User Guide

OSMO Tester

MBT tool

v3.3

Teemu Kanstrén

NOTE: You can also check the tutorials for shorter concrete information on a topic.

Table of Contents

[Introduction 3](#_Toc372289175)

[Modeling notation 4](#_Toc372289176)

[Test generation 9](#_Toc372289177)

[Special model elements 10](#_Toc372289178)

[Report builders 13](#_Toc372289179)

[Customization of algorithms and other components 13](#_Toc372289180)

[Examples 13](#_Toc372289181)

[Offline Optimization 13](#_Toc372289182)

[Online Optimizer 15](#_Toc372289183)

[Scenarios 16](#_Toc372289184)

[Conclusions 16](#_Toc372289185)

[References 16](#_Toc372289186)

# 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 test model to generate test from. This model can be seen as a form of a computer program and is sometimes referred to as 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.

The model program is essentially a set of rules and actions. The rules define when the actions are allowed, and the actions provide stimuli to the SUT. The rules are represented in the model program as “guard” statements. These are associated to test steps in the model, which invoke the actions. In the case of OSMO Tester, the model program is practically a Java program with specific annotations. The test generator part takes this model program, reads the annotations and invokes the test steps in different sequences as allowed by the guards to produce test cases. The model program typically has some state to reflect the state of the SUT. Guard statements can then use this state to evaluate when their associated test steps are allowed. Similarly, test oracles can be written to check this model state against the SUT state at different points.

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). A bottle costs 100 cents, so a minimum of 100 cents must have been inserted to be able to perform the “vend” action/test step. The model test steps are *insert money* and *vend*. *Insert money* inserts a random coin from supported coin types (e.g., 10, 20 or 50 cent coin). It is allowed always, so the guard would be always true. *Vend* is only allowed when 100 cents or more have been inserted. Thus the guard for it is true if the model state tells us we have at least 100 cents worth of coins inserted. Additionally, we could add another guard for both of these to disallow either step if there are no bottles left in the machine.



Figure 1. Vending-machine example.

For model state, *insert money* increases value of the variable that defines how many cents have been inserted by the coin size. *Vend* 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, …) using the available test execution framework.

# Modeling notation

This section describes the OSMO Tester modeling notation in terms of an example of the vending-machine described before. This is an example commonly used in MBT literature (e.g., [1]) and should provide some point of comparison for those interested. The 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.

Listing 1 illustrates the OSMO Tester notation for this example. The state of the model program are the four variables at the top. The actions/test steps are the methods annotated as *@TestStep*. The guards for these are the methods annotated as *@Guard*. Here we see how executing a step changes the model state, and how the guards enforce the relevant rules. For example, inserting a 10 cent coin increases the number of inserted cents by 10. The first guard (“all”) is matched to all test steps in the model to check that there are some bottles left to use. The second is only associated to the “vend” step to allow it when at least 100 cents have been inserted. The @Post(“all”) annotated method defines a test oracle to check after each test step that the state of the SUT matches the state of the model.

public class VendingExample {

private final Scripter scripter;

private int cents = 0;

private int bottles = 10;

private TestSuite suite;

public VendingExample() {

scripter = new Scripter(System.out);

}

@Guard(“all”)

public boolean gotBottles() {

return bottles > 0;

}

@BeforeTest

public void start() {

cents = 0;

bottles = 10;

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

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

}

@AfterSuite

public void done() {

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

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

}

@TestStep("10c")

public void insert10cents() {

scripter.step("INSERT 10");

cents += 10;

}

@TestStep("20c")

public void insert20cents() {

scripter.step("INSERT 20");

cents += 20;

}

@TestStep("50c")

public void insert50cents() {

scripter.step("INSERT 50");

cents += 50;

}

@Guard("vend")

public boolean allowVend() {

return cents >= 100;

}

@TestStep("vend")

public void vend() {

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

cents -= 100;

bottles--;

}

@EndCondition

public boolean end() {

return bottles <= 0;

}

@Post(“all”)

public void checkState() {

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

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

}

public static void main(String[] args) {

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

tester.generate(55);

}

}

Listing 1. Example vending machine model in OSMO Tester notation.

