Chapter 1

Testing JPF Components

In order to test components of JPF, such as listeners, search strategies and reporters, JPF provides a testing framework similar to JUnit. We assume that the reader is already familiar with JUnit. For an introduction to Junit, we refer the reader to junit.org/junit4.

Before we introduce JPF's testing framework, let us first consider the two components that are needed to test that JPF can detect failed assertions. We need an app with an assertion that fails and an application properties file. As app, we can use, for example, the following.

```
public class FailedAssertion {
  public static void main(String[] args) {
    assert false;
  }
}
```

To run JPF on this app, we create the following application properties file.

```
target=FailedAssertion
classpath=.
```

If we run JPF on the above application properties file, it produces the following output.

```
JavaPathfinder core system v8.0 (rev 26e11d1de726c19ba8ae10551e048ec0823aabc6) - (C) 2005-2


====================================================== system under test
FailedAssertion.main()

====================================================== search started: 2/24/20 4:08 PM

====================================================== error 1
gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
java.lang.AssertionError
      at FailedAssertion.main(FailedAssertion.java:3)


====================================================== snapshot #1
thread java.lang.Thread:{id:0,name:main,status:RUNNING,priority:5,isDaemon:false,lockCount
  call stack:
      at FailedAssertion.main(FailedAssertion.java:3)
```

```
====================================================== results
error #1: gov.nasa.jpf.vm.NoUncaughtExceptionsProperty "java.lang.AssertionError
at FailedAssertion.main(..."

====================================================== statistics
elapsed time:  00:00:00
states:        new=1,visited=0,backtracked=0,end=0
search:        maxDepth=1,constraints=0
choice generators: thread=1 (signal=0,lock=1,sharedRef=0,threadApi=0,reschedule=0), data=0
heap:          new=366,released=0,maxLive=0,gcCycles=0
instructions:  3207
max memory:    57MB
loaded code:   classes=65,methods=1371

====================================================== search finished: 2/24/20 4:08 PM
```

From the above output we can conclude that JPF indeed detects the failed assertion in the app. Checking the output produced by JPF manually is cumbersome. Having the app and the corresponding properties in two different files introduces the risk that the files get out of sync when changing either file. JPF's framework addresses both shortcomings: it allows us to check the result of the test case programmatically and it also combines the app and its properties into a single file.

1.1 A Simple Example

Before discussing the details of JPF's testing framework, let us first recast the above test in the framework. The abstract class `TestJPF`, which is part of the package `gov.nasa.jpf.util.test`, is the heart of JPF's framework. To implement a test, we extend this class using the following skeleton.

```
import gov.nasa.jpf.util.test.TestJPF;
import org.junit.Test;

public class ...Test extends TestJPF {

  @Test
  public void test...() {
    ...
  }
}
```

To test that JPF detects failed assertions, we combine the above app and the corresponding properties file into a single test method as follows.

```
1 public class AssertionTest extends TestJPF {
2
3   @Test
4   public void testAssertFails() {
5     if (this.verifyAssertionError("+classpath=.")) {
6       assert false;
7     }
8   }
9 }
```

The method `verifyAssertionError` takes as argument the content of the application properties file. Note that these properties are provided in the same format as when given to JPF as command line arguments, that is, each property is prefixed by a `+`. Line 6 contains the body on the `main` method of the app.

To test that a successful assertion is handled properly by JPF, we can add, for example, the following test method to the above class.

```
@Test
public void testAssertSucceeds() {
  if (this.verifyNoPropertyViolation("+classpath=.")) {
    assert true;
  }
}
```

1.2 Running a JPF Test

A JPF test can be run as an ordinary JUnit test in Eclipse. For convenience, we add a `main` method, as shown below, so that the JPF test can also be run as an ordinary app. This app takes the names of the test methods to be run as command line arguments. If no command line arguments are provided, then all tests are run.

```
public static void main(String[] testMethods) {
  TestJPF.runTestsOfThisClass(testMethods);
}
```

