JPF-Core-X:
Tool Verification Cases and Procedures (TVCAP)

Contents

Introduction

This document is one of a several exemplar documents prepared as part of a research Case Study whose objective is to simulate a Formal Methods Tool qualification exercise under DO-330. The specific tool considered in this case study is the core module of Java PathFinder (JPF-Core). As with any tool qualification exercise, the qualification is done with respect to a specific version of the tool. Therefore, throughout this document, we refer to our version of JPF-Core as "JPF-Core-X", as described in Section a of the Tool Qualification Plan. This particular document provides a representative example of the "Tool Verification Cases and Procedures" (TVCAP) for JPF-Core-X, and is written according to the guidelines in DO-330 Section 10.2.5.

Because this is a research study, in which there is no actual qualifying organization and accompanying context, and because the tool under consideration is a research implementation without specific versions, release control, user documentation, or standard configurations, some of the sections of an actual TVCAP will not be relevant. This document is thus a Framework for an actual TVCAP, organized according to the DO-330 enumeration of required contents. Accordingly, some sections will represent content that is concrete enough to be part of an actual TVCAP; some will discuss how a more concrete implementation of this tool might fulfill the required contents; and some will not be applicable for the purposes of this research simulation.

The sections that follow are organized according to sub-parts a) through c) of Section 10.2.5 in DO-330. In each section, we attempt to provide representative content of what should appear in an actual TVCAP for a real qualification exercise. In addition, we offer supplemental *meta-level* comments throughout the document.

Discussion

This is how a meta-level comment appears in the text. These comments are meant to provide insight into our process of writing the document, and to suggest interesting or important topics that relate to the qualification of formal methods tools.

Organization of the Document

Section 10.2.5 of the authoritative DO-330 states the following:

The Tool Verification Cases and Procedures detail how the tool verification process activities are implemented. This data should include:

1. *Review and analysis procedures: The scope and depth of the review or analysis methods to be used, in addition to the description in the Tool Verification Plan.*
2. *Test cases: The purpose of each test case, set of inputs, conditions, expected results to achieve the required coverage criteria, and the pass/fail criteria.*

3. *Test procedures: The step-by-step instructions for how each test case is to be set up and executed, how the test results are evaluated, and the test environment to be used.*

This document is organized according to the above outline. Section a. describes the scope and depth of the analysis methods to be used; Section b. describes the test cases; and Section c. describes the test procedures for each of our test categories.

a. Review and Analysis Procedures

The tool verification plan for JPF-Core-X involves running two different categories of tests.

1. Full end-to-end tests of JPF-Core-X to verify that it appropriately finds (or confirms the absence of) certain property violations for specific examples of SUT's. Each of these tests support higher-level usage requirements that are listed in the TOR document, such as the requirement to find and report all deadlocks.

2. A suite of low-level unit tests to verify correct implementation of distinct methods in JPF-Core-X. Each of these tests support one or more lower-level functional requirements that are identified in the TR document.

The purpose and scope of the end-to-end tests is to demonstrate that JPF-Core-X satisfies the external interface requirements and the operational requirements that are listed in the Tool Operational Requirements (TOR) document. These tests are particularly important because they involve full execution of the entire tool. In other words, these tests call JPF-Core-X in the same manner that an actual user would under normal circumstances.

Taken by themselves, however, the end-to-end tests do not provide sufficient evidence that the tool is implemented correctly. The low-level unit tests verify the correct implementation of JPF-Core-X with respect to the requirements in the TR.

a.1. End-to-End Tests

The purpose of the end-to-end test suite is to demonstrate that JPF-Core-X correctly exhibits the required behavior as described in the Tool Operational Requirements (TOR) document. Successful completion of each test will demonstrate that the external interface requirements and the verification operational requirements listed in the TOR are satisfied for that test case.

The complete set of test cases is designed to capture the four outcome categories which, as stated in Section e.1.2. of the TOR, must be readily apparent to the user:

- *SUT passes*, when JPF-Core-X completely explores the system state space without finding any property violations;
- *SUT failure*, when JPF-Core-X discovers one or more property violations in the LLRs;
- *JPF-Core-X error*, when either inputs or internal fault cause JPF-Core-X to fail without completing its exploration of the state space;
- *JPF-Core-X interruption*, when an external source (such as the operating system, a user, or a test automation system) terminates JPF-Core-X before it has completed its run.

To show that *SUT pass* conditions are found and reported correctly, JPF-Core-X will be run on example Java code programs that are known to have no property violations. Successful evaluation of these tests will demonstrate that JPF-Core-X meets reporting requirements for the *SUT pass* condition and, secondarily, will provide partial evidence that JPF-Core-X does not issue false negatives.

