Java PathFinder: a tool to detect bugs in Java code

Franck van Breugel

February 18, 2017

©2016 Franck van Breugel

Abstract

It is well known that software contains bugs. Since Java is among the most popular programming languages, it is essential to have tools that can detect bugs in Java code. Although testing is the most used technique to detect bugs, it has its limitations, especially for nondeterministic code. Concurrency and randomization are the two main sources of nondeterminism. To find bugs in nondeterministic code, testing needs to be complemented with other techniques such as model checking. Java PathFinder (JPF) is the most popular model checker for Java code. In this book, we describe how to install, configure, run and extend JPF.

Preface

According to a 2002 study commissioned by the US Department of Commerce's National Institute of Standards and Technology, "estimates of the economic costs of faulty software in the US range in the tens of billions of dollars per year and have been estimated to represent approximately just under one percent of the nation's gross domestic product." Since software development has not changed drastically in the last decade, but the footprint of software in our society has increased considerably, it seems reasonable to assume that this number has increased as well and ranges in the hundreds of billions of dollars per year on a world wide scale. This was confirmed by a recent study [BJC$^+$13] which included that "wages-only estimated cost of debugging is US $312 billion per year." Hence, *tools to detect bugs* in software can impact the software industry and even the world economy. The topic of this book is such a tool.

The TIOBE programming community index[1], the transparent language popularity index[2], the popularity of programming language index[3], the RedMonk programming language rankings[4], and Trendy Skills[5], all rank *Java* among the most popular programming languages. Popularity of the language and impact of a tool to detect bugs of software written in that language go hand in hand. Therefore, we focus on a popular language in this book, namely Java.

Testing is the most commonly used method to detect bugs. However, for *nondeterministic* code testing may be less effective. Code is called nondeterministic if it gives rise to different executions even when all input to the code is fixed. Randomization and concurrency both give rise to nondeterminism. Since concurrency is generally considered more intricate than randomization, our examples will predominantly focus on the latter. Chapter **??** will concentrate on concurrency. To illustrate the limitations of testing when it comes to nondeterministic code, consider the following Java application.

```
1  import java.util.Random;
2
3  public class Example {
4    public static void main(String[] args) {
5      Random random = new Random();
6      System.out.print(random.nextInt(10));
7    }
8  }
```

The above application may result in ten different executions, since it prints a randomly chosen integer in the interval $[0, 9]$. Now, let us replace line 6 with

```
    System.out.print(1 / random.nextInt(9));
```

In 80% of the cases, the application prints zero, in 10% it prints one, and in the remaining 10% it crashes because of an uncaught exception due to a division by zero. Of course, it may take more than ten executions before we encounter the exception. In case we choose an integer in the interval $[0, 999, 999]$ it may take many executions before encountering

[1] www.tiobe.com
[2] lang-index.sourceforge.net
[3] http://pypl.github.io/PYPL.html
[4] redmonk.com/sogrady/2013/02/28/language-rankings-1-13
[5] trendyskills.com

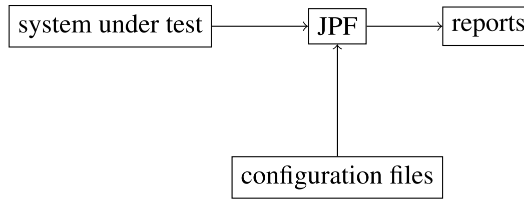


Figure 1: Overview of JPF.

the exception. If we execute the application one million times, there is still a 37% chance that we do not encounter the exception.[6]

In the presence of nondeterminism, testing does not guarantee that all different executions are checked. Furthermore, if a test detects a bug in nondeterministic code, it may be difficult to reproduce. Therefore, in that case methods which complement testing are needed. *Model checking* is such an alternative. It aims to check all potential executions of nondeterministic code in a systematic way.

We will not discuss model checking in much detail. Instead, we refer the interested reader to textbooks such as [BK08], [CGP01] and [BBF$^+$01]. In this book, we introduce the reader to a model checker, a tool that implements model checking. In particular, we focus on a model checker for Java.

Although there are several model checkers for Java, including Bandera [CDH$^+$00] and Bogor [RDH03] to name a few, *Java PathFinder* (JPF) is the most popular one. Its popularity is reflected by several statistics. For example, the conference paper [VHBP00] and its extended journal version [VHB$^+$03] have been cited more than 1200 times according to Google scholar, making it the most cited work on a Java model checker. In this book, we focus on JPF. Although JPF can do much more than detect bugs, we concentrate on that functionality.

Overview of JPF

In Figure 1 we provide a high level overview of JPF. It takes as input a system under test and configuration files and produces reports as output. The *system under test* is the application, a Java class with a `main` method, we want to check for bugs. JPF not only checks that `main` method but also all other code that is used by that `main` method. JPF can only check a closed system. That is, a system for which all input is provided, be it obtained from the keyboard, the mouse, a file, a URL, etcetera. Handling such input can be far from trivial and we will come back to this in Chapter **??**.

JPF can be configured in two different ways: by command line arguments or in configuration files. We will concentrate on the second option. There are three different types of configuration file. We will discuss them in Chapter 3.

The reports that JPF produces can take different forms. For example, a report can be written to the console or to a file, and it can be text or XML. In the configuration files one can specify what type of reports should be produced by JPF. We will discuss this in more detail in Chapter **??**.

Overview of the Book

This book has been written for both students and developers who are interested in tools that can help them with detecting bugs in their Java code. In Chapter 1 we discuss how to install JPF. How to run JPF is the topic of Chapter 2.

[6] The probability of choosing zero is $\frac{1}{1{,}000{,}000}$. The probability of not choosing zero is $1 - \frac{1}{1{,}000{,}000} = \frac{999{,}999}{1{,}000{,}000}$. The probability of not choosing zero one million times in a row is $(\frac{999{,}999}{1{,}000{,}000})^{1{,}000{,}000} \approx 0.37$.

Chapter 1

Installing JPF

As we have already discussed in the preface, JPF is a tool to detect bugs in Java code. Since the reader is interested in JPF, we feel that it is safe to assume that the reader is familiar with Java and has installed the Java development kit (JDK). The JDK should be at least version 8.

JPF can be installed in several different ways on a variey of operating systems. A road map for Section 1.2–1.10 can be found in Figure 1.1. Since changes are made to JPF on a regular basis, it is best to obtain its souces from JPF's Mercurial repository. Mercurial is a version control system. Information about Mercurial, including instructions how to install Mercurial, can be found at www.mercurial-scm.org. We describe three different ways to install (and update) the sources of JPF's Mercurial repository: au naturel, within Eclipse, and within NetBeans, in Section 1.1 (and 1.2), 1.3 (and 1.4), and 1.6 (and 1.7), respectively. For those using Eclipse or NetBeans, the latter two options are more convenient. Also, there are JPF plugins for Eclipse or NetBeans. How to install those is discussed in Section 1.5 and 1.8, respectively. How to use these plugins to run JPF is discussed in Chapter 2.

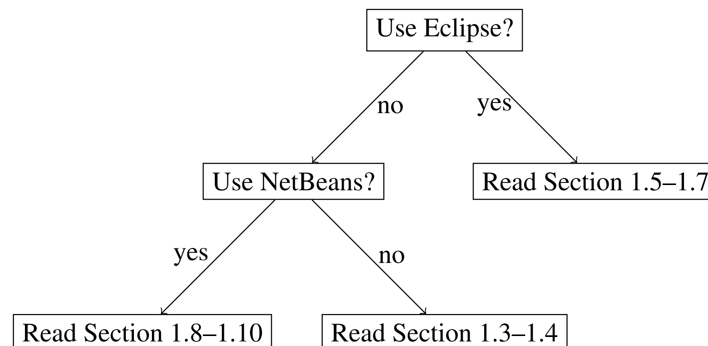


Figure 1.1: Road map for Section 1.2–1.10.

As we already mentioned in the preface, JPF is easily extensible. Therefore, it should come as no surprise that there are numerous extensions of JPF. In Section 1.9 we will discuss how to install such an extension.

1.1 Installing Sources with Mercurial

How to install Mercurial is beyond the scope of this book. We refer the reader to www.mercurial-scm.org. We assume that the path to the `hg` command is already part of the system environment variable `PATH` (see Section 1.1.4). To install the JPF sources with Mercurial, follow the seven steps below.

1. Create a directory named `jpf`.

2. To get the JPF sources with Mercurial, open a shell (Linux or OS X) or command prompt (Windows), go to the `jpf` directory and type

   ```
   hg clone http://babelfish.arc.nasa.gov/hg/jpf/jpf-core
   ```

   This results in output similar to the following.

   ```
   destination directory: jpf-core
   requesting all changes
   adding changesets
   adding manifests
   adding file changes
   added 1057 changesets with 9561 changes to 2504 files (+2 heads)
   updating to branch default
   932 files updated, 0 files merged, 0 files removed, 0 files unresolved
   ```

3. Set the user environment variable `JAVA_HOME` as described in Section 1.1.1.

4. Install JUnit as described in Section 1.1.2.

5. Run ant as described in Section 1.1.3.

6. Set the user environment variable `JPF_HOME` to the path of `jpf-core`. For example, if the `jpf` directory, created in step 1, has path `/cs/home/franck/projects/jpf`, then the path of `jpf-core` is `/cs/home/franck/projects/jpf/jpf-core`. Similarly, if the `jpf` directory has path `C:\Users\franck\projects\jpf`, then the path of `jpf-core` is `C:\Users\franck\projects\jpf\jpf-core`.

7. Add the path of the `jpf` command to the system environment variable `PATH` as described in Section 1.1.4.

8. Create the `site.properties` file as described in Section 1.1.5.

Once the above steps have been successfully completed, the reader can move on to Chapter 2 and run JPF.

1.1.1 Setting User Environment Variable `JAVA_HOME`

Linux

1. Locate the directory of the JDK. Unless the install path for the JDK was changed during installation, it will be a subdirectory of `/usr/java`. Inside that directory will be one or more subdirectories whose name starts with `jdk`, such as `jdk1.8.0_51`. Choose the latest version. For example, if the directory contains both `jdk1.6.0_37` and `jdk1.8.0_51`, then the JDK install path is `/usr/java/jdk1.8.0_51`.

2. Set the user environment variable named `JAVA_HOME` to the directory of the JDK by using the `set` or `setenv` command in a startup script. For more details, do a web search for how to set an environment variable in Linux.

Windows

1. Locate the directory of the JDK. Unless the install path for the JDK was changed during installation, it will be a subdirectory of `C:\Program Files\Java`. Inside that directory will be one or more subdirectories whose name starts with `jdk`, such as `jdk1.8.0_51`. Choose the latest version. For example, if the directory contains both `jdk1.6.0_37` and `jdk1.8.0_51`, then the JDK install path is `C:\Program Files\Java\jdk1.8.0_51`.

2. Set the user environment variable named `JAVA_HOME` to the directory of the JDK. For more details, do a web search for how to set an environment variable in Windows.

OS X

1. Locate the directory of the JDK. Unless the install path for the JDK was changed during installation, it will be a subdirectory of `/Library/Java/JavaVirtualMachines`. Inside that directory will be one or more subdirectories whose name ends with `jdk`, such as `jdk1.8.0_06.jdk`. Choose the latest version. For example, if the directory contains both `jdk1.7.0_60.jdk` and `jdk1.8.0_06.jdk`, then the JDK install path is `/Library/Java/JavaVirtualMachines/jdk1.8.0_06.jdk/Contents/Home`.

2. Set the user environment variable named `JAVA_HOME` to the directory of the JDK. For more details, do a web search for how to set an environment variable in OS X.