When we run the above JPF test as an ordinary app, it produces the following output.

```
..................................... testing testAssertSucceeds()
  running jpf with args: +classpath=.
JavaPathfinder core system v8.0 (rev 32+) - (C) 2005-2014 United States Government. All ri


====================================================== system under test
AssertionTest.runTestMethod()

====================================================== search started: 2/25/20 4:25 PM

====================================================== results
no errors detected

====================================================== search finished: 2/25/20 4:25 PM
..................................... testAssertSucceeds: Ok

..................................... testing testAssertFails()
  running jpf with args: +classpath=.
JavaPathfinder core system v8.0 (rev 32+) - (C) 2005-2014 United States Government. All ri


====================================================== system under test
AssertionTest.runTestMethod()

====================================================== search started: 2/25/20 4:25 PM
```

```
====================================================== error 1
gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
java.lang.AssertionError
      at AssertionTest.testAssertFails(test/BasicTest.java:25)
      at java.lang.reflect.Method.invoke(gov.nasa.jpf.vm.JPF_java_lang_reflect_Method)
      at gov.nasa.jpf.util.test.TestJPF.runTestMethod(gov/nasa/jpf/util/test/TestJPF.java:


====================================================== snapshot #1
thread java.lang.Thread:{id:0,name:main,status:RUNNING,priority:5,isDaemon:false,lockCount
  call stack:
      at gov.nasa.jpf.util.test.TestJPF.runTestMethod(TestJPF.java:650)


====================================================== results
error #1: gov.nasa.jpf.vm.NoUncaughtExceptionsProperty "java.lang.AssertionError
at AssertionTest.testAs..."

====================================================== search finished: 2/25/20 4:25 PM
..................................... testAssertFails: Ok

..................................... execution of testsuite: AssertionTest SUCCEEDED
.... [1] testAssertSucceeds: Ok
.... [2] testAssertFails: Ok
..................................... tests: 2, failures: 0, errors: 0
```

Now that we have developed and run a simple example of a JPF test, let us dive into the details.

1.3 The `verifyNoPropertyViolation` Method

We have already seen the `verifyNoPropertyViolation` method in the above simple example. The method takes as argument an array of strings. These are provided to JPF as command line arguments. Usually, these are the properties one would put in the application properties file. However, one can also use command line arguments such as `-log` and `-show` as discussed in Section **??**. The method expects that no property is violated when JPF is run on the test method with the given command line arguments. If, however, a property violation is encountered by JPF, then the test fails.

Recall that JPF is a virtual machine and that JPF, since it is implemented in Java, runs on a Java virtual machine. We call the latter the host JVM. When a JPF test method is run, it is both executed by the host JVM and model checked by JPF, that is, executed by JPF's virtual machine. Consider the following test method.

```
1 private static final String PROPERTIES = { "+classpath=." };
2
3 @Test
4 public void test() {
5   System.out.println("1");
6   System.out.println(this.verifyNoPropertyViolation(PROPERTIES));
7   System.out.println("2");
8 }
```

When the above test is run, it is executed by the host JVM. In line 6, the host JVM invokes the `verifyNoPropertyViolation` method. As a result, JPF is invoked on the test method itself with the argument of `verifyNoPropertyViolation` as its command line arguments. Subsequently, JPF model checks the test method. In line 6, JPF's virtual machine invokes the `verifyNoPropertyViolation` method. This method invocation simply returns true. Once JPF has

model checked the test method, the host JVM continues its execution of the test method with the invocation of the method `verifyNoPropertyViolation`. This time the method returns false. (For the curious reader, the fact that the method `verifyNoPropertyViolation` behaves differently when being executed by the host JVM from being executed by JPF's virtual machine is accomplished by introducing a native peer class.) Subsequently, the host JVM executes the remainder of the test method. The above test method, executed as a JUnit test, is successful and gives rise to the following output.

```
1
  running jpf with args: +classpath=.
JavaPathfinder core system v8.0 (rev 32+) - (C) 2005-2014 United States Government. All ri


============================================================ system under test
SampleTest.runTestMethod()

============================================================ search started: 2/25/20 4:48 PM
1
true
2

============================================================ results
no errors detected

============================================================ search finished: 2/25/20 4:48 PM
false
2
```

The host JVM first executes line 5 of the test method, resulting in the output on line 1. Next, JPF is run on the test method, giving rise to line 2–17 of the output. Note that the invocation of the `verifyNoPropertyViolation` method by JPF returns true, as can be seen on line 11 of the output. Subsequently, the host JVM executes the remainder of the test method: the `verifyNoPropertyViolation` method invocation returns false, which can be seen at line 18 of the output, and finally 2 is printed.

