Java PathFinder: a tool to detect bugs in Java code

Franck van Breugel

January 12, 2016

©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. To illustrate the limitations of testing when it comes to nondeterministic code, let us concentrate on the former.

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 $[0, 99999]$ it may take many executions before encountering the exception. If we execute the application one million times, there is still a 36% chance that we do not encounter the

[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

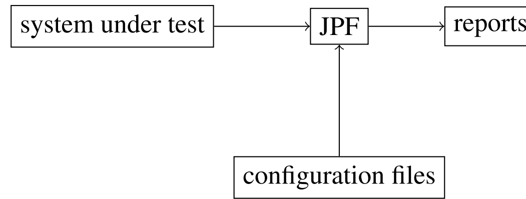


Figure 1: Overview of JPF.

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 .

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. In Chapter we focus on the configuration of JPF.

[6]The probability of choosing zero is $\frac{1}{1000000}$. The probability of not choosing zero is $1 - \frac{1}{1000000} = \frac{99999}{1000000}$. The probability of not choosing zero one million times in a row is $(\frac{99999}{1000000})^{1000000} \approx 0.36$.

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. In Section 1.1, we start with the simplest way to install JPF: just install the JPF binaries. This approach is ideal if one just wants to try JPF. In Section 1.2, we describe how to install the JPF sources. The main advantage of this second approach over the first approach is that one can easily make changes to JPF.

However, since changes are made to JPF almost every week, it is better to obtain its souces from JPF's Mercurial repository if one wants to use JPF regularly. Mercurial is a version control system. Information about Mercurial, including instructions how to install Mercurial, can be found at mercurial.selenic.com. 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 1.3 (and 1.4), 1.5 (and 1.6), and 1.8 (and 1.9), 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 1.7 and 1.10, respectively. How to use these plugins to run JPF is discussed in Chapter 2.

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 1.11 we will discuss how to install such an extension.

1.1 Installing Binaries

To install the JPF binaries, follow the seven steps below.

1. Create a directory[1] named `jpf`.

2. From the URL babelfish.arc.nasa.gov/trac/jpf/attachment/wiki/projects/jpf-core/ download the latest binary snapshot. The file is named `jpf-core-r`*dddd*`.zip`, where *dddd* are four digits. Save this file in the `jpf` directory.

3. Extract all files from the zip file `jpf-core-r`*dddd*`.zip` into a subdirectory of `jpf` named `jpf-core`.

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

5. 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`.

6. Add the path of the `jpf` command to the system environment variable `PATH` as described in Section 1.1.2.

7. Create the `site.properties` file as described in Section 1.1.3.

[1] Directories are also known as folders.

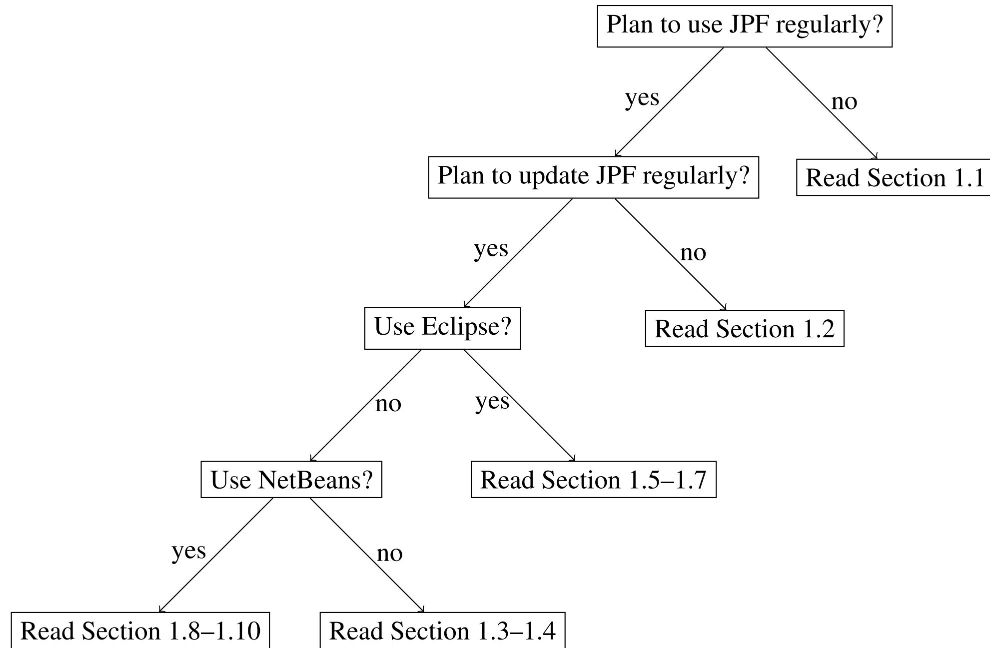


Figure 1.1: Road map for Section 1.2–1.10.