To show that *SUT failure* conditions are found and reported correctly, JPF-Core-X will be run on example Java code programs with known property violations. Successful evaluation of these tests will demonstrate that JPF-Core-X meets reporting requirements for the *SUT failure* condition and, secondarily, will provide partial evidence that JPF-Core-X does not issue false positives.

To show that *SUT error* conditions are dealt with correctly, JPF-Core-X will be run on example Java code programs with properties that will cause the search method to reach a prescribed limit on memory usage. This will cause JPF-Core-X to exit without completing its exploration of the state space. Successful evaluation of these tests will demonstrate that JPF-Core-X meets reporting requirements for the *SUT error* condition and, secondarily, will provide further partial evidence that JPF-Core-X does not issue false positives.

To show that *SUT interruption* conditions are dealt with correctly, JPF-Core-X will be run on example Java code programs in parallel with another "interrupter" process that will terminate JPF-Core-X before it has completed its run. Successful evaluation of these tests will demonstrate that JPF-Core-X meets reporting requirements for the *SUT interruption* condition and, secondarily, will provide further partial evidence that JPF-Core-X does not issue false positives.

a.2. Unit Tests

The end-to-end tests discussed in the preceding section demonstrate JPF-Core-X running from beginning to end, showing that it produces the correct results under a finite set of example test cases. The purpose of the unit tests is to verify that each method used within JPF-Core-X (in the processing of model-checking the SUT), is implemented correctly.

Each unit test verification is based upon the expectation that, for a given input, the method being tested has a corresponding correct output. Tests pass when the output (or some part or property of it) matches the expected output. The argument for correctness of the expected output is provided in the comments attached to each unit test.

a.3. Test Suite Obligations

Developing the full test suite for JPF-Core-X involves selecting a finite set of test cases, where each test case is unique from the others in one or more ways. The rationale for designing the individual test cases is based on the following set of obligations for the entire test suite:

1. Each test case in the suite should provide direct evidence to support at least one requirement

in the TOR or TR. Similarly, every requirement in the TOR and TR should be covered by at least one test case.

2. Test cases should be designed to test against state spaces of varying size and complexity, by using SUTs with varying numbers of threads.

3. Test cases should be designed to test against state spaces of varying size and complexity, by using SUTs with different underlying source code.

4. End-to-end test cases should be designed to demonstrate successful detection of all property violations covered in this qualification (deadlocks, race conditions, assertions, uncaught exceptions).

5. Additional end-to-end test cases should be designed to demonstrate that the tool does not falsely detect property violations when there are none in the SUT.

Discussion

Note that item 5 identifies an obligation to include specific tests cases that show the absence of false positives. Technically, this is *not* required for the tool to be qualified under DO-330, because it does not impact soundness. We elect to include the obligation, though, beacuse it is an important feature from a usability standpoint. A tool that reports false positives may still be qualified, but if it reports too many of them, it may no longer provide value to the verification process.

b. Test Cases

In this section, we describe the set of test cases that will be used to perform tool verification of JPF-Core-X. Test cases are organized into two main categories: 1) full end-to-end tests of JPF-Core-X, and 2) low-level unit tests of individual methods.

Discussion

When we began writing this document, we initially examined the existing JPF-Core examples and regression tests, expecting to find a sufficient set of representative test cases to seed our discussion. We found a small but useful set of examples, and have used a subset of those as candidate "end-to-end" test cases to support our JPF-Core-X verification. We also found that, while JPF-Core does include a regression test suite, it does not provide a comprehensive set of unit tests. The methods that implement the most fundamental aspects of the tool's functionality, such as *search*, *choice generation*, and *property violation detection*, do not have their own dedicated unit tests. Instead, most of the regression tests involve running the full JPF-Core tool on a test class with assertions. Rather than write our own unit tests, which is beyond the scope of the case study, we instead identify the actual methods in the source code that support each requirement, and list a corresponding name for the unit test.

b.1. End-to-End Tests

The purpose of the end-to-end test suite is to demonstrate that JPF-Core-X correctly exhibits the required behavior as described in the Tool Operational Requirements (TOR) document. Successful completion of each test will demonstrate that the external interface requirements and the verification operational requirements listed in the TOR are satisfied for that test case.

Each of the end-to-end test cases will consist of two main inputs: 1) a distinct set of Java source code representing the SUT, and 2) a corresponding *.jpf property file. For each end-to-end test case, we describe the purpose of the test, the set of inputs, the expected results to achieve the required coverage criteria, and the pass/fail criteria.

Tracing to the TOR requirements is provided in the tables below. Note that, in the TOR, requirements are organized into three general categories: 1) functional requirements, 2) requirements for abnormal activation modes, and 3) performance requirements.