1.1.2 Installing JUnit

1. Create a directory named `junit`.

2. Download the latest versions of `junit.jar` and `hamcrest-core.jar` from junit.org.

3. Set the user environment variable `JUNIT_HOME` to the directory that contains `junit.jar` and `hamcrest-core.jar` (see Section 1.1.1).

1.1.3 Running Ant

Ant is a Java library and command-line tool that can be used to compile the JPF sources, test them, generate jar files, etcetera. For more information about ant, we refer the reader to ant.apache.org. We assume that the reader has already installed ant (see ant.apache.org for installation instructions) and has added its directory to the `PATH` environment variable (see Section 1.1.4).

Linux and OS X

In a shell, go to the subdirectory `jpf-core` of the created directory `jpf`. The directory `jpf-core` contains the file `build.xml`. To run ant, type `ant test`. This results in a lot of output, the beginning and end of which are similar to the following.

```
Buildfile: /cs/home/franck/projects/jpf/jpf-core/build.xml

-cond-clean:

clean:

-init:
[mkdir] Created dir: /cs/home/franck/projects/jpf/jpf-core/build

...

[junit] Running gov.nasa.jpf.util.script.ScriptEnvironmentTest
[junit] Tests run: 3, Failures: 0, Errors: 0, Time elapsed: 0.04 sec

BUILD SUCCESSFUL
Total time: 3 minutes 31 seconds
```

Windows

In a command prompt, go to the subdirectory `jpf-core` of the created directory `jpf`. The directory `jpf-core` contains the file `build.xml`. To run ant, type `ant test`. This results in a lot of output, the beginning and end of which are similar to the following.

```
Buildfile: C:\Users\franck\projects\jpf\jpf-core\build.xml

-cond clean:

clean:

-init:
[mkdir] Created dir: C:\Users\franck\projects\jpf\jpf-core\build

...

[junit] Running gov.nasa.jpf.util.script.ScriptEnvironmentTest
[junit] Tests run: 3, Failures: 0, Errors: 0, Time elapsed: 0.04 sec

BUILD SUCCESSFUL
Total time: 3 minutes 31 seconds
```

1.1.4 Adding to the System Environment Variable `PATH`

The system environment variable `PATH` consists of a list of directories in which programs are located. Below, we discuss how to add the directory containing the `jpf` program can be added to `PATH`.

Linux

Add to the system environment variable named `PATH` the directory of the `jpf` program by changing the `set` or `setenv` command for `PATH` in a startup script. If the `jpf` directory has path `/cs/home/franck/projects/jpf`, then add `/cs/home/franck/projects/jpf/jpf-core/bin` to the system environment variable `PATH`. For more details, do a web search for how to change an environment variable in Linux.

Windows

In Windows, environment variables are not case sensitive. Hence, the system environment variable `PATH` can also be named, for example, `Path` or `path`. If the `jpf` directory has path `C:\Users\franck\projects\jpf`, then add `C:\Users\franck\projects\jpf\jpf-core\bin` to the system environment variable `PATH`. For more details, do a web search for how to change an environment variable in Windows.

OS X

If the `jpf` directory has path `/Users/franck/projects/jpf`, then add `/Users/franck/projects/jpf/jpf-core/bin` to the system environment variable `PATH`. For more details, do a web search for how to change an environment variable in OS X.

1.1.5 Creating the `site.properties` File

1. Find the value of the standard Java system property *user.home* by running the following Java application.

```java
public class PrintUserHome {
  public static void main(String[] args) {
    System.out.println("user.home = " + System.getProperty("user.home"));
  }
}
```

2. Create a directory named `.jpf` within the directory *user.home*[1].

3. Create in the directory *user.home*`/.jpf` a file named `site.properties`[2]. Assuming, for example, that `jpf-core` is a subdirectory of *user.home*`/projects/jpf`, the file `site.properties` has the following content.

   ```
   # JPF site configuration
   jpf-core=${user.home}/projects/jpf/jpf-core
   extensions=${jpf-core}
   ```

Next, we provide a few examples.

Linux

Assume that the `jpf` directory has path `/cs/home/franck/projects/jpf` and *user.home* is `/cs/home/franck`. Then `site.properties` is located in the directory `/cs/home/franck/.jpf` and its content is

```
# JPF site configuration
jpf-core=${user.home}/projects/jpf/jpf-core
extensions=${jpf-core}
```

If the `jpf` directory has path `/cs/packages/jpf` and *user.home* is `/cs/home/franck`, then `site.properties` is located in the directory `/cs/home/franck/.jpf` and its content is

```
# JPF site configuration
jpf-core=/cs/packages/jpf/jpf-core
extensions=${jpf-core}
```

Windows

Assume that the `jpf` directory has path `C:\Users\franck\projects\jpf` and *user.home* is `C:\Users\franck`. Then `site.properties` is located in the directory `C:\Users\franck\.jpf` and its content is

```
# JPF site configuration
jpf-core=${user.home}/projects/jpf/jpf-core
extensions=${jpf-core}
```

Note that we use `/` instead of `\` in the path. If the `jpf` directory has path `C:\Program Files\jpf` and *user.home* is `C:\Users\franck`, then `site.properties` is located in the directory `C:\Users\franck\.jpf` and its content is

```
# JPF site configuration
jpf-core=C:/Program Files/jpf/jpf-core
extensions=${jpf-core}
```

OS X

Assume that the `jpf` directory has path `/Users/franck/projects/jpf` and *user.home* is `/Users/franck`. Then `site.properties` is located in the directory `/Users/franck/.jpf` and its content is

[1]To create a directory named `.jpf` in Windows Explorer, use `.jpf.` as its name. The dot at the end is necessary, and will be removed by Windows Explorer.

[2]To create a file named `site.properties` in Window Explorer, configure Windows Explorer so that file extensions are visible, create a text file named `site.txt` with the above content, and rename the file to `site.properties`. For more details, do a web search for how to change a file extension in Windows.

```
# JPF site configuration
jpf-core=${user.home}/projects/jpf/jpf-core
extensions=${jpf-core}
```

If the `jpf` directory has path `/System/Library/jpf` and *user.home* is `/Users/franck`, then `site.properties` is located in the directory `/Users/franck/.jpf` and its content is

```
# JPF site configuration
jpf-core=/System/Library/jpf/jpf-core
extensions=${jpf-core}
```

1.2 Updating Sources with Mercurial

Since the sources of JPF change regularly, one should update JPF regularly as well. This can be done as follows.

1. Open a shell (Linux or OS X) or command prompt (Windows), go to the `jpf-core` directory and type

   ```
   hg pull -u
   ```

   We distinguish two cases. If the above command results in output similar to the following, then the sources of JPF have not changed and, hence, we are done.

   ```
   pulling from http://babelfish.arc.nasa.gov/hg/jpf/jpf-core
   searching for changes
   no changes found
   ```

   Otherwise, the above command results in output similar to the following, which indicates that the source of JPF have changed and, therefore, we continue with the next step.

   ```
   pulling from http://babelfish.arc.nasa.gov/hg/jpf/jpf-core
   searching for changes
   adding changesets
   adding manifests
   adding file changes
   added 22 changesets with 117 changes to 71 files
   71 files updated, 0 files merged, 3 files removed, 0 files unresolved
   ```

2. Run ant as described in Section 1.1.3.

1.3 Installing Sources with Mercurial within Eclipse

How to install Eclipse and Mercurial is beyond the scope of this book. We refer the reader to eclipe.org and www.mercurial-scm.org, respectively. We assume that both have been installed. Eclipse should at least be version 3.5 and it should use at least Java version 1.6. We also assume that the path to the `hg` command is already part of the system environment variable `PATH` (see Section 1.1.4). To install JPF within Eclipse with Mercurial, follow the steps below.

1. In Eclipse, select the "Help" menu and its "Eclipse Marketplace ..." submenu.
2. Find "Mercurial" and install it.
3. In Eclipse, select the "File" menu and its "Import" submenu. Select "Mercurial" and "Clone Existing Mercurial Repository" and provide the URL `http://babelfish.arc.nasa.gov/hg/jpf/jpf-core`.
4. Set the user environment variable `JAVA_HOME` as described in Section 1.1.1.

5. Set the user environment variable JUNIT_HOME as described in Section 1.1.2.

6. In Eclipse, build the project jpf-core by expanding the project in the package explorer, locating the file "build.xml," right clicking on it and selecting "Run As" and "Ant Build." This results in output similar to the following.

```
Buildfile: C:\Users\franck\workspace\jpf-core\build.xml
-cond-clean:
-init:
-compile-annotations:
-compile-main:
-compile-peers:
-compile-classes:
-compile-tests:
-compile-examples:
compile:
-version:
build:
[copy] Copying 1 file to C:\Users\franck\workspace\jpf-core\build\main\gov\
nasa\jpf
[jar] Building jar: C:\Users\franck\workspace\jpf-core\build\jpf.jar
BUILD SUCCESSFUL
Total time: 3 seconds
```

7. Create the site.properties file as described in Section 1.1.5. Note that Eclipse places the jpf-core directory within Eclipse's workspace directory by default. Hence, assuming that the workspace has path /cs/home/franck/workspace and *user.home* is /cs/home/franck, the content of site.properties is

```
# JPF site configuration
jpf-core=${user.home}/workspace/jpf-core
extensions=${jpf-core}
```

1.4 Updating Sources with Mercurial within Eclipse

Simply build the project jpf-core again.

1.5 Installing JPF Plugin for Eclipse

1.6 Installing Sources with Mercurial within NetBeans

How to install NetBeans and Mercurial is beyond the scope of this book. We refer the reader to netbeans.org and www.mercurial-scm.org, respectively. We assume that both have been installed. NetBeans should at least be version 6.5. We also assume that the path to the hg command is already part of the system environment variable PATH (see Section 1.1.4). To install JPF within NetBeans with Mercurial, follow the steps below.

1. In NetBeans, set up Mercurial. For more details, do a web search for how to set up Mercurial in NetBeans.

2. In NetBeans, clone the Mercurial repository http://babelfish.arc.nasa.gov/hg/jpf/jpf-core. For more details, do a web search for how to clone a Mercurial repository in NetBeans.

3. Set the user environment variable JAVA_HOME as described in Section 1.1.1.

4. Set the user environment variable `JUNIT_HOME` as described in Section 1.1.2.
5. In NetBeans, build the project `jpf-core`. For more details, do a web search for how to build a project in NetBeans.
6. Create the `site.properties` file as described in Section 1.1.5. Note that NetBeans places the `jpf-core` directory within NetBeans' `NetBeansProjects` directory by default. Hence, assuming that the `NetBeansProjects` has path `/cs/home/franck/NetBeansProjects` and *user.home* is `/cs/home/franck`, the content of `site.properties` is

```
# JPF site configuration
jpf-core=${user.home}/NetBeansProjects/jpf-core
extensions=${jpf-core}
```

1.7 Updating Sources with Mercurial within NetBeans

Simply build the project `jpf-core` again.

1.8 Installing JPF Plugin for NetBeans

As we will discuss in Chapter 2, the JPF plugin can be used to run JPF within NetBeans. This plugin can be installed as follows.

1. Download babelfish.arc.nasa.gov/trac/jpf/attachment/wiki/install/netbeans-plugin/gov-nasa-jpf-netbeans-runjpf.nbm.
2. Install the plugin. For more details, do a web search for how to install a plugin in NetBeans.

1.9 Installing an Extension of JPF

As running example, we consider the extension `jpf-numeric`. This extension allows us to check for numeric properties like overflow. We assume that the reader has already successfully installed `jpf-core`.

1.9.1 Installing Sources with Mercurial

In case sources are available via Mercurial, as is the case for `jpf-numeric`, these can be installed as follows.

