSuiteEndCondition(new RequirementsCoverageCondition());

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).

**Composition of End Conditions**

In addition to specifying specific end conditions, it is also possible to compose several to form a single end condition that is composed of several different end conditions. This can be done using the AndComposition and OrComposition end condition classes. Alone these have no meaning but together with others can combine several into one. For example, the following creates a test suite to have test cases that have a minimum length of 5 transitions taken in the model, maximum of ten transitions and the probability of 25% for having a length of 6-9 transitions taken:

osmo.addTestEndCondition(new LengthCondition(5));

EndCondition max = new OrComposition(new LengthCondition(10), new ProbabilityCondition(0.75d));

osmo.addTestEndCondition(max);

**Data-Flow Modelling**

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.dataflow 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. These obejcts includes the following elements:

* ReadableCharacterSet: Character data that is human-readable.
* 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) and on our work in defining a taxonomy of runtime invariance in software behavior. They will be updated over time with additional objects and features.

**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 transition taken, the test case identifier, and a possibility to mark the test case as failed (for online testing) and to store the test script with the test case for optimization of test order later.

**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 OSMOTester to describe the model state at different points of time. Currently OSMOTester will not act on them but will just 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 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 field name as available through reflection (typically the field name in the source code) and the value will be the value of the object stored in the field. In case of primitives or Objects not implementing the VariableValue, 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.

# Test optimization

Automatically generating a lot of test cases can be a nice boost to test coverage, east test maintenance and so on. But commonly there is a limit to how many tests can be executed. Using a combination of test generation algorithms and other components, a large number of tests can be generated. OSMOTester also comes with a set of basic optimizer components that can be applied to optimize the set of generated test cases in relation to a set of specified criteria.

Currently OSMOTester comes with two optimizers. RequirementsOptimizer optimizes test cases to cover a maximum number of requirements with a minimum number of test cases. TransitionOptimizer optimizes test cases to cover a maximum number of transitions with a minimum number of test cases. Both take as input a generated test suite and organize the ordering of test cases in the test suite so that the one that provides the most added coverage in relation to the used criteria always comes first (and the second one next, and so on). They do not remove any tests but any additional tests are added to the end of the test suite once all additional coverage has been achieved.

Additional coverage algorithms could implement, for example, parameter space optimization and combinations of different coverage options into multi-parameter optimization techniques. While OSMOTester does not currently support such features, it can be extended by the user in any way they wish.

# Customization of algorithms and other components

In addition to the existing test generation algorithms, suite end conditions and other components provided with OSMOTester, 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.

# Conclusions

OSMOTester 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 OSMOTester Javadocs. There are also more examples in the OSMOTester source code under the osmo.tester.examples package and in the source code test directory in form of JUnit tests.

# References

OSMOTester home page & source code: <http://code.google.com/p/osmo/>