Discussion

In this case study document, only a small portion of the full required set of end-to-end test cases are described. The tables below provide an organized trace of the end-to-end test cases to TOR requirements. All test cases shown in italics are placeholders, indicating that one or more test cases would be required.

Table 1: TOR Functional Requirements Trace

	Requirement Categories			
Test Case	*Inputs*	*Outputs*	*Properties*	*Errors*
Bounded Buffer 1	TOR-e..1...TOR-e..4	TOR-e..5...TOR-e..9	TOR-e..18...TOR-e..20	n/a
Bounded Buffer 2	TOR-e..1...TOR-e..4	TOR-e..5...TOR-e..9	TOR-e..18...TOR-e..20	n/a
Bounded Buffer 3	TOR-e..1...TOR-e..4	TOR-e..12	n/a	n/a
RemoteAgent	TOR-e..1...TOR-e..4	TOR-e..5...TOR-e..9	TOR-e..18...TOR-e..20	n/a
Race Cond. Tests	TOR-e..1...TOR-e..4	TOR-e..5...TOR-e..9	TOR-e..21	n/a
Assertion Tests	TOR-e..1...TOR-e..4	TOR-e..5...TOR-e..9	TOR-e..22	n/a
XML Output Tests	TOR-e..1...TOR-e..4	TOR-e..10...TOR-e..11	n/a	n/a
JPF Error Tests	TOR-e..1...TOR-e..4	n/a	n/a	TOR-e..14...TOR-e..15
Interruption Tests	TOR-e..1...TOR-e..4	n/a	n/a	TOR-e..16...TOR-e..17

Table 2: TOR Abnormal Activation Requirements Trace

	Requirement Categories			
Test Case	*Inputs*	*Outputs*	*Properties*	*Errors*
Syntax Error Test	TOR-e..1...TOR-e..4	n/a	n/a	TOR-f..1
Missing Listener Test	TOR-e..1...TOR-e..4	n/a	n/a	TOR-f..2
Java Class Error Test	TOR-e..1...TOR-e..4	n/a	n/a	TOR-f..3
Non-Standard Listener Test	TOR-e..1...TOR-e..4	n/a	n/a	TOR-f..4
Thread Error Test	TOR-e..1...TOR-e..4	n/a	n/a	TOR-f..5

Table 3: TOR Performance Requirements Trace

	Requirement Categories			
Test Case	*Inputs*	*Outputs*	*Properties*	*Errors*
Normal Op. Tests	TOR-e..1...TOR-e..4	TOR-i..1...TOR-i..6	n/a	n/a
Depth Limit Tests	TOR-e..1...TOR-e..4	TOR-i..7...TOR-i..8	n/a	n/a
Memory Limit Tests	TOR-e..1...TOR-e..4	TOR-i..9	n/a	n/a
Abnormal Op. Tests	TOR-e..1...TOR-e..4	TOR-i..10...TOR-i..12	n/a	n/a

b.1.1. Deadlock - Bounded Buffer 1

Purpose The purpose of this test case is to verify that JPF-Core-X properly detects the presence of a deadlock that is known to exist in the SUT.

The SUT includes a bounded buffer of size 1, 2 producer threads and 1 consumer thread. The bounded buffer consists of a fixed number of slots; in this case the buffer size is 1. Producer threads put an object into the buffer, and consumer threads remove an object from the buffer. The deadlock depends on a notification choice between a consumer and a producer in a context where only threads of the notifier type are still runnable. A deadlock results if the number of producer or consumer threads is at least 2x the size of the buffer, which is true for this test case.

This particular test is adapted from "Concurrency: State Models & Java Programs" [2].

Inputs

- SUT: jpf-core/src/examples/BoundedBuffer.java
- JPF Properties: jpf-core/src/examples/BoundedBuffer1.jpf

The JPF properties file for this test specifies "BoundedBuffer" as the target, and defines the following target arguments: `target_args = 1,2,1`. This specifies buffer of size 1, 2 producer threads, and 1 consumer thread.

Expected Results In this test, we expect a deadlock to be encountered. The printout to the console should include an error message indicating the occurrence of a deadlock, a snapshot showing the current status of each thread when the deadlock is found, and a summary results message.