1. To get the `jpf-numeric` sources with Mercurial, open a shell (Linux or OS X) or command prompt (Windows), go to the `jpf` directory and type

```
hg clone http://babelfish.arc.nasa.gov/hg/jpf/jpf-numeric
```

2. Run ant as described in Section 1.1.3.

1.9.2 Updating Sources with Mercurial

1. Open a shell (Linux or OS X) or command prompt (Windows), go to the `jpf-numeric` directory and type

```
hg pull -u
```

We distinguish two cases. If the above command results in output similar to the following, then the sources of `jpf-numeric` have not changed and, hence, we are done.

```
pulling from http://babelfish.arc.nasa.gov/hg/jpf/jpf-numeric
searching for changes
no changes found
```

Otherwise, the above command results in output similar to the following, which indicates that the source of `jpf-numeric` have changed and, therefore, we continue with the next step.

```
pulling from http://babelfish.arc.nasa.gov/hg/jpf/jpf-numeric
searching for changes
adding changesets
adding manifests
adding file changes
added 4 changesets with 23 changes to 7 files
7 files updated, 0 files merged, 1 files removed, 0 files unresolved
```

2. Run ant as described in Section 1.1.3.

1.9.3 Installing Sources with Mercurial within Eclipse

In case sources are available via Mercurial, as is the case for `jpf-numeric`, these can be installed as follows.

1. In Eclipse, clone the Mercurial repository `http://babelfish.arc.nasa.gov/hg/jpf/jpf-numeric`.
2. In Eclipse, build the project `jpf-numeric`.

1.9.4 Updating Sources with Mercurial within Eclipse

To update `jpf-numeric`, follow the steps below.

1. In Eclipse, pull the Mercurial repository `http://babelfish.arc.nasa.gov/hg/jpf/jpf-numeric`. For more details, see bitbucket.org/mercurialeclipse/main/wiki/Home.
2. In Eclipse, build the project `jpf-numeric`. For more details, do a web search for how to build a project in Eclipse.

1.9.5 Installing Sources with Mercurial within NetBeans

In case sources are available via Mercurial, as is the case for `jpf-numeric`, these can be installed as follows.

1. In NetBeans, clone the Mercurial repository `http://babelfish.arc.nasa.gov/hg/jpf/jpf-numeric`. For more details, do a web search for how to clone a Mercurial repository in NetBeans.
2. In NetBeans, build the project `jpf-core`. For more details, do a web search for how to build a project in NetBeans.

1.9.6 Updating Sources with Mercurial within NetBeans

To update `jpf-numeric`, follow the steps below.

1. In NetBeans, pull the Mercurial repository `http://babelfish.arc.nasa.gov/hg/jpf/jpf-numeric`. For more details, do a web search for how to clone a Mercurial repository in NetBeans.
2. In NetBeans, build the project `jpf-numeric`. For more details, do a web search for how to build a project in NetBeans.

Chapter 2

Running JPF

Now that we have discussed how to install JPF, let us focus on how to run JPF. It can be run in several different ways. In Section 2.1 and 2.2 we first show how to run JPF in a shell (Linux or OS X) or command prompt (Windows). How to run JPF within Eclipse and NetBeans are the topics of Section 2.3 and 2.4, respectively.

2.1 Running JPF within a Shell or Command Prompt

Let us use the notorious "Hello World" example to show how to run JPF in its most basic form. Consider the following Java application.

```
public class HelloWorld {
 public static void main(String [] args) {
  System.out.println("Hello World!");
 }
}
```

In the directory where we can find `HelloWorld.class`, we create the application properties file named `HelloWorld.jpf`[1] with the following content.

```
target=HelloWorld
classpath=.
```

The key `target` has the name of the application to be checked by JPF as its value. The key `classpath` has JPF's classpath as its value. In this case, it is set to the current directory. It is important not to mix up JPF's classpath with Java's classpath. We will come back to this later.

To run JPF on this example, open a shell (Linux or OS X) or command prompt (Windows) and type `jpf HelloWorld.jpf`. This results in output similar to the following.

```
1  JavaPathfinder v6.0 (rev 1035+) - (C) RIACS/NASA Ames Research Center
2
3
4  ==================================================== system under test
5  HelloWorld.main()
6
7  ==================================================== search started: 6/20/13 1:25 PM
8  Hello World!
9
10 ==================================================== results
```

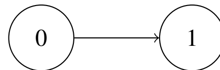
[1]Although the name of the application properties file does not have to match the name of the Java application—we could have called it, for example, `Test.jpf`—we will use that convention in this book.

```
11 no errors detected
12
13 ====================================================== statistics
14 elapsed time:      00:00:00
15 states:            new=1, visited=0, backtracked=1, end=1
16 search:            maxDepth=1, constraints hit=0
17 choice generators: thread=1 (signal=0, lock=1, shared ref=0), data=0
18 heap:              new=366, released=14, max live=0, gc-cycles=1
19 instructions:      4158
20 max memory:        236MB
21 loaded code:       classes=61, methods=1195
22
23 ====================================================== search finished: 6/20/13 1:25 PM
```

Line 1 contains general information. It tells us that we used version 6.0, revision 1035 of JPF. The Research Institute for Advanced Computer Science (RIACS)/NASA Ames Research Center holds the copyright of JPF. The remainder of the output is divided into several parts. The number of parts, their headings and content can be configured. The above output is produced by the default configuration. The first part, line 4–5, describes the system under test. In this case, it is the `main` method of the `HelloWorld` class. The second part, line 7–8, contains the output produced by the system under test and the date and time when JPF was started. In this case, the output is `Hello World!` If the output `I won't say it!` is produced instead, the `classpath` has not been set correctly and, as a consequence, JPF checks the `HelloWorld` application which is part of `jpf-core`. The third part, line 10–11, contains the results of the model checking effort by JPF. In this case, no errors were detected. By default, JPF checks for uncaught exceptions and deadlocks. The fourth and final part, line 13–21, contains some statistics. We will discuss them below. The output ends with line 23 which contains the date and time when JPF finished.

It remains to discuss the statistics part. Line 14 describes the amount of time it took JPF to model check the `HelloWorld` application. Since it took less than one second, JPF reports zero hours, zero minutes and zero seconds.

Line 15 categorizes the states visited by JPF. A state is considered *new* the first time it is visited by JPF. If a state is visited again, it is counted as *visited*. The final states are also called *end* states. Those states reached as a result of a backtrack are counted as *backtrack*. In the above example, JPF visits a state which is an end state (1) and subsequently backtracks to the initial state (0).



We will come back to this classification in Section **??**.

Line 16 provides us with some data about the search for bugs by JPF. The search of JPF is similar to the traversal of a directed graph. The states of JPF correspond to the vertices of the graph and the transitions of JPF correspond to the edges of the graph. In a search, the *depth* of a state is the length of the partial execution, a sequence of transitions, along which the state is discovered. From the above diagram, we can conclude that the maximal depth is one in our example. During the search, JPF checks some *constraints*. By default, it checks two constraints. Firstly, it checks that the depth of the seach is smaller than or equal to the value of the key `search.depth_limit`. By default, its value is $2^{31} - 1$. This JPF property can be configured as we will discuss in Section **??**. Secondly, it checks that the amount of remaining memory is smaller than or equal to the value of the key `search.min_free`. By default, its value is 2^{20}. Also this JPF property can be configured. In our example, no constraints are violated and, hence, the number of constraint hits is zero.

Line 17 contains information about the choice generators. These capture the scheduling and will be discussed in more detail in Section **??**. Some statistics about the heap of JPF's virtual machine are given in line 18.

Line 19 specifies the number of bytecode instructions that have been checked by JPF. The maximum amount of memory used by JPF is given in line 20. Line 21 contains the number of classes and methods that have been checked by JPF.

If the class `HelloWorld` were part of the package `test`, then the application properties file would contain the following.

```
target=test.HelloWorld
classpath=.
```

2.2 Detecting Bugs with JPF

Let us now present some examples of JPF detecting a bug. The examples are kept as simple as possible. As a consequence, they are not realistic representatives of applications on which one might want to apply JPF. However, they allow us to focus on detecting bugs with JPF.

Recall that JPF checks for uncaught exceptions and deadlock by default. Consider the following system under test.

```
1 public class UncaughtException {
2   public static void main(String [] args) {
3     System.out.println(1 / 0);
4   }
5 }
```

Obviously, line 3 throws an exception that is not caught. Running JPF on this example results in output similar to the following.

```
1  JavaPathfinder v6.0 (rev 1035+) - (C) RIACS/NASA Ames Research Center
2
3
4  ====================================================== system under test
5  UncaughtException.main()
6
7  ====================================================== search started: 08/07/13 7:05 PM
8
9  ====================================================== error 1
10 gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
11 java.lang.ArithmeticException: division by zero
12 at UncaughtException.main(UncaughtException.java:3)
13
14
15 ====================================================== snapshot #1
16 thread java.lang.Thread:{id:0,name:main,status:RUNNING,priority:5,
17 lockCount:0,suspendCount:0}
18   call stack:
19   at UncaughtException.main(UncaughtException.java:3)
20
21
22 ====================================================== results
23 error #1: gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
24 "java.lang.ArithmeticException: division by zero a..."
25
26 ====================================================== statistics
27 elapsed time:  00:00:00
28 states:        new=1, visited=0, backtracked=0, end=0
29 search:        maxDepth=1, constraints hit=0
30 choice generators: thread=1 (signal=0, lock=1, shared ref=0), data=0
31 heap:          new=380, released=0, max live=0, gc-cycles=0
32 instructions:  3312
33 max memory:    90MB
```

```
34 loaded code:   classes=65, methods=1226
35
36 ====================================================== search finished: 08/07/13 7:05 PM
```

Line 9–12 report the bug detected. JPF can be configured to detect multiple errors, as we will discuss in Chapter **??**. By default, JPF finishes after detecting the first bug. Line 10 describes the type of bug detected. In this case, the `NoUncaughtExceptionProperty` is violated and, hence, an exception has not been caught. Line 11 and 12 provide the stack trace. From this stack trace we can deduce that the uncaught exception is an `ArithmeticException` and it occurs in line 3 of the `main` method of the `UncaughtException` class. Line 15–19 provides some information for each relevant thread. In this case, there is only a single thread. For each thread JPF records a unique identifier, its name, its status, its priority and two counters. Furthermore, it prints the stack trace of each relevant thread. Line 22-24 summarize the results.

If an assertion, specified by the `assert` statement, fails, an `AssertionError` is thrown. Hence, JPF can detect these. Consider the following application.

```
1 public class FailingAssertion {
2   public static void main(String[] args) {
3     int i = 0;
4     assert i == 1;
5   }
6 }
```

The output produced by JPF for this example is very similar to that produced for the previous example. JPF reports that an uncaught `AssertionError` occurs in line 4 of the `main` method of the `FailingAssertion` class.

2.3 Running JPF within Eclipse

We assume that the reader has installed the JPF plugin (see Section 1.5). Let us also assume that we have created an Eclipse project named `example` which contains the class `HelloWorld` in the default package. If we installed JPF as described in Section 1.3, then the file `HelloWorld.java` can be found in the directory `/cs/home/franck/workspace/examples/src` (Linux) and `C:\Users\franck\workspace\examples\src` (Windows). The corresponding file `HelloWorld.class` can be found in the directory `/cs/home/franck/workspace/examples/bin` (Linux) and `C:\Users\franck \workspace\examples\bin` (Windows).

