User Guide

OSMOTester

MBT tool

Table of Contents

[Introduction 3](#_Toc294599823)

[Modeling notation 3](#_Toc294599824)

[Test generation 8](#_Toc294599825)

[Conclusions 9](#_Toc294599826)

[References 9](#_Toc294599827)

# 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.

# 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. 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. 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 the solid line. There is only one guard statement shown in this model, saying that *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. This could be described in terms of compositions of states such as more than 100 coins, more than zero bottles and so on. However, in MBT the transitions typically represent test steps and thus here the transitions are given more focus as they in practice form the steps of the generated test cases. Specific state conditions such as viewing the number of coins and bottles as state compositions to be covered by generated test cases (transition coverage).



Figure 1. Vending machine as a state-machine.

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 using a vending-machine as an 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;

}

@Before

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--;

}

@EndCondition

public boolean end() {

return bottles <= 0;

}

@Oracle

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.
* @Oracle: Defines a check that is evaluated after an associated transition is taken.
* @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.
* @Before: Called before a test case is generated.
* @After: 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 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.
* @Before: Method must not take any parameters.
* @After: Method must not take any parameters.
* @BeforeSuite: Method must not take any parameters.
* @AfterSuite: Method must not take any parameters.
* @Oracle: Method must not take any parameters.
* @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.



Figure 2. Generation flow with annotation elements.

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.

Before any test case is generated, all methods annotated with @Before are executed. New test case generation is considered to start at suite start, when an end condition returns true and suite strategy defines the test cases as continuing, and when test strategy tells to stop test generation but suite strategy tells to continue with generation of new test cases. 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. Transitions are intended to describe test steps.

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”}).

After a transition is taken, all associated @Oracle annotated methods are taken. These are named and associated to transitions similar to guard statements. They are intended to describe checks in terms of test oracles.

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 @After are executed. One 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 the test suite end strategy, test case end strategy, test generation algorithm, and debug logging.

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.setTestStrategy(new LengthStrategy(3));

4 osmo.setSuiteStrategy(new LengthStrategy(2));

5 osmo.setAlgorithm(new OptimizedRandomAlgorithm());

6 tester.setDebug(true);

7 tester.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 strategy for ending generation of single test cases. This strategy causes each generated test case to have 3 steps. By default this is set to end with 5% probability.
4. Sets the strategy for ending generation of all tests (the test suite). This strategy causes the generator to generate 2 tests. By default this is set to end with 5% probability.
5. Sets the test generation algorithm. By default this is set to randomly take one of the available transitions. With this optimized version, it 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.

For more details, check the OSMOTester Javadocs. You can define your own test strategies and algorithms. 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.

# Conclusions

OSMOTester provides means to create test models and to generate test cases from these models.

# References

OSMOTester home page: <http://osmo.testautomation.fi>

OSMOTester source code: <https://github.com/tkanstren/osmo>