The expected error message is:

```
1 ====================================================== error #1
2 gov.nasa.jpf.jvm.NotDeadlockedProperty
3 deadlock encountered:
4   thread BoundedBuffer$Producer:{id:1,name:P1,status:WAITING,priority:5,lockCount
      :1,suspendCount:0}
5   thread BoundedBuffer$Producer:{id:2,name:P2,status:WAITING,priority:5,lockCount
      :1,suspendCount:0}
6   thread BoundedBuffer$Consumer:{id:3,name:C1,status:WAITING,priority:5,lockCount
      :1,suspendCount:0}
```

The expected snapshot message is:

```
1 ====================================================== snapshot #1
2 thread BoundedBuffer$Producer:{id:1,name:P1,status:WAITING,priority:5,lockCount:1,
      suspendCount:0}
3   waiting on: BoundedBuffer@146
```

```
  call stack:
  at java.lang.Object.wait(Object.java)
  at BoundedBuffer.put(BoundedBuffer.java:55)
  at BoundedBuffer$Producer.run(BoundedBuffer.java:91)

thread BoundedBuffer$Producer:{id:2,name:P2,status:WAITING,priority:5,lockCount:1,
    suspendCount:0}
  waiting on: BoundedBuffer@146
  call stack:
  at java.lang.Object.wait(Object.java)
  at BoundedBuffer.put(BoundedBuffer.java:55)
  at BoundedBuffer$Producer.run(BoundedBuffer.java:91)

thread BoundedBuffer$Consumer:{id:3,name:C1,status:WAITING,priority:5,lockCount:1,
    suspendCount:0}
  waiting on: BoundedBuffer@146
  call stack:
  at java.lang.Object.wait(Object.java)
  at BoundedBuffer.get(BoundedBuffer.java:66)
  at BoundedBuffer$Consumer.run(BoundedBuffer.java:110)
```

The expected results message is:

```
====================================================== results
error #1: gov.nasa.jpf.jvm.NotDeadlockedProperty "deadlock encountered:   thread
    BoundedBuffer$Produ..."
```

Pass/Fail Criteria For this test to pass, all of the following must be true:

- The error message shows: "deadlock encountered:".
- The error message lists all Producer threads and the Consumer thread. (Note that they may be listed in any order.)
- The snapshot message displays the status of each thread.
- The status of each thread in the snapshot message displays "waiting on:", with another thread name listed after the colon.
- The results message shows "error".
- The results message shows "error #1: gov.nasa.jpf.jvm.NotDeadlockedProperty deadlock encountered: thread" followed by the name of any thread.

If any of the above conditions are not met, the test fails.

b.1.2. Deadlock - Bounded Buffer 2

This test case is based on the case in Section b.1.1., except that here the buffer size is 2, and the number of producer threads is 4.

Discussion

This test case involves a SUT with twice as many threads as the previous test case. In every other way, the SUTs in these two test cases are identical. Therefore, the only rationale for including both tests (as opposed to just one) is to satisfy test suite obligation #2, using SUTs with varying numbers of threads to test against state spaces of varying size and complexity.

Inputs

- SUT: jpf-core/src/examples/BoundedBuffer.java
- JPF Properties: jpf-core/src/examples/BoundedBuffer2.jpf

The JPF properties file for this test specifies "BoundedBuffer" as the target, and defines the following target arguments: `target_args = 2,4,1`. This specifies a buffer of size 2, 4 producer threads, and 1 consumer thread.

Expected Results In this test, we expect a deadlock to be encountered. The printout to the console should include an error message indicating the occurrence of a deadlock, a snapshot showing the current status of each thread when the deadlock is found, and a summary results message.

The expected error message is:

```
====================================================== error #1
gov.nasa.jpf.jvm.NotDeadlockedProperty
deadlock encountered:
  thread BoundedBuffer$Producer:{id:1,name:P1,status:WAITING,priority:5,lockCount
    :1,suspendCount:0}
  thread BoundedBuffer$Producer:{id:2,name:P2,status:WAITING,priority:5,lockCount
    :1,suspendCount:0}
  thread BoundedBuffer$Producer:{id:3,name:P3,status:WAITING,priority:5,lockCount
    :1,suspendCount:0}
  thread BoundedBuffer$Producer:{id:4,name:P4,status:WAITING,priority:5,lockCount
    :1,suspendCount:0}
  thread
  BoundedBuffer$Consumer:{id:5,name:C1,status:WAITING,priority:5,lockCount:1,
    suspendCount:0}
```