Next, we create the `HelloWorld.jpf` file. Although this file can be placed in any directory, it is most convenient to place it in the same directory as the `HelloWorld.java` file. As before, the most basic application properties file only contains two keys: `target` and `classpath`. In this case, the value of `classpath` is the directory that contains `HelloWorld.class`. For example, for Windows the content of `HelloWorld.jpf` becomes

```
target=HelloWorld
classpath=C:/Users/franck/workspace/examples/bin
```

Finally, to run JPF on this example within Eclipse, right click on `HelloWorld.jpf` in the package explorer and select the option `Verify...` It results in the output similar to what we have seen in the previous section, preceded by something like

```
Executing command: java -ea -jar C:\Users\franck\workspace\jpf-core\build\RunJPF.jar
+shell.port=4242 C:\Users\franck\workspace\examples\src\HelloWorld.jpf
```

2.4 Running JPF within NetBeans

We assume that the reader has installed the JPF plugin (see Section 1.8). Let us also assume that we have created a NetBeans project named `example` which contains the class `HelloWorld` in the default package. If we installed JPF as described in Section 1.6, then the file `HelloWorld.java` can be found in the directory `/cs/home/franck/`

NetBeansProjects/examples/src (Linux) and C:\Users\franck\NetBeansProjects\examples\src (Windows). The corresponding file HelloWorld.class can be found in the directory /cs/home/franck/NetBeansProjects/examples/build/classes (Linux) and C:\Users\franck\NetBeansProjects\examples\build\classes (Windows).

Next, we create the HelloWorld.jpf file. Although this file can be placed in any directory, it is most convenient to place it in the same directory as the HelloWorld.java file. As before, the most basic application properties file only contains two keys: target and classpath. In this case, the value of classpath is the directory that contains HelloWorld.class. For example, for Windows the content of HelloWorld.jpf becomes

```
target=HelloWorld
classpath=C:/Users/franck/NetBeansProjects/examples/build/classes
```

Finally, to run JPF on this example within NetBeans, right click on HelloWorld.jpf and select the option Verify... It results in the output similar to what we have seen in the previous section, preceded by something like

```
Executing command: java -ea -jar
C:\Users\franck\NetBeansProjects\jpf-core\build\RunJPF.jar
+shell.port=4242 C:\Users\franck\NetBeansProjects\examples\src\HelloWorld.jpf
```

Chapter 3

Configuring JPF

Since JPF has been designed so that it can be easily extended, it should come as no surprise that little is "hard wired" in JPF. As a consequence, there is a need for a mechanism that allows us to configure JPF. How to configure JPF is the topic of this chapter.

3.1 Properties Files

JPF can be configured by command line arguments and properties files. We focus here only the latter. A property file defines properties. Each property consists of a key and a (string) value, separated by an = sign. For example, `target=HelloWorld` assigns to the key `target` the value `HelloWorld`. We distinguish between three different types of properties file:

1. the site properties file,
2. the projects properties files, and
3. the applications properties files.

In general, these files are loaded in the above order. To determine the exact order, one can use the command line argument `-log`. For example, typing `jpf -log HelloWorld.jpf` in a shell or command prompt produces the output as seen in Section 2.1 preceeded by output similar to the following.

```
loading property file: /cs/home/franck/.jpf/site.properties
loading property file: /cs/home/franck/projects/jpf/jpf-core/jpf.properties
loading property file: HelloWorld.jpf
collected native_classpath=/cs/home/franck/projects/jpf/jpf-core/build/jpf.jar,
/cs/home/franck/projects/jpf/jpf-core/lib/junit-4.10.jar
collected native_libraries=null
```

First, the site properties file is loaded. After that, the properties file of `jpf-core` is loaded. And finally, the application properties file `HelloWorld.jpf` is loaded. We will discuss the `native_classpath` and `native_libraries` later in this chapter. Next, we will discuss each type of properties file in some detail.

3.1.1 The Site Properties File

This file is named `site.properties`. In Section 1.1.5 we already discussed how to create this file. It contains the key `extensions`, whose value is the directory where `jpf-core` can be found. For example, `site.properties` may look like

```
# JPF site configuration
jpf-core=${user.home}/projects/jpf/jpf-core
extensions=${jpf-core}
```

Note that the properties file contains two keys: `jpf-core` and `extensions`. In the above file, `${jpf-core}` represents the value associated with the key `jpf-core`, that is, it is the directory in which `jpf-core` can be found.

3.1.2 The Project Properties Files

Each project, such as `jpf-core` and `jpf-numeric`, has its own properties file. This file is named `jpf.properties`. The file can be found in the root directory of the project. This file is not only used to configure the project. It is also used to build the project.

The first entry of `jpf.properties` consists of the project's name as key and `${config_path}` as value. The value of the key `config_path` is the directory of the `jpf.properties` file. For example, the `jpf.properties` file of the project `jpf-numeric` starts with

```
jpf-numeric=${config_path}
```

JPF is implemented as a Java virtual machine (JVM). Therefore, JPF has a classpath. JPF's classpath should contain the Java bytecode of the classes that need to be model checked. That is, it should contain the bytecode of the system under test and any classes used by it.

Since JPF is implemented in Java, it runs on top of a JVM, which we will call the host JVM. This host JVM has a classpath as well. This classpath should contain the bytecode of the classes that are needed to run JPF and its extensions. To distinguish this classpath from JPF's classpath, we call it the native classpath. Both classpaths may contain directories and jar files.

In general, each project adds the examples it includes to the classpath. For example, `jpf-core` includes `HelloWorld` as an example. This class is part of the classpath of `jpf-core`. It also adds those classes needed to run the project to the native classpath. For example, the jar file `/cs/home/franck/projects/jpf/jpf-numeric/build/jpf-numeric.jar`, which contains Java classes that are needed to run `jpf-numeric`, is part of the native classpath of `jpf-numeric`.

Each project has its own classpath and native classpath. To distinguish them, they are prefixed by the project's name. For example, the classpath of `jpf-core` is named `jpf-core.classpath` and the native classpath of `jpf-numeric` is named `jpf-numeric.native_classpath`. Furthermore, each project may also have its own sourcepath. This path contains the Java source code of the classes that are model checked. Although JPF checks Java bytecode, it uses the Java source code to generate feedback to the user when it detects a bug. In particular, it refers to the line numbers of the source code, which makes it easier for the user to locate the bug. For example, the directory `/cs/home/franck/projects/jpf/jpf-core/src/example`, which contains the Java source code of the examples of the `jpf-core` project, is part of the source path of `jpf-core`. How the classpaths, the native classpaths and the source paths are combined will be discussed later in this chapter.

The `jpf.properties` file of the `jpf-core` contains the following.

```
jpf-core.classpath=\
  ${jpf-core}/build/jpf-classes.jar;\
  ${jpf-core}/build/examples

jpf-core.native_classpath=\
  ${jpf-core}/build/jpf.jar;\
  ${jpf-core}/build/jpf-annotations.jar

jpf-core.sourcepath=\
  ${jpf-core}/src/examples
```

The classpath of `jpf-core` contains the jar file `${jpf-core}/build/jpf-classes.jar`. This file contains the bytecode of the classes that model some classes of the Java standard library such as `java.io.File`. These classes are known as model classes. When model checking an application that uses any of these classes, JPF checks

the model class rather than the original class of the Java standard library. We will discuss model classes in more detail in Chapter **??**. The classpath of `jpf-core` also contains the directory `${jpf-core}/build/examples` which contains the bytecode of the examples of `jpf-core`.

The native classpath of `jpf-core` consists of two jar files. The file `${jpf-core}/build/jpf.jar` contains the bytecode of the classes that make up the core of JPF. The file `${jpf-core}/build/jpf-annotations.jar` contains ???. The source path of `jpf-core` solely consists of the directory `${jpf-core}/src/examples` which contains the source code of the examples of `jpf-core`.

The `jpf.properties` file of `jpf-core` defines more than one hundred other properties. Some of them we will discuss in Section **??**.

3.1.3 The Application Properties Files

As we have already seen in Chapter 2, to run JPF we have to write an application properties file. In this file we need to provide values for the keys `target` and `classpath`. The former is the name of the system under test and the latter is part of JPF's classpath. In the application properties file we can set other properties as well.

3.2 JPF Properties

There are a large number of JPF properties that can be set. To determine which JPF properties have been set, one can use the `-show` command line argument. For example, typing `jpf -show HelloWorld.jpf` in a shell or command prompt produces the output as seen in Section 2.1 preceeded by output whose beginning and end are similar to the following.

```
----------- Config contents
branch_start = 1
cg.boolean.false_first = true
cg.break_single_choice = false
...
vm.tree_output = true
vm.untracked = true
vm.verify.ignore_path = true
```

Chapter 4

Using a Listener

To extract information from JPF during its verification effort, JPF uses event driven programming. We assume that reader is already familiar with this programming paradigm. Those readers who are not yet familiar with that paradigm we refer to, for example, [].

In JPF events signal, for example, that a bytecode instruction has been executed, that a class has been loaded, or that the garbage collector has started. To handle these events, JPF contains a numebr of listeners. Such a listener is registered with JPF by setting the value of the key `listener` to the class implementing the listener in the application properties file. For example, extracting a representation of the state space in the DOT format by means of the `StateSpaceDot` listener, which is part of the package `gov.nasa.jpf.listener`, can be specified in the application properties file as follows.

```
listener=gov.nasa.jpf.listener.StateSpaceDot
```

The package `gov.nasa.jpf.listener` contains a number of listeners. Extensions of JPF, such as `jpf-numeric`, contain listeners as well. In the remainder of this chapter we discuss some of them.

4.1 Using the `BudgetChecker`

The listener `BudgetChecker`, which is part of the package `gov.nasa.jpf.listener`, allows us to set constraints for JPF. For example, one can set the maximum number of milliseconds used by JPF. If JPF takes more time than this maximum when using this listener, JPF will terminate and will report that it took more than the set maximum. Apart from the amount time used, we can also constrain the amount of heap space used, the number of instructions executed, the depth of the search and the number of new states. To specify these constraints, the listener has several properties that can be set in the application properties file, as we will show below.

To illustrate the `BudgetChecker` listener, we consider the following app.

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class Constraints {
  public static void main(String[] args) {
    Random random = new Random();
    List<Integer> list = new ArrayList<Integer>();
    for (int i = 0; random.nextBoolean(); i++) {
      list.add(new Integer(i));
    }
  }
}
```

We can run the listener `BudgetChecker` without setting any of its properties and, hence, not setting any constraints, as shown in the application properties file below.

```
target=Constraints
classpath=.
cg.enumerate_random=true
listener=gov.nasa.jpf.listener.BudgetChecker
```

In this case, no constraints are set. As a result, JPF will run out of memory producing the following output

```
JavaPathfinder core system v8.0 (rev 29) - (C) 2005-2014 United States Government. All righ


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

====================================================== search started: 1/27/17 1:01 PM
[SEVERE] JPF out of memory
...
```

We can constrain the maximum amount time used by JPF to 1000 milliseconds as follows. We can assign any long value to the property `budget.max_time`.

```
target=Constraints
classpath=.
cg.enumerate_random=true
listener=gov.nasa.jpf.listener.BudgetChecker
budget.max_time=1000
```

Since JPF cannot complete within one second, it stops after one second and produces the following output.

```
JavaPathfinder core system v8.0 (rev 29) - (C) 2005-2014 United States Government. All righ


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

====================================================== search started: 1/27/17 1:22 PM

====================================================== search constraint
max time exceeded: 00:00:01 >= 00:00:01
...
```

The amount heap space used by JPF can be constrained to a maximum number of bits as follows. We can assign any long value to the property `budget.max_heap`.

```
target=Constraints
classpath=.
cg.enumerate_random=true
listener=gov.nasa.jpf.listener.BudgetChecker
budget.max_heap=10000000
```

Note that 10000000 bits = 10000000 / (1024 * 1024) megabytes = 9.5 megabytes. Since JPF needs more heap space, it stops and produces the following output.

```
====================================================== system under test
Constraints.main()
```

```
============================================================ search started: 1/27/17 1:36 PM

============================================================ search constraint
max heap exceeded: 10MB >= 9MB
...
```

One can constrain JPF from checking more than 100 bytecode instructions as follows. We can assign any long value to the property `budget.max_insn`.