The following annotations are supported by OSMO Tester:

* @Guard: Defines a rule for an associated test step to be enabled.
* @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 rule to have the generator stop current test case generation.
* @BeforeTest: Called before a new test case is started.
* @AfterTest: Called after a test case 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.
* @LastStep: Similar to @AfterTest but the annotated method is executed as part of the actual test generation flow. This means if it throws an Exception, the test case fails and not the test generation process.
* @Variable: Identifies a variable value to capture for analysis purposes.
* @CoverageValue: Allows defining custom calculations for targeted coverage values.
* @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 is the generation seed value. The test generator makes heavy use of randomization to produce test case variants. The seed defines what random values are generated. It is passed as an argument to the generator when invoking the generate() method as illustrated in Listing 1. This enforces deterministic test generation. If you want non-deterministic generation, use a random source such as the system clock to generate the seed itself.



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. Some constraints on the method and field signature are as follows:

* @Guard: Method must return Boolean value. Method must not take any parameters.
* @TestStep: 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.
* @LastStep: Method must not take any parameters.
* @Pre: Method must not take any parameters.
* @Post: Method must not take any parameters.
* @EndCondition: Method must return Boolean value. Method must not take any parameters.
* @Variable: Field can be of any type. If it implements the VariableValue interface, the value returned by the value() method is stored.
* @CoverageValue: 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.
* @Group: Defines a class-level grouping for test steps.

There are also two special types of model variables. These are:

* Requirements: Only one instance of this type of object is allowed in any given model objects. Note that you can share the same instance in several classes. It must be non-null and initialized by the user. The generator then uses this to track the set of covered requirements in the model as tests are generated. The requirements will be discussed in detail later.
* TestSuite: Must be set to null. The generator will set this field value before test suite generation is started. Can be used to track the status of the generated test suite in the model.

Description of the annotations:

**@BeforeSuite**

These methods are executed only once during test generation before any tests are generated. Suite refers to all generated test cases and a test case refers to a set of test steps. This annotation is useful for one time setup of state or test resources before starting test generation.

**@BeforeTest**

Before a test case is generated, all methods annotated with @BeforeTest are executed. For example, in Listing 1 there is one method called start() with this annotation, which resets the model state between generated tests.

**@Guard**

Defines rules for associated test steps and when they are allowed. 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 a @Guard defines no name, it is associated to a single step based on the method name by removing any lower-caps text from the beginning. For example, a method @Guard public boolean gBob() is associated to a test step named “Bob”. Note that step name association is always case insensitive so “bob” and “Bob” is the same thing.

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, or with @Guard(“all”), which causes it to be associated to all steps in the model. Every guard method is always executed to identify all enabled steps. 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 given name for the guard must match the name of the group. The name given for the guard is matched to either a step name or a group name, whichever the generator finds the match for. Duplicates are not allowed and the parser will complain if it sees a duplicate step/group name.

**@TestStep**

Test steps are named similar to guards, i.e. @TestStep, @TestStep(“name”), @TestStep({“name1”, “name2”}). Similar to guards, if there is no name parameter, it is picked from the method definition. However, unlike guards the whole name of the method is used. For example, @TestStep public void bob() would define a test step named “bob”. The set of enabled steps are identified by the 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.

It is also possible to define a group name for a test step. This is done with a group field 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”).

**@Group**

Besides defining groups for specific test steps, it is also possible to define groups for every step in a class at once. This is done with a class level @Group annotation. If the class has a @Group annotation and contains a @TestStep method with a specific group definition, the specific group for the step overwrites the class level definition for that step.

**@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.

**@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.

**@EndCondition**

After executing a chosen test step, the generator executes all @EndCondition tagged methods. If one returns true, the current test is ended. 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. Note that there are specific end condition objects that can be used to configure the generator and those are the general way to configure an end condition. This annotation is to make it easier to define one that uses the model state for customization.

**@AfterTest**

When test generation for a single test case is finished, all methods annotated with @AfterTest are executed.

**@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

The purpose of OSMO Tester is to generate test cases by executing the given model program in different ways based on the annotations described above. Note that the specific order of executing several guards or pre/post methods is not explicitly defined. It should be deterministic (at least when using a specific generator version on same platform) 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 test suite end condition, a test case end condition, a test generation algorithm, and enabling or disabling 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(55);

}

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. You can add as many as you like, and they will be combined together where steps, 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 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 steps, the generator stops). The default condition used, if no end condition is set by the user, is to end after generating a minimum of 1 test step, with 5% probability after each test steps.
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 giving higher probability to less covered of the available steps and step-pairs.
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 steps, requirements and state variable definitions.

**Weighted Steps**

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

@TestStep(name="10c", 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 can 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. 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 into one. This can be done using the And and Or end conditions. Alone these have no meaning but together with others can combine several end conditions into one. For example, the following specifies that test cases need to have a minimum length of 5 steps, maximum of 10 steps and stop at any step between 5 and 10 with the probability of 25%:

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);

This can be rather confusing which is why the default set of provided end conditions tries to cover the common cases. It is also possible for you to write your own end condition by extending the EndCondition class, which can be a much more intuitive way to achieve the same.