The expected snapshot message is:

```
====================================================== snapshot #1
thread BoundedBuffer$Producer:{id:1,name:P1,status:WAITING,priority:5,lockCount:1,
    suspendCount:0}
  waiting on: BoundedBuffer@146
  call stack:
  at java.lang.Object.wait(Object.java)
  at BoundedBuffer.put(BoundedBuffer.java:55)
  at BoundedBuffer$Producer.run(BoundedBuffer.java:91)

thread BoundedBuffer$Producer:{id:2,name:P2,status:WAITING,priority:5,lockCount:1,
    suspendCount:0}
  waiting on: BoundedBuffer@146
  call stack:
  at java.lang.Object.wait(Object.java)
  at BoundedBuffer.put(BoundedBuffer.java:55)
  at BoundedBuffer$Producer.run(BoundedBuffer.java:91)

thread BoundedBuffer$Producer:{id:3,name:P3,status:WAITING,priority:5,lockCount:1,
    suspendCount:0}
  waiting on: BoundedBuffer@146
  call stack:
  at java.lang.Object.wait(Object.java)
  at BoundedBuffer.put(BoundedBuffer.java:55)
  at BoundedBuffer$Producer.run(BoundedBuffer.java:91)

thread BoundedBuffer$Producer:{id:4,name:P4,status:WAITING,priority:5,lockCount:1,
    suspendCount:0}
  waiting on: BoundedBuffer@146
  call stack:
  at java.lang.Object.wait(Object.java)
  at BoundedBuffer.put(BoundedBuffer.java:55)
  at BoundedBuffer$Producer.run(BoundedBuffer.java:91)

thread BoundedBuffer$Consumer:{id:5,name:C1,status:WAITING,priority:5,lockCount:1,
    suspendCount:0}
  waiting on: BoundedBuffer@146
  call stack:
  at java.lang.Object.wait(Object.java)
  at BoundedBuffer.get(BoundedBuffer.java:66)
  at BoundedBuffer$Consumer.run(BoundedBuffer.java:110)
```

The expected results message is identical to that of Test b.1.1..

```
====================================================== results
error #1: gov.nasa.jpf.jvm.NotDeadlockedProperty "deadlock encountered:   thread
    BoundedBuffer$Produ..."
```

Pass/Fail Criteria The same pass/fail criteria for Test b.1.1. are used.

b.1.3. NO Deadlock - Bounded Buffer 3

This test case is intended to show JPF-Core-X running to completion without finding a property violation. This test case is based on the bounded buffer cases above. However, in this case, we use a smaller number of producer threads so that a deadlock cannot occur.

Discussion

This test case is designed to support test suite obligation #5, that end-to-end test cases should demonstrate the tool does not falsely detect property violations.

Inputs

- SUT: jpf-core/src/examples/BoundedBuffer.java
- JPF Properties: jpf-core/src/examples/BoundedBuffer3.jpf

The JPF properties file for this test specifies "BoundedBuffer" as the target, and defines the following target arguments: `target_args = 2,3,1`. This specifies a buffer of size 2, 3 producer threads, and 1 consumer thread.

For this SUT to result in a deadlock, the number of producer threads must be greater than or equal to the size of the buffer. With a buffer size of 2 and only 3 producer threads, a deadlock cannot occur.

Expected Results This test should run to completion without finding any property violations.

Pass/Fail Criteria This test passes if the following results message is printed to the console:

```
1 ====================================================== results
2 no errors detected
```

b.1.4. Deadlock - Remote Agent

Discussion

This is another example of a deadlock. It is therefore similar in kind to the first two test cases, but in this case we use a completely different SUT. In general, it is desired to establish tests that operate on a diverse set of source code, as established in test suite obligation #3.

In this test case, a deadlock is caused as a result of a missed signal, where the `wait()` operation happens after the corresponding `notify()`. This occurs because of a violated monitor encapsulation. Specifically, the SUT directly accesses monitor internal data (`'Event.count'`) from concurrent clients. The requested operation of (`'FirstTask', 'SecondTask'`) is performed without synchronization with the corresponding monitor operations (`'wait_for-Event()' and 'signalEvent()'`).

Discussion

This problem is typical for unsafe optimizations, where local caches are used as an attempt to avoid expensive blocking calls. This particular example was inspired by an actual defect found in the "Remote Agent" spacecraft controller that flew on board of Deep Space 1 [1].

Inputs

- SUT: jpf-core/src/examples/oldclassic.java
- JPF Properties: jpf-core/src/examples/oldclassic-da.jpf

Expected Results The expected error message is:

```
1 ======================================================== error #1
2 gov.nasa.jpf.jvm.NotDeadlockedProperty
3 deadlock encountered:
4   thread FirstTask:{id:1,name:Thread-1,status:WAITING,priority:5,lockCount:1,
      suspendCount:0}
5   thread SecondTask:{id:2,name:Thread-2,status:WAITING,priority:5,lockCount:1,
      suspendCount:0}
6
```

The expected results message is:

```
1 ======================================================== results
2 error #1: gov.nasa.jpf.jvm.NotDeadlockedProperty "deadlock encountered:   thread
    FirstTask:{id:1,nam..."
```