```
target=Constraints
classpath=.
cg.enumerate_random=true
listener=gov.nasa.jpf.listener.BudgetChecker
budget.max_insn=100
```

Since this constraint is violated, JPF produces the following output.

```
JavaPathfinder core system v8.0 (rev 29) - (C) 2005-2014 United States Government. All rig


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

============================================================ search started: 1/27/17 1:47 PM

============================================================ search constraint
max instruction count exceeded: 100
...
```

We can also limit the number of states that JPF explores by setting the property `budget.max_state`. We can assign any int value to this property.

```
target=Constraints
classpath=.
cg.enumerate_random=true
listener=gov.nasa.jpf.listener.BudgetChecker
budget.max_state=100
```

Since the statespace of this app consists of more than 100 states, JPF stops and produces the following output.

```
JavaPathfinder core system v8.0 (rev 29) - (C) 2005-2014 United States Government. All rig


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

============================================================ search started: 1/27/17 1:48 PM

============================================================ search constraint
max states exceeded: 100
```

We can also constrain the depth of the search. The corresponding property, `budget.max_depth`, can take any int value.

```
target=Constraints
classpath=.
cg.enumerate_random=true
```

```
listener=gov.nasa.jpf.listener.BudgetChecker
budget.max_depth=100
```

Since the depth of the search is greater than 100 for this app, JPF stops and produces the following output.

```
JavaPathfinder core system v8.0 (rev 29) - (C) 2005-2014 United States Government. All righ


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

====================================================== search started: 1/27/17 1:53 PM

====================================================== search constraint
max search depth exceeded: 100
...
```

Chapter 5

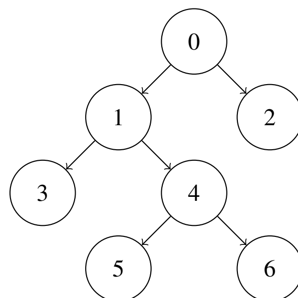
Using a Search Strategy

A search strategy determines in which order the states are visited and the transitions are traversed. Consider, for example, the following app.

```
import java.util.Random;

public class StateSpace {
  public static void main(String[] args) {
    Random random = new Random();
    System.out.print("0 ");
    if (!random.nextBoolean()) {
      System.out.print("1 ");
      if (!random.nextBoolean()) {
        System.out.print("3 ");
      } else {
        System.out.print("4 ");
        if (!random.nextBoolean()) {
          System.out.print("5 ");
        } else {
          System.out.print("6 ");
        }
      }
    } else {
      System.out.print("2 ");
    }
  }
}
```

The corresponding state space can be depicted as follows.



Suppose that the system is in initial state 0. In this state, the system executes line 5–7. If `random.nextBoolean()` of line 7 returns false, then the system transitions to state 1. Otherwise, it transitions to state 2. In state 1, the system executes line 8-9. If `random.nextBoolean()` of line 9 returns false, then the system transitions to state 3. Otherwise, it transitions to state 4. In state 3, the system executes line 10. In state 4, the system executes line 12-13. If `random.nextBoolean()` of line 13 returns false, then the system transitions to state 5. Otherwise, it transitions to state 6. In state 5 and 6, the system executes line 14 and 16, respectively. As a consequence, 0, 1, 2, 3, 4, 5 and 6 are printed when the system is in state 0, 1, 2, 3, 4, 5 and 6, respectively.

5.1 Depth-First and Breadth-First Search

The output produced by JPF for the above app provides us some insight in the order in which the states are visited.

```
JavaPathfinder core system v8.0 (rev 29+) - (C) 2005-2014 United States Government. All ri


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

====================================================== search started: 2/6/17 9:18 PM
0 1 3 4 5 6 2
====================================================== results
no errors detected
...
```

From the above we can conclude that JPF traverses the state space depth first. If we run JPF with the `-show` command line argument, we notice that the JPF property `search.class` is set as follows.

```
----------- Config contents
...
search.class = gov.nasa.jpf.search.DFSearch
...
```

The class `gov.nasa.jpf.search.DFSearch` implements depth-first search. This is the default search strategy. JPF also provides other search strategies. For example, the class `BFSHeuristic`, which is part of the package `gov.nasa.jpf.search.heuristic`, implements breadth-first search. If we set the JPF property `search.class` to `gov.nasa.jpf.search.heuristic.BFSHeuristic`, then JPF produces the following output.

```
JavaPathfinder core system v8.0 (rev 29+) - (C) 2005-2014 United States Government. All ri


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

====================================================== search started: 2/6/17 9:18 PM
0 1 2 3 4 5 6
====================================================== results
no errors detected
...
```

The packages `gov.nasa.jpf.search` and `gov.nasa.jpf.search.heuristic` contain a few other search strategies.

5.2 Search Properties

There are a number of JPF properties that can be set for every search. With the JPF property `search.depth_limit` we can limit the depth of the search. This property can be assigned any integer value. The default value is `Integer.MAX_VALUE`. For example, if we set this property to 2 and use depth-first search, then JPF produces the following output.

```
JavaPathfinder core system v8.0 (rev 29+) - (C) 2005-2014 United States Government. All rig


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

====================================================== search started: 2/6/17 9:18 PM
0 1 3 4 2
====================================================== results
no errors detected
...
```

The JPF property `search.min_free` captures the minimal amount of memory, in bytes, that needs to remain free. The default value is $1024 << 10$. By leaving some memory free, JPF can report that it ran out of memory and provide some useful statistics instead of simply throwing an `OutOfMemoryError`. For example, if we set this property to 100000000 and use depth-first search, then JPF produces the following output.

```
JavaPathfinder core system v8.0 (rev 29+) - (C) 2005-2014 United States Government. All rig


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

====================================================== search started: 2/6/17 9:18 PM
0
====================================================== search constraint
memory limit reached
...
```

The JPF property `search.multiple_errors` tells us whether the search should report multiple errors (or just the first one). The default value is false (that is, by default only the first error is reported after which the search ends).

Chapter 6

Checking a Property

Chapter 7

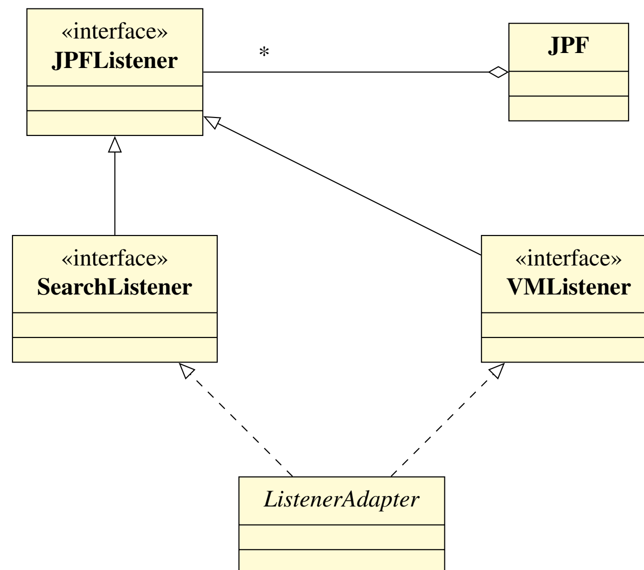
Using the `Verify` Class

Chapter 8

Implementing a Listener

As we already mentioned in Chapter 4, JPF relies on event driven programming. In JPF events signal, for example, that a bytecode instruction has been executed, that a class has been loaded, or that the garbage collector has started. To handle these events, we implement listeners.

Both the search and JPF's virtual machine notify listeners of particular events. The methods corresponding to those events can be found in the interfaces `SearchListener` and `VMListener`, which are part of the packages `gov.nasa.jpf.search` and `gov.nasa.jpf.vm`, respectively. A listener implements either one of these or both interfaces.



Since usually we are only interested in a few events, we can provide the methods corresponding to the remaining events with a default implementation (since all methods are void, we can just provide a method with an empty body). To avoid coding these methods, JPF has already provided the classes `SearchListenerAdapter` and `ListenerAdapter`, which are part of the packages `gov.nasa.jpf.search` and `gov.nasa.jpf`, respectively. The former provides default implementations for all the methods specified in the `SearchListener` interface, and the latter provides default implementations for all the methods specified in the `SearchListener` and `VMListener` interface.

The interface `JPFListener`, which is part of package `gov.nasa.jpf`, represents listeners of JPF. Both `SearchListener` and `VMListener` extend this interface. The class `JPF` is the core of JPF. Among many other objects, it creates the listeners. The relationships among the above mentioned classes is captured in the following UML diagram.

8.1 The SearchListener Interface

The SearchListener interface contains methods that are related to events triggered by JPF's search. For example, when JPF starts its search, the method searchStarted is invoked. The SearchListener contains the following methods.

```
public interface SearchListener extends JPFListener {
  void stateAdvanced(Search search);
  void stateProcessed(Search search);
  void stateBacktracked(Search search);
  void statePurged(Search search);
  void stateStored(Search search);
  void stateRestored(Search search);
  void searchProbed(Search search);
  void propertyViolated(Search search);
  void searchStarted(Search search);
  void searchFinished(Search search);
}
```

Note that these methods receive a reference to a Search object as an argument. This Search object contains information about the search of the state space by JPF. As we already mentioned, JPF invokes the method searchStarted at the start of the search.

8.2 Printing the State Space

As a first example, we implement a listener which prints the states and transitions visited by the search in the following simple format:

```
0 -> 1
1 -> 2
0 -> 3
3 -> 4
4 -> 2
```

The above tells us that the search started in the initial state 0 and made a transition to a new state 1. From state 1 it made a transition to a new state 2. Next, it made a transition from state 0 to state 2. Etcetera. We name our class StateSpacePrinter.

In our StateSpacePrinter, we are only interested in the events signalling a change of state. Therefore, we implement the SearchListener interface. Since we are not interested in all event notifications by the search, we start from default implementations by extending the SearchListenerAdapter class. Hence, we arrive at the following method header.

```
public class StateSpacePrinter extends SearchListenerAdapter
  implements SearchListener
```

The only three methods that signal a state change are stateAdvanced, stateBacktracked and stateRestored. In order to print a transition, we need both the source and target state of the transition. We introduce attributes to keep track of the source and target state. In JPF each state has a unique identifier, which is a nonnegative integer.

```
private int source;
private int target;
```

We initialize these attributes in the constructor to an unknown state, which we represent by -1.

```
public StateSpacePrinter() {
  source = -1;
```

```
  target = -1;
}
```

We have left to implement the methods `stateAdvanced`, `stateBacktracked` and `stateRestored`. These can be implemented in the following straightforward way.

```
public void stateAdvanced(Search search) {
  source = target;
  target = search.getStateId();
  if (source != -1) {
    System.out.printf("%d -> %d%n", source, target);
  }
}

public void stateBacktracked(Search search) {
  target = search.getStateId();
}

public void stateRestored(Search search) {
  target = search.getStateId();
}
```

In our implementation of the methods we use `search.getStateId()` to get the identifier of the current state.

8.3 The State Space in DOT Format

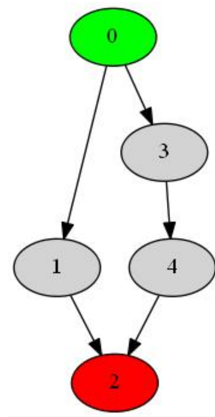
Rather than printing the transitions in the terminal and, hence, interleaving them with the output produced by the system under test, we can write the transitions to a file. Also, instead of representing them as text, we can save the states and transitions in DOT format. This allows us to use tools such as dotty to layout and view the states and transitions. We will develop such a listener next. We colour the initial state green and all final states red. The textual representation

```
0 -> 1
1 -> 2
0 -> 3
3 -> 4
4 -> 2
```

where 0 is the initial state and 2 is a final state corresponds to the following DOT representation.

