Using JPF/APROP for Infinite Mario

Chris Lewis and Ben Samuel and Jaeheon Yi

November 13, 2009

1 Introduction

Java PathFinder (JPF) is a stateful software model checker [9] developed at NASA AMES Research Center. It is a *software model checker* because it systematically explores all paths and inputs directly for software; it is *stateful* because it saves the program state after every logical operation.

Although JPF is known as a model checker, it is used more often as a systematic testing and analysis framework. Compared with simpler abstractions often used for hardware model checking, software is much more complex with potentially exponentially many more states; this quality makes exhaustive verification a difficult goal to attain. However, the methodical manner in which JPF generates program runs makes it ideal for testing programs; this compares much more favorably than with ad-hoc testing, which typically may only explore a fraction of the total state space without ever reaching a failure state.

JPF is also a powerful analysis framework; for example, one can implement data race and deadlock detection algorithms and have JPF run over different runs to find potentially disastrous program states.

JPF is a mature system with many extensions; for example, the core JPF system has been combined with symbolic execution [1] and the Bandera model constructor [3] to create a richer feature set for testing and verification. JPF has been used to find real errors in Remote Agent Spacecraft Controller, a mission critical software component that had previously deadlocked during operation in space [9]. It has also been used to check a complex traffic model with autonomous cars [5].

1.1 Modeling Language

The JPF system does not require a separate modeling phase; software is checked directly with a custom JVM. We believe this is an advantage, due to the absence of a learning curve for a discrete modeling language. However, to achieve proof of verification, one may need to abstract away much of the program state; in this case, the absence of a modeling language may hinder such a goal.

1.2 Specification Language

The JPF/APROP project [4] enables creating program property specifications using Java annotations, with listener programs to check that these properties hold during program execution. Some annotations that are currently supported include:

`@Nonnull` - check return values and field for null value

`@Const` - check for object modification within scope

`@GuardedBy` - check if object is guarded by specified lock

`@Requires, @Ensures, @Invariant` - check pre- and post- conditions and invariants

Since the specifications are largely based on Java annotations, we expect that these are not difficult to learn or use; furthermore, Java annotations support the Java Modeling Language [2], a behavioral interface specification language, which can be used in a design-by-contract approach.

2 What We Tried

JPF is distributed as a single core, `jpf-core`, with a series of sub-projects build on top of it. We were interested in `jpf-aprop`. All of the sub-projects that have been worked on include a series of example files, which appear to be used for pseudo-documentation in lieu of actual manuals or tutorials.

We created a plan to learn about JPF/APROP:

- Download JPF from the Mercurial repository and build the core and relevant subsystems
- Install the Eclipse plugin for rapid development
- Verify that the examples function as expected
- Use JPF/APROP on a small video game called Infinite Mario

2.1 Ease of Use

Downloading the repository and the Eclipse plugin went as well as could be hoped for; as can be expected from a small open-source research project, there were some quirks (such as building each project individually, and a few bugs in the Eclipse plugin).

2.2 Verifying the examples

The `jpf-aprop` package comes with some small examples that illustrate various functionality. A JPF/APROP example contains one Java file, and one file with configuration project-specific configuration parameters, commonly referred to as a JPF file. It is worth noting that it is not necessary to have a JPF file in order to run a project with the JPF JVM, as long as you specify the correct parameters on the command-line. However, the Eclipse plugin relies on JPF files in order to verify a program, and we have not encountered an example from the developers which have not utilized a JPF file.

An example JPF file from the JPF/APROP project can be found in Figure 1, and a code extract from the Java file it operates on in Figure 2.

We were able to verify the examples correctly, with an assertion exception being thrown when the non-null annotations were violated.

```
target = NonnullViolation

listener.autoload=\
  javax.annotation.Nonnull

listener.javax.annotation.Nonnull=.aprop.
    listener.NonnullChecker
```

Figure 1: A JPF file that registers the detection `@NonNull` violation. Here we see the target class to be tested, and a listener defined to listen for a specific annotation.

```
@Nonnull String id;

NonnullViolation (){
  id = null
}

@Nonnull
Object giveMeSomeObject() {
  return null;
}
```

Figure 2: A Java extract showing two violations of the @NonNull annotation. The first violation is the assignment of `id` to null, even though `id` must remain non-null. The second violation is the `giveMeSomeObject()` method, that returns null even though the annotation claims it won't.