Pass/Fail Criteria The test passes if JPF-Core-X prints out both error and results messages that match what the expected results (shown above).

b.1.5. Race Condition - Racer

Discussion

For this study, we focus our test case examples only on deadlocks. However, in a real verification effort, additional end-to-end test cases should be provided to include cases where other property violations (e.g. race conditions, assertions and exceptions) are reported when present. Similarly, more cases would be required to show that JPF-Core-X runs to completion without reporting these violations for SUT's that are known to be free of error.

b.1.6. Assertion

Discussion

Test cases to demonstrate the proper detection and reporting of "assertion" property violations would be listed here.

b.1.7. Uncaught Exception

Discussion

Test cases to demonstrate the proper detection and reporting of "uncaught exception" property violations would be listed here.

b.2. Unit Tests

The purpose of the unit test suite is to verify that all of the component methods used within JPF-Core-X are implemented correctly. This is established by directly running each method with a prescribed set of inputs and comparing the result to a corresponding set of expected (and demonstrably correct) outputs.

In order to provide a clear mapping to the requirements established in the TR document, the unit tests are organized according to the main steps of the execution process of JPF-Core-X, as described in the TR. That execution process is repeated here for convenience.

1. Initialization.
   (a) Load the JPF Properties file.
   (b) Initialize all listener and reporter objects.
   (c) Initialize the run-time object.

(d) Initialize the virtual machine object.

2. Search.

   (a) Initialize the search method.

   (b) Backtrack step.

   (c) Forward step.

   (d) Check property violations.

   (e) Check search limits.

   (f) Terminate.

3. Reporting.

   (a) Report "Out of Memory" errors.

   (b) Report property violations.

4. Exit.

b.2.1. Structure of Unit Tests

Discussion

The content of this section is adapted from the JPF user's manual online at: http://babelfish.arc.nasa.gov/trac/jpf/wiki/devel/jpf_tests.

Each unit test consists of a test driver that is executed under JUnit. It starts JPF-Core-X from within its `Test` annotated methods. In addition, each test includes a class that is executed by JPF-Core-X in order to check the verification goals.

The `main()` method of `TestJPF` derived classes are uniform throughout, with the following structure:

```
1 public static void main(String[] testMethods){
2   runTestsOfThisClass(testMethods);
3 }
```

Test classes consist of `Test` annotated JUnit test methods, which all share the following structure:

```
1 import org.junit.Test;
2
3 @Test public void testX () {
4   if (verifyNoPropertyViolation(JPF_ARGS){
5     .. code to verify by JPF
```

```
  }
}
```

The host VM checks the results of the JPF-Core-X run, and accordingly throws an AssertionError in case it does not correspond to the expected result. The most common goals are:

`verifyNoPropertyViolation` - JPF-Core-X is not supposed to find an error

`verifyPropertyViolation` - JPF-Core-X is supposed to find the specified property violation

`verifyUnhandledException` - JPF-Core-X is supposed to detect an unhandled exception of the specified type

`verifyAssertionError` - same for AssertionErrors

`verifyDeadlock` - JPF-Core-X is supposed to find a deadlock

Each of these methods delegate running JPF-Core-X to a corresponding method whose name does not start with 'verify..'. These workhorse methods expect explicit specification of the arguments (including SUT main class name and method names), but they return JPF-Core-X objects, and therefore can be used for more sophisticated inspection (e.g. to find out about the number of states).

TestJPF also provides the following methods that can be used within test methods to find out which environment the code is executed from:

`isJPFRun()` - returns true if the code is executed under JPF-Core-X

`isJUnitRun()` - returns true if the code is executed under JUnit by the host VM

`isRunTestRun()` - returns true if the code is executed by RunTest.jar

A typical test method is shown in the example below.

```
  @Test public void testIntFieldPerturbation() {

    if (!isJPFRun()){ // run this outside of JPF
      Verify.resetCounter(0);
    }

    if (verifyNoPropertyViolation("+listener=.listener.Perturbator",
                                  "+perturb.fields=data",
                                  "+perturb.data.class=.perturb.IntOverUnder",...
                                  "+perturb.data.delta=1")){
      // run this under JPF
      System.out.println("instance field perturbation test");

```

```
14        int d = data; // this should be perturbed
15        System.out.println("d = " + d);
16
17        Verify.incrementCounter(0);
18
19      } else { // run this outside of JPF
20        assert Verify.getCounter(0) == 3;
21      }
22    }
```

b.2.2. Unit Tests for Initialization

Discussion

In a complete test cases and procedures document, the unit tests that support the "Initialization" step would be provided here. For our study, we instead focus on a representative subset of the unit tests for the "Search" step. These are discussed in the next subsection.

b.2.3. Unit Tests for Search

Here, we list the unit tests that support the Search step of the JPF-Core-X execution process. All methods are shown in the format: `Class.method()`. If no class is shown, the method belongs to the DFSearch class. In some cases, we use $< listener >$ (or $< reporter >$) for the class name, indicating that the listed method of all listener (or reporter) objects is tested.