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 Adding to the System Environment Variable PATH

Linux

Add to the system environment variable named PATH the directory of the JDK 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.

1.1.3 Creating the site.properties File

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

```
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*[2].

3. Create in the directory *user.home*/.jpf a file named site.properties[3]. 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.

[2] 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.

[3] 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.

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

```
# 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 Installing Sources

To install the JPF sources, follow the eight steps below.

1. Create a directory named `jpf`.

2. From the URL babelfish.arc.nasa.gov/trac/jpf/attachment/wiki/projects/jpf-core/ download the latest sources snapshot. The file is named `jpf-core-r`*dddd*`-src.zip`, where *dddd* are four digits. Save this file in the `jpf` directory.

3. Extract all files from the zip file `jpf-core-r`*dddd*`-src.zip` into a subdirectory of `jpf` named `jpf-core`.

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

5. Run ant as described in Section 1.2.1.

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 1.1.2.

8. Create the `site.properties` file as described in Section 1.1.3.

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

1.2.1 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.

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 `bin/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 `bin\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
```

1.3 Installing Sources with Mercurial

How to install Mercurial is beyond the scope of this book. We refer the reader to mercurial.selenic.com. We assume that the path to the `hg` command is already part of the system environment variable `PATH` (see Section 1.1.2). 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. Run ant as described in Section 1.2.1.
5. 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`.
6. Add the path of the `jpf` command to the system environment variable `PATH` as described in Section 1.1.2.
7. Create the `site.properties` file as described in Section 1.1.3.

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

1.4 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 1.2.1.

1.5 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 mercurial.selenic.com, 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 1.1.2). 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. 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
```

6. Create the `site.properties` file as described in Section 1.1.3. 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}
```

1.6 Updating Sources with Mercurial within Eclipse

Simply build the project `jpf-core` again.

1.7 Installing JPF Plugin for Eclipse

As we will discuss in Chapter 2, the JPF plugin can be used to run JPF within Eclipse. This plugin can be installed as follows. In Eclipse, select the "Help" menu and its "Install New Software..." submenu. Click the "Add" button and enter `http://babelfish.arc.nasa.gov/trac/jpf/raw-attachment/wiki/install/eclipse-plugin/update/` as the "Location." Click the "Select All" button and subsequently the "Next" button.

1.8 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 mercurial.selenic.com, 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 1.1.2). 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. In NetBeans, build the project `jpf-core`. For more details, do a web search for how to build a project in NetBeans.

5. Create the `site.properties` file as described in Section 1.1.3. 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}
```

1.9 Updating Sources with Mercurial within NetBeans

Simply build the project `jpf-core` again.

1.10 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.

1.11 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`.

1.11.1 Installing Binaries

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

1. From the URL babelfish.arc.nasa.gov/trac/jpf/attachment/wiki/projects/jpf-numeric/ download the latest binary snapshot. The file is named `jpf-numeric-r`*dd*`.zip`, where *dd* are two digits. Save this file in the `jpf` directory.
2. Extract all files from the zip file `jpf-numeric-r`*dd*`.zip` into a subdirectory of `jpf` named `jpf-numeric`.

1.11.2 Installing Sources

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

1. From the URL babelfish.arc.nasa.gov/trac/jpf/attachment/wiki/projects/jpf-numeric/ download the latest source snapshot. The file is named `jpf-numeric-r`*dd*`-src.zip`, where *dd* are two digits. Save this file in the `jpf` directory.
2. Extract all files from the zip file `jpf-numeric-r`*dd*`-src.zip` into a subdirectory of `jpf` named `jpf-numeric`.
3. Run ant as described in Section 1.2.1.

1.11.3 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 1.2.1.

1.11.4 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 1.2.1.

1.11.5 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`.

1.11.6 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.

1.11.7 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.

1.11.8 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
```

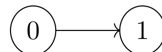
[1]Although the name of the application properties file does not have to match the name of the Java application—we could have called it, for example, `Test.jpf`—we will use that convention in this book.

```
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 1.7). 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 1.5, 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 1.10). 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 1.8, 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
```

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.

[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.