**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. 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). ValueSet also gives you a chance to mark some inputs as used by the reserve() method, and back to unused with free() method.

**State Variable Identification**

The @Variable annotation can be used to identify state variables to track in the model. The generator will then store each variable value for every test step. The value of the variable is taken as the object value or if the object implements the VariableValue interface, then by invoking that interface. The name used for the variable is either the field name from the code or the parameter given to annotation.

# 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.getStepCounts();

This will create 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.

A useful features is also the ModelPrinter class, which provides ways to print the structure of the model after parsing. This gives you a list of found steps, guards, pre-, and post- methods as well as their associations to each other.

# Customization of algorithms and other components

To create a test generation algorithm, 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.

# Examples

Have a look at the examples module (in the source repository) and the unit tests for several detailed examples.

# Offline Optimization

An “optimizing” 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, the resulting suite is provided as the result.

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 a @CoverageValue method gives N points.
* State-pairs: Covering different values in sequence for the custom states gives N points.

Notice the all the scores are calculated at the suite level. Thus, if one test case/step already covers a specific criteria, following test cases/steps will score nothing for it. The following is an example of running the offline optimizer:

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

public static void main(String[] args) {

ScoreConfiguration sc = new ScoreConfiguration();

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

sc.setLengthWeight(0);

sc.setRequirementWeight(100);

sc.setStateWeight(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

}

As a result of running the offline optimizer, we get 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 in the current test case using the method TestSuite.getCurrentTest().setAttribute(String).

The simplest way to get access to these is to iterate the given test set and write the final 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 instantiate a given number of single-core instances to run in parallel. Finally, it merges all of their results to provide a single optimized set.

# Online Optimizer

OSMO Explorer 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. This is a form of online optimization. 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). Additionally, this can also be used as an alternative optimization approach for offline test generation by supplying an offline scripter if desired.

The basic form of use is to first initialize a coverage score configuration (as discussed above for offline optimization). 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() {

MyModelFactory factory = new MyModelFactory(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). The score configuration is created in the form of the ExplorationConfiguration object in this case. Check the Javadoc for some more detail on what you can configure.

Another thing to note here is the use of the ModelFactory. The explorer will at the point of each test step to be taken initiate new versions of the model objects. For this reason, you need to provide a factory for creating your model. The @ExplorationEnabler and @GenerationEnabler methods are then invoked for the instances used to explore and generate respectively.

A related component is MultiOSMO. This is a version of the basic generator that runs a number of test generators in parallel similar to the GreedyOptimizer but does not try to optimize them afterwards. It is intended to run several online test generation/execution instances in parallel on a modern multicore system.

# Scenarios

Besides having the tool optimize test coverage, it is often also interesting to focus test generation on specific aspects of the test model. In OSMO Tester the means to do this is test scenarios. You can use these both with the basic test generator as well as the explorer. The following shows an example for the basic generator:

tester = new OSMOTester();

config = tester.getConfig();

scenario = new Scenario(false);

scenario.addStartup("start", "increase", "increase", "increase", "increase");

config.addModelObject(new CalculatorModel(NullPrintStream.stream));

config.setScenario(scenario);

config.setSuiteEndCondition(new Length(5));

This is an example borrowed from the OSMO Tester test models. It defines a scenario to start with the sequence of five steps, whose name is “start”, “increase”, “increase”, “increase”, “increase”. Thus every test case generated will start with this same sequence. From this initial point forward, the following steps can be any of the given steps. To limit the number of times a specific test steps is allowed or required in the generated test cases, we can also add a slice:

scenario.addSlice("decrease", 0, 2);

The 0 here is the minimum number of times the “decrease” step is required in generated test cases. The “2” is the maximum number of times it is allowed. This only applies after the startup sequence if specified.

To define scenarios for the explorer, the process is the same but you need to configure the scenarios into the ExplorationConfiguration.

CounterFactory factory = new CounterFactory();

config = new ExplorationConfiguration(factory, 2, 444);

explorer = new OSMOExplorer();

scenario = new Scenario(false);

scenario.addStartup("start", "increase", "increase", "increase", "increase");

config.setScenario(scenario);

config.setFactory(factory);

config.setMinSuiteLength(5);

config.setMinTestLength(5);

That’s all. Now go generate some tests or something...

# 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/>