Discussion

For this case study, we only list the minimum set of tests required to directly verify each of the search-related requirements that are described in Section d of the Tool Requirements (TR) document. It is important to recognize, however, that many of these individual requirements correspond to multiple nested methods in the JPF-Core-X source code. For a complete verification, all methods that are needed to support a tool requirement must be tested to verify their correct implementation.

It is also important to note that requirements coverage and code coverage are complementary measures. If a set of tests covers all TRs without exercising all of the methods, then the TRs must be incomplete. In other words, additional TRs would have to be added so that the coverage of those new TRs would necessarily exercise all of the (previously unexercised) methods.

Tests covering requirements for the "Initialize" step of the search method are listed below.

Initialize Search Method		
Unit Test	*Method(s) Tested*	*Requirement(s)*
DFSearchInitTest	search()	TR-d..1.2

Tests covering requirements for the "Backtrack" step of the search method are listed below.

Backtrack Step		
Unit Test	*Method(s) Tested*	*Requirement(s)*
BacktrackRequestTest	checkAndResetBacktrackRequest() requestBacktrack() search()	TR-d..1.3.1
IsNewStateTest	JVM.isNewState()	TR-d..1.3.2
IsEndStateTest	JVM.isEndState()	TR-d..1.3.3
IsIgnoredStateTest	JVM.isIgnoredState()	TR-d..1.3.4
BacktrackTest	DefaultBacktracker.backtrack()	TR-d..1.3.5
BacktrackSuccessTest	notifyStateBacktracked() < *listener* >.stateBacktracked() < *reporter* >.stateBacktracked()	TR-d..1.3.6
BacktrackTerminateTest	DefaultBacktracker.backtrack()	TR-d..1.3.7

Tests covering requirements for the "Forward" step of the search method are listed below.

Forward Step		
Unit Test	*Method(s) Tested*	*Requirement(s)*
ForwardTest	JVM.forward()	TR-d..1.4
InitNextTransTest()	SystemState.initializeNextTransition()	(not provided in sample TR)
PushKernelStateTest()	DefaultBackTracker.pushKernelState()	(not provided in sample TR)
PathCacheTest()	SystemState.getLast()	(not provided in sample TR)
ExecuteNextTransTest()	SystemState.executeNextTransition()	(not provided in sample TR)
PushSystemStateTest()	DefaultBackTracker.pushSystemState()	(not provided in sample TR)
UpdatePathTest()	JVM.updatePath()	(not provided in sample TR)
GCTest()	KernelState.gc()	(not provided in sample TR)
ForwardFailureTest()	notifyStateProcessed()	TR-d..1.4.1
ForwardSuccessTest()	notifyStateAdvanced()	TR-d..1.4.2

Discussion

"Choice generation" refers to the process by which the next state in the search is selected. This is an important component of the overall JPF design, and involves multiple different methods implemented across several classes. In a complete TR, a full set of requirements would be written to ensure that all aspects of choice generation are implemented correctly.

Additional tests covering requirements associated with "Choice Generation" are listed below. Choice generation is used within the SystemState.initializeNextTransition() method.

Choice Generation		
Unit Test	*Method(s) Tested*	*Requirement(s)*
NotifyCGSetTest	SystemState.notifyChoiceGeneratorSet() JVM.notifyChoiceGeneratorSet()	(not provided in sample TR)
GetCascadedParentTest	ChoiceGenerator.getCascadedParent()	(not provided in sample TR)
AdvanceCurCGTest	SystemState.advanceCurCg()	(not provided in sample TR)

Tests covering requirements for the "Check Property Violations" step of the search method are listed below.

Check Property Violations		
Unit Test	*Method(s) Tested*	*Requirement(s)*
CheckPropertyViolationTest	checkPropertyViolation()	TR-d..1.5
NotifyPropertyViolatedTest	notifyPropertyViolated()	TR-d..1.5.1
NotDeadlockedPropertyTest	NotDeadlockedProperty().check()	TR-d..1.5.2
PreciseRaceDetectorTest	PreciseRaceDetectorTest().check()	TR-d..1.5.3
AssertionPropertyTest	AssertionProperty().check()	TR-d..1.5.4
NoUncaughtExcptionsTest	NoUncaughtExcptions().check()	TR-d..1.5.5

Tests covering requirements for the "Check Search Limits" step of the search method are listed below.