```
digraph statespace {
node [style=filled]
0 [fillcolor=green]
0 -> 1
1 -> 2
2 [fillcolor=red]
0 -> 3
3 -> 4
4 -> 2
2 [fillcolor=red]
}
```

This DOT representation gives rise to the following graphical representation of the state space.



To write to a file, we introduce an attribute of type `PrintWriter` as follows.

```
private PrintWriter writer;
```

We initialize this attribute in the `searchStarted` method. As the name of the DOT file, we use the name of the system under test with ".dot" as suffix. The name of the system under test can be obtained using the method invocation `search.getVM().getSUTName()`. Whenever the creation of the `PrintWriter` fails, we want to print an error message and subsequently terminate JPF. The latter is done using `search.terminate()`.

In the search starts we also write

```
digraph statespace {
node [style=filled]
0 [fillcolor=green]
```

to the file. In JPF, 0 is the identifier of the initial state. This is accomplished as follows.

```
public void searchStarted(Search search) {
  String name = search.getVM().getSUTName() + ".dot";
  try {
    writer = new PrintWriter(name);
    writer.println("digraph statespace {");
    writer.println("node [style=filled]");
    writer.println("0 [fillcolor=green]");
  }
  catch (FileNotFoundException e) {
    System.out.println("Listener could not write to file " + name);
    search.terminate();
  }
}
```

The method `stateAdvanced` is modified as follows.

```
public void stateAdvanced(Search search) {
  source = this.target;
  target = search.getStateId();
  if (source != -1) {
    writer.printf("%d -> %d%n", source, target);
  }
  if (search.isEndState()) {
    writer.printf("%d [fillcolor=red]%n", target);
  }
}
```

When the search terminates, we write } to the file and close the PrintWriter object and release any system resources associated with it. This is done by adding the following method.

```
public void searchFinished(Search search) {
 writer.println("}");
 writer.close();
}
```

8.4 The VMListener Interface

The VMListener interface contains methods that are related to events triggered by JPF's virtual machine. For example, when JPF , the method is invoked. The VMListener contains the following methods.

```
public interface VMListener extends JPFListener {
 void vmInitialized(VM vm);

 void executeInstruction(VM vm, ThreadInfo currentThread,
  Instruction instructionToExecute);
 void instructionExecuted(VM vm, ThreadInfo currentThread,
  Instruction nextInstruction, Instruction executedInstruction);

 void threadStarted(VM vm, ThreadInfo startedThread);
 void threadBlocked(VM vm, ThreadInfo blockedThread, ElementInfo lock);
 void threadWaiting(VM vm, ThreadInfo waitingThread);
 void threadNotified(VM vm, ThreadInfo notifiedThread);
 void threadInterrupted(VM vm, ThreadInfo interruptedThread);
 void threadTerminated(VM vm, ThreadInfo terminatedThread);
 void threadScheduled(VM vm, ThreadInfo scheduledThread);

 void loadClass(VM vm, ClassFile cf);
 void classLoaded(VM vm, ClassInfo loadedClass);

 void objectCreated(VM vm, ThreadInfo currentThread, ElementInfo newObject);
 void objectReleased(VM vm, ThreadInfo currentThread, ElementInfo releasedObject);
 void objectLocked(VM vm, ThreadInfo currentThread, ElementInfo lockedObject);
 void objectUnlocked(VM vm, ThreadInfo currentThread, ElementInfo unlockedObject);
 void objectWait(VM vm, ThreadInfo currentThread, ElementInfo waitingObject);
 void objectNotify(VM vm, ThreadInfo currentThread, ElementInfo notifyingObject);
 void objectNotifyAll(VM vm, ThreadInfo currentThread, ElementInfo notifyingObject);
 void objectExposed(VM vm, ThreadInfo currentThread, ElementInfo fieldOwnerObject,
  ElementInfo exposedObject);
 void objectShared(VM vm, ThreadInfo currentThread, ElementInfo sharedObject);

 void gcBegin(VM vm);
 void gcEnd(VM vm);

 void exceptionThrown(VM vm, ThreadInfo currentThread, ElementInfo thrownException);
 void exceptionBailout(VM vm, ThreadInfo currentThread);
 void exceptionHandled(VM vm, ThreadInfo currentThread);

 void choiceGeneratorRegistered(VM vm, ChoiceGenerator<?> nextCG,
  ThreadInfo currentThread, Instruction executedInstruction);
```

```
  void choiceGeneratorSet(VM vm, ChoiceGenerator<?> newCG);
  void choiceGeneratorAdvanced(VM vm, ChoiceGenerator<?> currentCG);
  void choiceGeneratorProcessed(VM vm, ChoiceGenerator<?> processedCG);

  void methodEntered(VM vm, ThreadInfo currentThread, MethodInfo enteredMethod);
  void methodExited(VM vm, ThreadInfo currentThread, MethodInfo exitedMethod);
}
```

8.5 Printing the Bytecode Mnemonics

8.6 The `PublisherExtension` Interface

Rather than using print statements as in the `StateSpacePrinter`, we can also use JPF's report system to provide the results of the verification effort to the user. To exploit the report system, we implement the interface `PublisherExtension`, which is part of the package `gov.nasa.jpf.report`.

```
public interface PublisherExtension {
  void publishStart(Publisher publisher);
  void publishTransition(Publisher publisher);
  void publishPropertyViolation(Publisher publisher);
  void publishConstraintHit(Publisher publisher);
  void publishFinished(Publisher publisher);
  void publishProbe(Publisher publisher);
}
```

The `PublisherExtension` interface contains methods which are invoked whenever a particular event occurs. For example, the `publishTransition` method is invoked whenever a transition is traversed by JPF. These methods receive an object of type `Publisher` as argument. The class `Publisher`, which is part of the package `gov.nasa.jpf.report`, contains methods that provide information about how the data is formatted by JPF. For example, the method `getOut` returns the `PrintWriter` object which is used by JPF to print the data. The type of `PrintWriter` object that JPF uses can be set by means of the JPF property report.publisher. Its default value is console, but JPF can also produce XML by setting

```
report.publisher=xml
```

in the application properties file.

8.7 The State Space in XML Format

The earlier mentioned class `ListenerAdapter` provides default implementations of all method specifications in `PublisherExtension`. Hence, the header of our `StateSpacePrinter` becomes as follows.

```
public class StateSpacePrinter extends ListenerAdapter
  implements SearchListener, PublisherExtension
```

Since we want to publish data whenever a transition is traversed, we implement the `publishTransition` method. This method can be implemented as follows.

```
public void publishTransition(Publisher publisher) {
  PrintWriter out = publisher.getOut();
  if (source != -1) {
    ???
  }
}
```

Now that we have moved the printing of the transition to the `publishTransition` method, we can remove it from the `stateAdvanced` method. The latter method can be simplified to the following.

```
public void stateAdvanced(Search search) {
  source = target;
  target = search.getStateId();
}
```

We have left to register our `StateSpacePrinter` class as a `PublisherExtension`. This is done in the constructor as follows.

```
public StateSpacePrinter(Config config, JPF jpf) {
  source = -1;
  target = -1;
  jpf.addPublisherExtension(Publisher.class, this);
}
```

Note that this constructor takes two arguments: a `Config` object and a `JPF` object. The latter is needed for JPF's reporting system.

8.8 Parametrizing a Listener

Now assume that we want to parametrize the listener, that is, we want to customize it by means of some JPF property. Suppose we want to give the user the opportunity to specify which string is used to separate the states of a transition in our `StateSpacePrinter`. By default, this separator is "->". We introduce the JPF property statespaceprinter.separator. This property can be set by the user in a properties file or as a command line argument. For example, the user can add

```
statespaceprinter.separator=-->
```

to the application properties file. To our class, we add the attribute

```
private String separator;
```

We initialize this attribute in the constructor. Recall that a `Config` object contains the JPF properties. The `Config` class contains methods to extract the values of JPF properties. For example, the method `getString` extracts the string value of some JPF property. The method takes two arguments, the name of the JPF property and the default value (in case the `Config` object does not contain any value for the JPF property). For example,

```
config.getString("statespaceprinter.separator", "->");
```

returns the value of the JPF property statespaceprinter.separator if that property is defined in the `Config` object `config`, and returns "->" otherwise.

We change the signature of the constructor: we add a parameter of type `Config`. When JPF constructs a `StateSpacePrinter`, it provides a `Config` object as argument to the constructor. Hence, our modified constructor now looks as follows.

```
public StateSpacePrinter(Config config) {
  source = -1;
  separator = config.getString("statespaceprinter.separator", "->");
}
```

A constructor of a listener can either take no arguments, one argument of type `Config`, or two arguments, the first of type `Config` and the second of type `VM`. We have already seen examples of the first and second case above. We will present an example of the third case later.

Of course, we also have to modify the `stateAdvanced` method as follows.

```java
public void stateAdvanced(Search search) {
  source = target;
  int target = search.getStateId();
  if (source != -1) {
    System.out.printf("%d %s %d%n", source, separator, target);
  }
}
```

8.9 Compiling a new Listener

The `StateSpacePrinter` listener relies on several classes that are part of JPF such as, for example, `ListenerAdapter`, which is part of the package `gov.nasa.jpf`. The bytecode of these classes, which are needed to compile the `StateSpacePrinter` class, can be found in the jar file `jpf.jar`. This jar file can be found in the build directory of `jpf-core`. If `jpf-core` has been installed in the directory `/cs/home/franck/projects/jpf/jpf-core`, then the `ListenerAdapter` class can be compiled as follows.

```
javac -cp /cs/home/franck/projects/jpf/jpf-core/build/jpf.jar;. StateSpacePrinter.java
```

In Eclipse, if the listener `StateSpacePrinter` is in a different project than `jpf-core`, then the jar file `jpf.jar` needs to be added to the build path as an external library.

8.10 Using a new Listener: `classpath` versus `native_classpath`

As we already mentioned earlier, JPF is implemented as a JVM. Therefore, JPF has a classpath. This classpath can be set in a configuration file using the `classpath` property. JPF's classpath should contain the Java bytecode of the classes that need to be model checked. That is, it should contain the bytecode of the system under test and any classes used by it.

Since JPF is implemented in Java, it runs on top of a JVM, which we will call the host JVM. This host JVM has a classpath as well. This classpath can be set in a configuration file using the `native_classpath` property. It should contain the bytecode of the classes that are needed to run JPF and its extensions, such as listeners.

Assume that the bytecode of the listener `StateSpacePrinter` can be found in the directory `/cs/home/franck/projects/listeners/`. Suppose that the bytecode of the app `HelloWorld` can be found in the directory `/cs/home/franck/projects/code/`. In that case, we can run JPF on the `HelloWorld` app with the `StateSpacePrinter` listener using the following application configuration file.

```
target=HelloWorld
classpath=/cs/home/franck/projects/code/
native_classpath=/cs/home/franck/projects/listeners/
listener=StateSpacePrinter
```

Chapter 9

Implementing a Search Strategy

In this chapter we provide a recipe for implementing a search strategy within Java Pathfinder (JPF). As a running example, we use depth-first search (DFS). We name our class `DFSearch`.

9.1 The Structure of the Class

The `Search` class, which resides in the package `gov.nasa.jpf.search`, contains numerous attributes and methods that may be of use for implementing a search strategy. Hence, we extend this class.

```
import gov.nasa.jpf.search.Search;

public class DFSearch extends Search {
  ...
}
```

The constructor of the `Search` class takes two arguments. The first argument is a `Config` object. The class `Config` is part of the package `gov.nasa.jpf`. The `Config` object contains the JPF properties. These properties can be set, for example, in the jpf.properties file. The second argument is a `VM` object. The package `gov.nasa.jpf.vm` contains the `VM` class. The `VM` object provides the search a reference to JPF's virtual machine. To properly initialize the `Search` object, we add the following constructor to our `DFSearch` class (and import the classes `Config` and `VM`).