Although the `verifyNoPropertyViolation` method can be invoked multiple times in a test method, resulting in JPF being run multiple times, we will only include a single invocation of the `verifyNoPropertyViolation` method in our test methods.

Let us revisit the test that checks if a successful assertion is handled properly by JPF.

```
@Test
public void testAssertSucceeds() {
  if (this.verifyNoPropertyViolation(PROPERTIES)) {
    assert true;
  }
}
```

When executing the test `testAssertSucceeds`, the host JVM first invokes the `verifyNoPropertyViolation` method. As a result, JPF is run on the `testAssertSucceeds` method. When JPF invokes the `verifyNoPropertyViolation` method, the method returns true. As a result, JPF executes, that is, model checks, also line 4. Once JPF has model checked the `testAssertSucceeds` method, the host JVM returns from the invocation of the `verifyNoPropertyViolation` method. Since the return is false this time, the host JVM does not execute line 4 of the `testAssertSucceeds` method. Although preventing the host JVM from executing this line of code does not impact the test, this pattern will be very helpful in cases we will discuss next.

1.4 The `verifyAssertionError` Method

The method `verifyAssertionError` also takes an array of strings as argument. Again, these are provided to JPF as command line arguments. The method expects that JPF encounters an assertion error when run on the test method. If JPF does not encounter such an error, the test fails.

Consider the following test method.

```
1 @Test
2 public void test() {
3   System.out.println("1");
4   System.out.println(this.verifyAssertionError(PROPERTIES));
5   System.out.println("2");
6 }
```

The above test method, executed as a JUnit test, fails and produces the following output.

```
1  1
2  running jpf with args: +classpath=/courses/4315/workspace/4315/bin/
3  JavaPathfinder core system v8.0 (rev 32+) - (C) 2005-2014 United States Government. All ri
4
5
6  ============================================================ system under test
7  SampleTest.runTestMethod()
8
9  ============================================================ search started: 2/25/20 9:10 PM
10 1
11 true
12 2
13
14 ============================================================ results
15 no errors detected
16
17 ============================================================ search finished: 2/25/20 9:10 PM
```

The host JVM first executes line 3 of the test method, resulting in the output on line 1. Next, JPF is run on the test method, giving rise to line 2–17 of the output. Note that the invocation of the `verifyAssertionError` method by JPF returns true, as can be seen on line 11 of the output. Subsequently, the host JVM executes the remainder of the test method: the `verifyAssertionError` causes the test to fail, because no assertion error was detected while model checking the test method. Hence, no further output is produced.

Let us return to the following test method that we have already seen earlier.

```
1 @Test
2 public void testAssertFails() {
3   if (this.verifyAssertionError("+classpath=.")) {
4     assert false;
5   }
6 }
```

When executing the test `testAssertFails`, the host JVM first invokes the `verifyAssertionError` method. As a result, JPF is run on the `testAssertFails` method. When JPF invokes the `verifyAssertionError` method, the method returns true. As a result, JPF executes, that is, model checks, also line 4, and detects an assertion error. Once JPF has model checked the `testAssertFails` method, the host JVM returns from the invocation of the `verifyAssertionError` method. Since the return is false this time, the host JVM does not execute line 4 of the `testAssertFails` method. In this example, it is essential that the host JVM does not execute line 4. If it were to execute that line, the test would fail.

1.5 The `verifyUnhandledException` Method

1.6 The `verifyPropertyViolation` Method

1.7 The `verifyDeadlock` Method

1.8 The `isJPFRun` and `isJUnitRun` Methods