Check Search Limits		
Unit Test	*Method(s) Tested*	*Requirement(s)*
NotifySearchConstraintTest	notifySearchConstraint() < *listener* >.searchConstraintHit() < *reporter* >.searchConstraintHit()	TR-d..1.6.1
CheckStateSpaceLimitTest	checkStateSpaceLimit()	TR-d..1.6.2

b.2.4. Unit Tests for Reporting

Discussion

In a complete test cases and procedures document, the unit tests that support the "Reporting" step would be provided here. As mentioned earlier, however, for our study, we instead focus on a subset of the unit tests for the "Search" step. These are discussed in the previous subsection.

c. Test Procedures

In this section, we describe the procedures that will be used to verify the test cases that were enumerated in the previous section. For each of our two test categories, we describe the step-by-step instructions for how the test case is to be set up and executed, how the test results will be evaluated, and the test environment to be used.

c.1. End-to-End Tests

All end-to-end tests will follow a common procedure. The steps below assume that JPF-Core-X has already been installed and loaded as a project into the NetBeans IDE.

1. Open the NetBeans IDE.
2. In the "Projects" view, expand the "src/examples" folder.
3. Find the *.jpf properties file for the desired end-to-end test.
4. Right-click (or control-click) on the selected *.jpf properties file, and click on "Verify...".
5. Visually inspect the JPF report information that is printed to the NetBeans console.
6. Compare the printed results to the expected results, according to the pass/fail criteria for that test.

Each end-to-end test also produces an output log that contains the report printouts. These logs can be used, along with a formal representation of the expected results and pass/fail criteria, to perform automated testing of all end-to-end tests. However, such automated testing is beyond the scope of this verification effort.

Discussion

The end-to-end tests described here are meant to be run one at a time, with a human user starting the tool and inspecting the output, just as they would do when using the tool under normal circumstances. The final step is to compare the printed output to expected results according to the pass/fail criteria for each test. While some amount of human-in-the-loop testing is important to verify usability and correct understanding of the tool results, it is impractical to fully rely on this approach for large numbers of tests and/or tests that require very long run times.

This need for test automation is not specific to formal methods tools, though; it applies to all types of verification tools. Both the running of the end-to-end tests and the evaluation of their results can be automated, which would enable a large number of teststo be run in succession. It is important to note, however, that careful analysis and review of the test harness itself should be made to ensure its correctness.

c.2. Unit Tests

All unit tests are implemented using the JUnit test framework. Execution of all unit tests will follow a common procedure. The steps below assume that JPF-Core-X has already been installed and loaded as a project into the NetBeans IDE.

Running Individual Tests JPF-Core-X is configured to run the full test suite either through the IDE, or via the command line. To run through the IDE:

1. Open the NetBeans IDE.
2. Expand "src/tests" under JPF-Core-X in the Projects view.
3. Select the test to be run.
4. Right-click (or control-click) on the selected test and choose "Run".
5. Review test results printed to the NetBeans console.

Alternatively, to run via the command line:

1. Open a terminal window.
2. Change directories to the JPF-Core-X root directory.
3. Enter the following command:
   `bin/test gov.nasa.jpf.test.<CONTAINER>.<TESTFILE>`
4. Review test results printed to the terminal window.

If the unit test passes, the final line printed will be:

```
.....................................  tests:  1, failures:  0, errors:  0
```

If a unit test does not pass, then either "failures" or "errors" will be assigned a value of 1 (but not both). A "failure" means the test ran to completion, the result was evaluated, but it did not meet the pass/fail criteria. An "error" means that one of several possible errors occurred either during the execution of the code being tested, or during execution of the test harness. The test framework of JPF-Core-X provides descriptive error messages for any failures or errors that are encountered, along with a backtrace of the stack.

Running the Full Test Suite JPF-Core-X is configured to run the full test suite only through the IDE:

1. Open the NetBeans IDE
2. Right-click (or control-click) on the JPF-Core-X project and select "Test".

Results from each test will be printed out to the console. Once all tests have completed, the following will be printed:

..... `tests:` T`, failures:` F`, errors:` E

where actual numbers would be printed in place of the T, E, F symbols. The full unit test suite passes when $E = F = 0$.

References

[1] K. Havelund, M. Lowry, S. Park, C. Pecheur, J. Penix, W. Visser, and J. L. White, "Formal Analysis of the Remote Agent Before and After Flight," in *Proc. of the 5th NASA Langley Formal Methods Workshop*, June 2000.

[2] J. Magee and J. Kramer, *Concurrency: State Models and Java Programs*, Wiley, 2006.