Tutorial for Basic Features

OSMOTester

MBT tool

v2.1

Teemu Kanstrén

Table of Contents

[Introduction 3](#_Toc314955899)

[Hello World 3](#_Toc314955900)

[Controlling the length of generated tests 7](#_Toc314955901)

[Adding more test steps 9](#_Toc314955902)

[Conclusions 13](#_Toc314955903)

[References 13](#_Toc314955904)

# Introduction

This tutorial describes the basic modeling concepts of OSMO Tester using simple examples. It skips the background, the theoretical descriptions and just shows some examples. Check the OSMO manual for more lengthy descriptions.

The reader is expected to have basic knowledge of Java programming and ability to use their own favourite IDE such as Eclipse, IntelliJ, or Netbeans. The code shown in this tutorial is available in the OSMO Tester examples package.

# Hello World

The test models for OSMOTester are in practice executable programs written in the Java programming language. A good way to think about them is actually in terms of a “Model Program”. We start with a simple example model that just prints “HELLO” on the console. Listing 1 shows an example model that does just that.

public class HelloModel {

@TestStep("hello")

public void sayHello() {

System.out.println("HELLO");

}

}

Listing 1. A model program that prints “HELLO”.

Cute. So how do we run this model? We run the OSMO Tester generator as a normal Java program and pass it the model object as an argument. This is shown in Listing 2

public class Main {

public static void main(String[] args) {

OSMOTester tester = new OSMOTester(new HelloModel());

tester.generate();

}

}

Listing 2. Running the model program.

Now when we run the Main class (assumption: you know how to compile and run a Java program), we get output that is something like shown in Figure 1.

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

generated 4 tests.

Figure 1. Example output.

Notice that since we have not configured any options on the OSMO Tester generation, your output may be different since the randomization seed is taken from the system clock and the output is basically different every time in this case.

Basically, this output tells us that OSMO Tester generated 4 different test cases from the model. Since they all just print “HELLO” on the console and nothing in between, all the printouts look like a long list where the text is merged into one single long list.

To make the output deterministic, we can set the seed used in randomization. Listing 3 shows how to do this, with the changed part in **bold**.

public class Main {

public static void main(String[] args) {

OSMOTester tester = new OSMOTester(new HelloModel());

**tester.setSeed(345);**

tester.generate();

}

}

Listing 3. Setting the seed for deterministic generation.

Running the modified model should now always yield the output shown in Figure 2.

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

generated 5 tests.

Figure 2. Example output.

Sweet. But wouldn’t it be nice to see how many “HELLO”’s belong to each test?

To get this, we add some additional OSMO elements to the model. The modified model is shown in Listing 4.

public class HelloModel {

**@BeforeTest**

**public void startTest() {**

**System.out.println("TEST START");**

**}**

**@AfterTest**

**public void endTest() {**

**System.out.println("TEST END");**

**}**

@TestStep("hello")

public void sayHello() {

System.out.println("HELLO");

}

}

Listing 4. Modified model program showing a test start and end.

Running this model provides the output shown in Figure 3.

TEST START

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

TEST END

TEST START

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

TEST END

TEST START

HELLO

TEST END

TEST START

HELLO

TEST END

TEST START

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

HELLO

TEST END

generated 5 tests.

Figure 3. Example output.

This @BeforeTest and @AfterTest is what you would actually use to set up and tear down your generated test cases.

# Controlling the length of generated tests

Now, we see that there is a lot of variation in the length of the generated test cases. This is because the generator is not configured with information on what length of tests are wanted and the default uses a random probability and minimum length of 1 to end a test case and test suite. We can also specify the length by configuration as shown in Listing 5.

public class Main {

public static void main(String[] args) {

OSMOTester tester = new OSMOTester(new HelloModel());

tester.setSeed(345);

**tester.addTestEndCondition(new Length(5));**

**tester.addSuiteEndCondition(new Length(3));**

tester.generate();

}

}

Listing 5. Setting specific end conditions.

Running this new model now produces the output shown in Figure 4.

TEST START

HELLO

HELLO

HELLO

HELLO

HELLO

TEST END

TEST START

HELLO

HELLO

HELLO

HELLO

HELLO

TEST END

TEST START

HELLO

HELLO

HELLO

HELLO

HELLO

TEST END

generated 3 tests.

Figure 4. Example output.

OSMO Tester Javadocs describe a number of algorithms that can be used as end conditions. Changing the ones used can provide a powerful means to control test generation for different types of variants in terms of length. It is also possible to combine several into one as shown in Listing 6.

public class Main {

public static void main(String[] args) {

OSMOTester tester = new OSMOTester(new HelloModel());

tester.setSeed(345);

**tester.addTestEndCondition(**

**new And(new Length(5), new Probability(0.33)));**

**tester.addSuiteEndCondition(new Length(3));**

tester.generate();

}

}

Listing 6. Combined end conditions.