Of interest is that it is not the fact that an uncaught exception is thrown that causes JPF to complain of an error, it is that a *listener*, `gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty`, is defined to throw an error when an uncaught exception is thrown. If you disable this listener, much of the JPF functionality is disabled (including this example), and JPF will report a verified error-free execution, even though it is not.

```
gov.nasa.jpf.jvm.
    NoUncaughtExceptionsProperty
java.lang.UnsatisfiedLinkError: java.awt.
    image.ColorModel.initIDs()V (no peer)
    at java.awt.image.ColorModel.<clinit>(
        ColorModel.java:197)
        at java.awt.image.BufferedImage.<
            clinit>(BufferedImage.java:277)
    at com.mojang.mario.Scale2x.<init>(
        Scale2x.java:45)
```

Figure 3: An error thrown by JPF when trying to load an image

2.3 Implementing annotations with Infinite Mario

Infinite Mario [6] is a small video game written in Java. We chose to analyze this code as Chris had significant experience with the codebase, and it represents a simple, yet non-trivial, example program.

It is here that our problems with JPF began. We were consistently encountering an error message (Fig. 3) claiming that there was an uncaught UnsatisifedLinkError inside the Java APIs themselves. This error occurs when attempting to load an image file from disk. Wrapping the original call in the Infinite Mario code with a `try/catch` did not alleviate the problem, and we were unable to coerce JPF to continue execution. At this point, we decided to refocus our efforts on trying to ascertain the limitations with JPF than attempting to continue with new examples. It was clear that there was a fundamental issue with JPF that we had not anticipated.

2.4 Finding the limitations

We decided to return to the original papers and presentations of JPF, as the web site was short on real documentation. We found a presentation from one of the JPF developers, Willem Visser, which was presented at UC Santa Cruz in 2004 [8]. The important quote is "Handle full Java language, mostly only for closed systems, cannot handle native code, no input/output through GUIs, files, Networks, must be modeled by java code instead." This is where our problems lied: Infinite Mario was attempting to open an image file from disk, and JPF would not allow it. We discuss these problems more widely in Section 3.2.

3 What We Learned

3.1 Quality and Ease of Use

Our initial impression of JPF was of a mature software system, able to be used for heavy-duty verification purposes as well as comprehensive testing.

3.2 Limitations

Although quite powerful, JPF has some limitations, some more serious to us than others. For example, JPF cannot track JNI code [7], and requires developers to "mock up" the native code that gets executed as Java code. Another issue we encountered is that JPF requires a closed system to be tested. Although our Infinite Mario game already had a testing harness to automatically play the game, for other projects it represents a potentially significant undertaking to create such a testing harness for closing the system. Many projects have entangled input/output, as Infinite Mario did, and it may be impossible to run JPF with these projects without significant architectural changes.

It appears that JPF is not a tool that should be run post-development, but actually planned for at the beginning. While this seems stringent, it seems that enforcing such restrictions to allow JPF interoperability will encourage better software architecture, removing entanglement with other systems, or at least making it easier to mock the system for a test harness.

4 Conclusion

References

[1] Saswat Anand, Corina S. Pasareanu, and Willem Visser. Jpf-se: A symbolic execution extension to java pathfinder. In Orna Grumberg and

Michael Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 134–138. Springer, 2007.

[2] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of jml tools and applications. *STTT*, 7(3):212–232, 2005.

[3] Matthew B. Dwyer. Software model checking: The bandera approach. In Bart Jacobs and Arend Rensink, editors, *FMOODS*, volume 209 of *IFIP Conference Proceedings*, pages 3–4. Kluwer, 2002.

[4] Java PathFinder APROP Project, 2009. `http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-aprop`.

[5] Doron Peled and Yih-Kuen Tsay, editors. *Automated Technology for Verification and Analysis, Third International Symposium, ATVA 2005, Taipei, Taiwan, October 4-7, 2005, Proceedings*, volume 3707 of *Lecture Notes in Computer Science*. Springer, 2005.

[6] Markus Persson.

[7] Sun Microsystems. Java Native Interface, 2009. `http://java.sun.com/javase/6/docs/technotes/guides/jni/index.html`.

[8] Willem Visser. Symbolic execution and test-input generation.

[9] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.