```
public DFSearch(Config config, VM vm) {
  super(config, vm);
}
```

If JPF is configured to use our `DFSearch`, JPF will construct a `DFSearch` object with a `Config` object capturing JPF's configuration and a `VM` object representing JPF's virtual machine. For `DFSearch` to properly interact with JPF, its constructor has to be declared public.

The `Search` class is abstract. It contains the abstract method lstinlinesearch. This method drives the search. It visits the states of the system under test in a systematic way by traversing transitions. In our `DFSearch` we implement the `search` method. We develop our implementation of the `search` method in a number of steps.

9.2 The Basic Search

To implement the basis of the search, we use the following three methods from the `Search` class that categorize the current state. The method `isNewState` tests whether the current state has been visited before by JPF. The method `isEndState` tests whether the current state is a final state by checking that no threads are alive anymore. The method `isIgnoredState` tests whether the current state should be ignored by the search. States can, for example,

be ignored by using in the system under test the method `ignoreIf(boolean)` of JPF's class `Verify` which is part of the package `gov.nasa.jpf.vm`. Several other methods that characterize the current state can be found in the `Search` class.

To visit the states, the `Search` class provides the following two methods. The `backtrack` method returns the search to the source state of the transition along which the current state was discovered by the search. The method returns a boolean: whether the backtrack was successful. A backtrack fails, for example, in the initial state. The `forward` method moves the search from the current state to another state along an unexplored transition. The method returns a boolean: whether the forward was successful. A forward fails, for example, if there are no unexplored transitions from the current state. The `forward` method also checks whether any property is violated after the unexplored transition has been traversed (we will come back to this later).

The class `VM` contains the method `restoreState(VMState)` which restores the given state, which has been visited before. Although we do not need this method to implement DFS, it comes in handy when implementing breadth-first search (BFS) as we will show later.

Our implementation of the `search` method is similar to the implementation of depth-first search by means of a stack (see, for example, [CLRS09,]). JPF keeps track of the stack and which states have already been visited by the traversal. The latter can be tested with the `isNewState` method inherited from the `Search` class. Before the start of the `search` method JPF has already put the initial state on the stack. The methods `forward` and `backtrack`, also inherited from the `Search` class, are the JPF counterparts of push and pop, respectively. Note that the stack is empty if and only if a pop fails. In our setting this amounts to the `backtrack` failing, that is, returning false.

```
public void search() {
  while (true) {
    if (!isNewState() || !forward()) {
      if (!backtrack()) {
        break;
      }
    }
  }
}
```

Whenever we reach a final state or an ignored state, we can backtrack.

```
public void search() {
  while (true) {
    if (!isNewState() || isEndState() || isIgnoredState() || !forward()) {
      if (!backtrack()) {
        break;
      }
    }
  }
}
```

Below, we extend the above search method in several ways by adding different, orthogonal, aspects to the search.

9.3 Other Components

Other components of JPF, such as listeners, can end a search by setting the attribute `done` of the class `Search` to true. This is reflected in the `search` method as follows.

```
public void search() {
  while (!done) {
    if (!isNewState() || isEndState() || isIgnoredState() || !forward()) {
      if (!backtrack()) {
        break;
```

```
      }
    }
  }
}
```

Other components of JPF can also request a search to backtrack by means of the method `requestBacktrack` of the class `Search`. This method simply sets a boolean attribute to true. The method `checkAndResetBacktrack-Request` of the class `Search` tests whether a backtrack has been requested and resets the attribute to false. Requests of backtracks can be addressed in our search method as follows.

```
public void search() {
  while (true) {
    if (!isNewState() || isEndState() || isIgnoredState() || checkAndResetBacktrackRequest(
        || !forward()) {
      if (!backtrack()) {
        break;
      }
    }
  }
}
```

A search can be configured in several ways. Next, we will introduce the JPF properties relevant to a search.

9.4 Search Properties

Recall that JPF can be configured to limit the depth of the search by setting the JPF property `search.depth_limit`. The `Search` class contains the attribute `depth` that can be used to keep track of the depth of the search. It also provides the method `getDepthLimit` which returns the maximal allowed depth of the search.

We can limit the depth of the search as follows.

```
public void search() {
  final int MAX_DEPTH = getDepthLimit();
  depth = 0;
  while (true) {
    if (!isNewState() || isEndState() || isIgnoredState() || depth >= MAX_DEPTH || !forward
      if (!backtrack()) {
        break;
      } else {
        depth--;
      }
    } else {
      depth++;
    }
  }
}
```

Recall that the JPF property `search.min_free` captures the minimal amount of memory, in bytes, that needs to remain free. The method `checkStateSpaceLimit` of the class `Search` checks whether the minimal amount of memory that should be left free is still available.

We end the search if we run almost out of memory as follows. Note that we only check this if the `forward` succeeds as a traversal of a transition by JPF may lead to a new state and, hence, may require additional memory.

```
public void search() {
  while (true) {
```

```
    if (!isNewState() || isEndState() || isIgnoredState() || !forward()) {
      if (!backtrack()) {
        break;
      }
    } else {
      if (!checkStateSpaceLimit()) {
        break;
      }
    }
  }
}
```

Recall that the JPF property `search.multiple_errors` tells us whether the search should report multiple errors (or just the first one). The `forward` method also checks whether any property is violated after the unexplored transition has been traversed. If a violation has been detected then the attribute `done` is set to true if and only if JPF has been configured to report at most one error. The method `hasPropertyTermination` of the class `Search` checks whether a violation was encountered during the last transition. The method returns true if and only if a violation was encountered and the attribute `done` is set to true. In the context below (that is, being invoked immediate after the `forward` method), the latter denotes that JPF has been configured to report at most one error. Furthermore, if `done` is set to false (which in this context denotes that JPF has been configured to report multiple errors), then the attribute requesting a backtrack is set to true.

```
public void search() {
  while (true) {
    if (!isNewState() || isEndState() || isIgnoredState() || checkAndResetBacktrackRequest(
        || !forward()) {
      if (!backtrack()) {
        break;
      }
    } else {
      if (hasPropertyTermination()) {
        break;
      }
    }
  }
}
```

Assume that we are in a state that is new and neither final nor ignored and also suppose that the `forward` call returns true. We distinguish the following three cases.

1. If the `forward` method does not encounter a violation, then `hasPropertyTermination` returns false and, hence, line 9 is not executed and the search continues.

2. If the `forward` method encounters a violation and `search.multiple_errors` is set to false, then the attribute `done` is set to true and, therefore, `hasPropertyTermination` returns true. Hence, line 9 is executed and the search terminates.

3. If the `forward` method encounters a violation and `search.multiple_errors` is set to true, then the attribute `done` is set to false and, therefore, `hasPropertyTermination` returns false and the attribute requesting a backtrack is set to true. Hence, line 9 is not executed and the search continues. In the next iteration, the search attempts to backtrack since `checkAndResetBacktrackRequest` returns true.

9.5 Notifications

A search should also notify listeners of particular events. The `Search` class provides the following methods. Note that the methods below correspond to the methods of the interface `SearchListener`, which can be found in the package `gov.nasa.jpf.search`.

The first group contains methods that notify listeners of events related to the current state of the search. The method `notifyStateAdvanced` notifies the listeners that the current state has been reached as a result of a successful forward invocation. The method `notifyStateProcessed` notifies the listeners that the current state has been fully explored, that is, it has no unexplored outgoing transitions. The method `notifyStateBacktracked` notifies the listeners that the current state has been reached by means of a backtrack.

The method `notifySearchStarted` notifies the listeners that the search has started and the method `notifySearchFinished` notifies the listeners that the search has finished. Below, we present the simplest extension of the basic search so that we can introduce all the above mentioned notifications.

```
public void search() {
  notifySearchStarted();
  while (true) {
    if (!isNewState() || isEndState() || isIgnoredState() || !forward()) {
      notifyStateProcessed();
      if (!backtrack()) {
        break;
      } else {
        notifyStateBacktracked();
      }
    } else {
      notifyStateAdvanced();
    }
  }
  notifySearchFinished();
}
```

9.6 The Complete Search

We have left to adding the appropriate invocations of the `notifyPropertyViolated` and `notifySearchConstraintHit` methods. The method `notifyPropertyViolated` notifies the listeners that a violation has been encountered in the current state. Recall that in our setting the method `hasPropertyTermination` returns true if and only if a violation of a property has been detected and JPF has been configured to report at most one error. Hence, this method cannot be used to report violations of properties. Immediately after an invocation of the `forward` method, the attribute `currentError` of the class `Search` is null if and only if no violation has been detected. Therefore, we can use this attribute to determine whether we should report a violation of a property.

As we already mentioned in Section 2.1, JPF checks two constraints by default: the depth of the seach and the amount of remaining memory. The method `notifySearchConstraintHit(String)` notifies the listeners that a constraint has been violated. Combining all the above and adding the appropriate invocations of the `notifyPropertyViolated` and `notifySearchConstraintHit` methods, we arrive at the following.

```
public void search() {
  notifySearchStarted();
  final int MAX_DEPTH = getDepthLimit();
  depth = 0;
  while (!done) {
    if (!isNewState() || isEndState() || isIgnoredState() || depth >= MAX_DEPTH
        || checkAndResetBacktrackRequest() || !forward()) {
```

```
      notifyStateProcessed();
      if (!backtrack()) {
        break;
      } else {
        depth--;
        notifyStateBacktracked();
      }
    } else {
      depth++;
      notifyStateAdvanced();
      if (currentError != null) {
        notifyPropertyViolated();
        if (hasPropertyTermination()) {
          break;
        }
      }
      if (depth >= MAX_DEPTH) {
        notifySearchConstraintHit("depth limit reached");
      }
      if (!checkStateSpaceLimit()) {
        notifySearchConstraintHit("memory limit reached");
        break;
      }
    }
  }
  notifySearchFinished();
}
```

The above search method is slightly different from the one in the `DFSearch` class of the `gov.nasa.jpf.search` package.

9.7 Breadth-First Search

As a second example, we implement BFS in a class named `BFSearch`. We start with extending the class `Search`.

```
import gov.nasa.jpf.Config;
import gov.nasa.jpf.vm.VM;
import gov.nasa.jpf.search.Search;

public class BFSearch extends Search {
  public DFSearch(Config config, VM vm) {
    super(config, vm);
  }
}
```

To implement the basic search, we need a few new ingredients. In particular, we use two methods of the class `VM` which represents JPF's virtual machine. The method `getRestorableState` returns a `RestorableVMState` object which represents the current state. The class `RestorableVMState` is part of the package `gov.nasa.jpf.vm`. The method `restoreState(RestorableVMState)` restores the given state, that is, the current state becomes a previously visited state provided as an argument. Furthermore, the method `isNewState` of the class `Search` tests whether the current state is new, that is, it has not been visited before by the search.

```
public void search() {
```

```
  List<RestorableVMState> currentLevel = new LinkedList<RestorableVMState>();
  currentLevel.add(vm.getRestorableState());
  while (!currentLevel.isEmpty()) {
    List<RestorableVMState> nextLevel = new LinkedList<RestorableVMState>();
    for (RestorableVMState state : currentLevel) {
      vm.restoreState(state);
      while (forward()) {
        if (isNewState() && !isEndState() && !isIgnoredState()) {
          nextLevel.add(vm.getRestorableState());
        }
        backtrack();
      }
    }
    currentLevel = nextLevel;
  }
}
```

As before, the attribute `done` is used to end the search.