This results in test cases with a minimum length of 5 and after that ending with a probability of 33%. From the program viewpoint, it requires both conditions to be true so that length is at least 5 and generating a random value between 0 and 1 gives a result smaller than 0.33. The output is left as an exercise to the reader.

# Adding more test steps

So far, the examples have focused only on one test step. No real test case has only one possible step so let’s add some more. Listing 7 shows a second step added to the model, one that prints “WORLD”.

public class HelloModel {

@BeforeTest

public void startTest() {

System.out.println("TEST START");

}

@AfterTest

public void endTest() {

System.out.println("TEST END");

}

@TestStep("hello")

public void sayHello() {

System.out.println("HELLO");

}

**@TestStep("world")**

**public void sayWorld() {**

**System.out.println("WORLD");**

**}**

}

Listing 7. Modified model program showing a test start and end.

This results in the output shown in Figure 5. The configuration for execution used in this section is the one shown in Listing 5.

TEST START

WORLD

HELLO

WORLD

HELLO

HELLO

TEST END

TEST START

WORLD

WORLD

HELLO

WORLD

WORLD

TEST END

TEST START

HELLO

HELLO

HELLO

WORLD

WORLD

TEST END

generated 3 tests.

Figure 5. Example output.

That’s nice. But in real life things have to happen in correct order for it to make sense. So what if we only want “WORLD” to appear if “HELLO” has already appeared? We modify the model by adding a guard statement for the “WORLD” part. This is shown in Listing 8.

public class HelloModel {

**private int helloCount = 0;**

@BeforeTest

public void startTest() {

**helloCount = 0;**

System.out.println("TEST START");

}

@AfterTest

public void endTest() {

System.out.println("TEST END");

}

@TestStep("hello")

public void sayHello() {

System.out.println("HELLO");

**helloCount++;**

}

**@Guard("world")**

**public boolean thisNameIsIrrelevant() {**

**return helloCount >= 1;**

**}**

@TestStep("world")

public void sayWorld() {

System.out.println("WORLD");

}

}

Listing 8. Modified model program with state and a guard condition.

This association works because both the TestStep and Guard have the same parameter name “world”. For the first time it is also visible here how the model has state that guides test generation. In this case the state is the number of times “HELLO” has been printed. The state is also reset at the beginning of each test generation or otherwise the condition would only apply in the first generated test case (which would increment the value to >= 1 and the rest of the tests would always see this as the state). The resulting output is shown in Figure 6.

TEST START

HELLO

HELLO

WORLD

HELLO

HELLO

TEST END

TEST START

HELLO

WORLD

HELLO

WORLD

WORLD

TEST END

TEST START

HELLO

HELLO

HELLO

WORLD

WORLD

TEST END

generated 3 tests.

Figure 6. Example output.

Notice how in this case each test cases starts with “HELLO” as we wanted. But what if we want to have “HELLO” and “WORLD” to appear always in pairs? We add another state variable for the number of “WORLD”’s and also a guard for “HELLO”. This is shown in Listing 9.

public class HelloModel {

private int helloCount = 0;

**private int worldCount = 0;**

@BeforeTest

public void startTest() {

helloCount = 0;

worldCount = 0;

System.out.println("TEST START");

}

@AfterTest

public void endTest() {

System.out.println("TEST END");

}

**@Guard("hello")**

**public boolean thisNameReallyIsIrrelevant() {**

**return helloCount == worldCount;**

**}**

@TestStep("hello")

public void sayHello() {

System.out.println("HELLO");

helloCount++;

}

@Guard("world")

public boolean thisNameIsIrrelevant() {

**return helloCount > worldCount;**

}

@TestStep("world")

public void sayWorld() {

System.out.println("WORLD");

**worldCount++;**

}

}

Listing 9. Modified model program with added state and a guard condition.

In this case our model tells that “HELLO” can appear if an equal number of “WORLD”’s has been printed. The “WORLD” can only appear if more “HELLO”’s have been printed than “WORLD”’s. Thus this will always print the text is pairs, with “HELLO” preceding “WORLD”. Figure 7 shows the printout for executing this model.

TEST START

HELLO

WORLD

HELLO

WORLD

HELLO

TEST END

TEST START

HELLO

WORLD

HELLO

WORLD

HELLO

TEST END

TEST START

HELLO

WORLD

HELLO

WORLD

HELLO

TEST END

generated 3 tests.

Figure 7. Example output.

And what if we wanted always to have “WORLD” as the last word (hah!)? We could just modify the @AfterTest method to compare that helloCount and worldCount are equal and if not, print out the last “WORLD”.

# Conclusions

This tutorial showed the very basics of building and running models with OSMO Tester. For more information on all the model elements and how to use them, check the OSMO Tester manual and other tutorials.

# References

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