```
public void search() {
  List<RestorableVMState> currentLevel = new LinkedList<RestorableVMState>();
  currentLevel.add(vm.getRestorableState());
  while (!currentLevel.isEmpty() && !done) {
    List<RestorableVMState> nextLevel = new LinkedList<RestorableVMState>();
    for (RestorableVMState state : currentLevel) {
      vm.restoreState(state);
      while (forward() && !done) {
        if (isNewState() && !isEndState() && !isIgnoredState()) {
          nextLevel.add(vm.getRestorableState());
        }
        backtrack();
      }
    }
    currentLevel = nextLevel;
  }
}
```

In our BFS implementation we decided *not* to support backtrack requests by other JPF components. The class `Search` contains the method `supportBacktrack` which tests whether a search supports backtrack requests. This method of the `Search` class always returns true. In our subclass `BFSearch`, we override this method as follows.

```
public boolean supportBacktrack() {
  return false;
}
```

The depth of the search can be limited as follows.

```
public void search() {
  final int MAX_DEPTH = getDepthLimit();
  depth = 0;
  List<RestorableVMState> currentLevel = new LinkedList<RestorableVMState>();
  currentLevel.add(vm.getRestorableState());
  while (!currentLevel.isEmpty() && depth < MAX_DEPTH) {
    List<RestorableVMState> nextLevel = new LinkedList<RestorableVMState>();
    for (RestorableVMState state : currentLevel) {
```

```
      vm.restoreState(state);
      while (forward()) {
        if (isNewState() && !isEndState() && !isIgnoredState()) {
          nextLevel.add(vm.getRestorableState());
        }
        backtrack();
      }
    }
    currentLevel = nextLevel;
    depth++;
  }
}
```

To end the search when insufficient memory is available, we use the method `checkStateSpaceLimit` and the attribute `done` as follows.

```
public void search() {
  List<RestorableVMState> currentLevel = new LinkedList<RestorableVMState>();
  currentLevel.add(vm.getRestorableState());
  while (!currentLevel.isEmpty() && !done) {
    List<RestorableVMState> nextLevel = new LinkedList<RestorableVMState>();
    for (RestorableVMState state : currentLevel) {
      vm.restoreState(state);
      while (forward() && !done) {
        if (!checkStateSpaceLimit()) {
          done = true;
        } else if (isNewState() && !isEndState() && !isIgnoredState()) {
          nextLevel.add(vm.getRestorableState());
        }
        backtrack();
      }
    }
    currentLevel = nextLevel;
  }
}
```

The JPF property `search.multiple_errors` can be dealt with in the same way as in our implementation of DFS.

```
public void search() {
  List<RestorableVMState> currentLevel = new LinkedList<RestorableVMState>();
  currentLevel.add(vm.getRestorableState());
  while (!currentLevel.isEmpty() && !done) {
    List<RestorableVMState> nextLevel = new LinkedList<RestorableVMState>();
    for (RestorableVMState state : currentLevel) {
      vm.restoreState(state);
      while (forward() && !done) {
        if (hasPropertyTermination()) {
          done = true;
        } else if (isNewState() && !isEndState() && !isIgnoredState()) {
          nextLevel.add(vm.getRestorableState());
        }
        backtrack();
      }
```

```
    }
    currentLevel = nextLevel;
  }
}
```

In the notification code, we use two methods that we have not discussed before. The method `notifyStateStored` notifies the listeners that the current state has been stored (so that it can be restored later). The method `notifyStateRestored` notifies the listeners that the current state has been restored (by means of the `restoreState` method). Below, we introduce all the notifications apart from `notifyPropertyViolated` and `notifySearchConstraintHit`.

```
public void search() {
  notifySearchStarted();
  List<RestorableVMState> currentLevel = new LinkedList<RestorableVMState>();
  currentLevel.add(vm.getRestorableState());
  notifyStateStored();
  while (!currentLevel.isEmpty()) {
    List<RestorableVMState> nextLevel = new LinkedList<RestorableVMState>();
    Iterator<RestorableVMState> iterator = currentLevel.iterator();
    while (iterator.hasNext()) {
      vm.restoreState(iterator.next());
      notifyStateRestored();
      while (true) {
        if (!forward()) {
          notifyStateProcessed();
          break;
        } else {
          notifyStateAdvanced();
          if (isNewState() && !isEndState() && !isIgnoredState()) {
            nextLevel.add(vm.getRestorableState());
            notifyStateStored();
          }
          if (backtrack()) {
            notifyStateBacktracked();
          }
        }
      }
    }
    currentLevel = nextLevel;
  }
  notifySearchFinished();
}
```

```
public void search() {
  notifySearchStarted();
  List<RestorableVMState> currentLevel = new LinkedList<RestorableVMState>();
  currentLevel.add(vm.getRestorableState());
  notifyStateStored();
  while (!currentLevel.isEmpty()) {
    List<RestorableVMState> nextLevel = new LinkedList<RestorableVMState>();
    for (RestorableVMState state : currentLevel) {
      vm.restoreState(state);
      notifyStateRestored();
```

```
      while (forward()) {
        notifyStateAdvanced();
        if (isNewState() && !isEndState() && !isIgnoredState()) {
          nextLevel.add(vm.getRestorableState());
          notifyStateStored();
        }
        notifyStateProcessed();
        backtrack();
        notifyStateBacktracked();
      }
    }
    currentLevel = nextLevel;
  }
  notifySearchFinished();
}
```

We conclude by combining all the above and adding the appropriate invocations of the `notifyPropertyViolated` and `notifySearchConstraintHit` methods.

```
public void search() {
  notifySearchStarted();
  final int MAX_DEPTH = getDepthLimit();
  depth = 0;
  List<RestorableVMState> currentLevel = new LinkedList<RestorableVMState>();
  currentLevel.add(vm.getRestorableState());
  notifyStateStored();
  while (!currentLevel.isEmpty() && !done && depth < MAX_DEPTH) {
    List<RestorableVMState> nextLevel = new LinkedList<RestorableVMState>();
    for (RestorableVMState state : currentLevel) {
      vm.restoreState(state);
      notifyStateRestored();
      while (forward() && !done) {
        notifyStateAdvanced();
        if (currentError != null) {
          notifyPropertyViolated();
        }
        if (!checkStateSpaceLimit()) {
          notifySearchConstraintHit("memory limit reached");
          done = true;
        } else if (hasPropertyTermination()) {
          done = true;
        } else if (isNewState() && !isEndState() && !isIgnoredState()) {
          nextLevel.add(vm.getRestorableState());
          notifyStateStored();
        }
        notifyStateProcessed();
        backtrack();
        notifyStateBacktracked();
      }
    }
    currentLevel = nextLevel;
    depth++;
    if (depth >= MAX_DEPTH) {
```

```
      notifySearchConstraintHit("depth limit reached");
      done = true;
    }
  }
  notifySearchFinished();
}
```

The above `search` method is quite different from the one in the class `BFSHeuristic` of the package `gov.nasa.jpf.search`.

Chapter 10

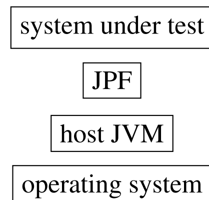
Implementing a Property

Chapter 11

Handling Native Methods

Every JVM includes a native method interface (JNI) that allows Java applications to invoke native methods, that is, methods that are implemented in a language other than Java but that are invoked from the Java application). This feature allows programmers to use code that has been already implemented in other languages. In some cases, accessing code such as C and C++ from applications written in Java can increase the performance. Another advantage of JNI is that it can be used when Java does not support certain platform-dependent features. Many of the classes of the Java standard library include invocations of native code.

The core of the JPF is a JVM that is able to execute all of the bytecode instructions that are created by a Java compiler. JPF itself is written in the Java programming language. That means that JPF is running on top of another JVM which we call it the host JVM. The following figure demonstrates the different layers that are involved in model checking a system under test using JPF.



Since the core of the JPF is a JVM that executes only Java bytecode instructions, it is not able to execute native methods. Consider, for example, the following application.

```
public class Sine {
  public static void main(String[] args) {
    System.out.println(StrictMath.sin(0.3));
  }
}
```

If we verify the above code, JPF reports the following error.

```
====================================================== error 1
gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
java.lang.UnsatisfiedLinkError: cannot find native java.lang.StrictMath.sin
at java.lang.StrictMath.sin(no peer)
at Sinus.main(Sine.java:3)
```

Its verification effort fails as it encounters the native method `sin` of the class `StrictMath`. However, JPF provides two different mechanisms, which can be combined, to handle calls to native methods. We will discuss these in the next sections. We will also present an extension of JPF, called jpf-nhandler[1]. It automates the use of both mechanisms to handle calls to native methods. The only thing left to the developer is the application configuration file.

[1] `bitbucket.org/nastaran/jpf-nhandler`

11.1 Model Classes

One way to handle native calls in JPF is using model classes. These are the special classes that are considered as part of the system under test. The model classes are verified by JPF and they are completely unknown to the host JVM. By implementing model classes, we force JPF to use an alternative version of certain Java classes instead of the original ones. More specifically, if there exists a model class for some class, JPF loads and uses the model class instead of the class itself. Therefore, we can make JPF not verify certain classes, and use the model classes as alternatives.

Recall that the `StrictMath` class includes the `sin` method, which is defined as native. One way handle the `sin` native call is to create a model class `StrictMath` that implements the `sin` method in pure Java. Since the original class `StrictMath` is part of the package `java.lang`, our model class is part of that package as well. To implement the `sin` method in pure Java, we can use, for example, Bhaskara I's sine approximation formula as follows.

```
package java.lang;

public class StrictMath {
  public static double sin (double a) {
    return 16 * a * (Math.PI - a) / (5 * Math.PI * Math.PI - 4 * a * (Math.PI - a));
  }
}
```

Note that we do not need to implement all methods of the `StrictMath` class. After we have implemented this model class, we still have to ensure that JPF never loads and verifies the standard class `StrictMath`. This is accomplished by adding the model class `StrictMath` to JPF's classpath.

```
target=Sine
classpath=C:/Users/franck/workspace/examples/bin
```

In this case, the directory `C:\Users\franck\workspace\examples\bin\java\lang` should contains the file `StrictMath.class`.

11.2 Native Peers

JPF's *model Java interface* (MJI) can be used to transfer the execution from JPF to the host JVM. The so called *native peer* classes play a key role in MJI. JPF uses a specific name pattern to associate the native peer classes and their methods with the corresponding classes and methods. For example, the native peer class associated with `sun.misc.Unsafe` is named `JPF_sun_misc_Unsafe` (see Figure **??**). Whenever JPF gets to a call associated with a native peer method, it delegates the call to the host JVM. Hence, the native call is not model checked, which is impossible since JPF can only handle Java bytecode, but executed on the host JVM. Great care has to be taken when developing a native peer class. For example, since classes and objects are represented differently in JPF than in an ordinary JVM, in a native peer class one often has to translate from the one representation to the other and back.

Chapter 12

Testing Extensions

Bibliography

[BBF+01] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, Philippe Schnoebelen, and Pierre McKenzie. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer-Verlag, 2001.

[BJC+13] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. Reversible debugging software. Technical report, Cambridge University, Cambridge, United Kingdom, January 2013.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[CDH+00] James Corbett, Matthew Dwyer, John Hatcliff, Shawn Laubach, Corina Păsăreanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from Java source code. In Carlo Ghezzi, Mehdi Jazayeri, and Alexander Wolf, editors, *Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448, Limerick, Ireland, June 2000. ACM.

[CGP01] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, 2001.

[CLRS09] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge MA, USA, 3rd edition, 2009.

[RDH03] Robby, Matthew Dwyer, and John Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 267–276, Helsinki, Finland, September 2003. ACM.

[VHB+03] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003.

[VHBP00] Willem Visser, Klaus Havelund, Guillaume Brat, and Seungjoon Park. Model checking programs. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, pages 3–12, Grenoble, France, September 2000. IEEE.