Evaluating Java PathFinder on Log4J

David A. Dickey, B. Sinem Dorter, J. Michael German,
Benjamin D. Madore, Mark W. Piper, Gabriel L. Zenarosa

Master of Software Engineering Program
School of Computer Science
Carnegie Mellon University

ddickey@andrew.cmu.edu, bsd@andrew.cmu.edu, jgerman@andrew.cmu.edu,
madorb@cs.cmu.edu, mwpiper@cmu.edu, gzen@cs.cmu.edu

1. Introduction

Advances in finite-state model checking have made possible initial successes in tool-based checking of complex, realistic, multi-threaded programs, without the need for specialized modeling languages. In this paper, one such tool is evaluated through verifying the presence of known bugs in the 12,546-line Log4J Java library. The model checker evaluated is NASA Ames Research Center's recently released Java PathFinder (JPF).

First, similar model checking tools will be considered. An introduction to the approach, features, and extensibility mechanisms used by JPF will be followed by simple examples of the applications of JPF. JPF will then be evaluated in detail through assessing its ability to confirm six known bugs in Log4J involving uncaught runtime exceptions, unclosed resources, and threading issues. Lastly, reflections upon the capabilities of the tool will be given.

1.1 An Introduction to Log4J

Log4J is a mature open-source project, started in 1996 and widely used throughout industry today, providing advanced logging capabilities in the form of a Java library. Among the unique features of Log4J are its hierarchical logging "categories," which allow categories of output to be enabled or disabled, formatted, and sent to different output sources, even at runtime. Different output sources are supported by allowing loggers to write to different types of appenders,[1] including file, screen, archive, socket, event log, SMTP, and multi-threaded asynchronous messaging appenders. Configuration occurs dynamically via local files, or configurations stored on remote servers.

2. Selection of a Model Checking Tool

While Log4J provides a useful example of a realistic project used in industry, it was at first unclear which model checkers would be appropriate for checking Log4J. Although PathFinder is the tool used in this evaluation, it is only one of several available model checkers. Two alternative tools that were considered for verifying defects in Log4J were

[1] An appender allows you to specify destination for log data and provide a custom layout for it, through which all log messages will be formatted [www.mojavi.org/docs/api/3.0/mojavi/logging/Appender.html].

Bandera, and Crystal/Fluid. This section describes the reasons why these tools were not chosen.

2.1 Bandera

Bandera is a toolset for model checking (developed at Kansas State University's SAnToS laboratory) that focuses on checking concurrency issues [3]. Bandera constructs abstracted models from Java code using a combination of program slicing techniques and a rule-based abstraction engine [4]. These intermediate abstracted models can then be automatically converted into verifier-specific languages, which will be used by the different model checking engines (such as SPIN, SMV, and SAL).

There are, however, several reasons why Bandera was not chosen for verifying Log4J.

- **Installation —** Due to the fact that Bandera cannot function without other model-checking engines, installation of Bandera is potentially difficult.
- **Maturity —** Although Bandera has been in development for over 5 years, the code was branched after the 0.2 release to focus on a re-implementation of the entire tool [6], featuring the object-oriented Bogor model checker as a backend, and a set of Eclipse plugins as the front-end [5]. Only a very limited command-line prerelease version of this re-implementation (which is planned to eventually become the 1.0 version) is available. In the meantime, the only realistic option is to use the abandoned (and relatively immature) 0.3 release.
- **User Interface —** Bandera 0.3's user interface is very outdated. While this is not a problem of functionality, in evaluation, it was difficult to determine the types of errors it reports.
- **Learning Curve —** Bandera's learning curve is considerable. Users have to understand not only the Bandera abstraction system itself, but also the backend model checkers. Since no analysis capabilities are included as 'out-of-the-box' functionality, it is a somewhat daunting task to discover how to use the tool.

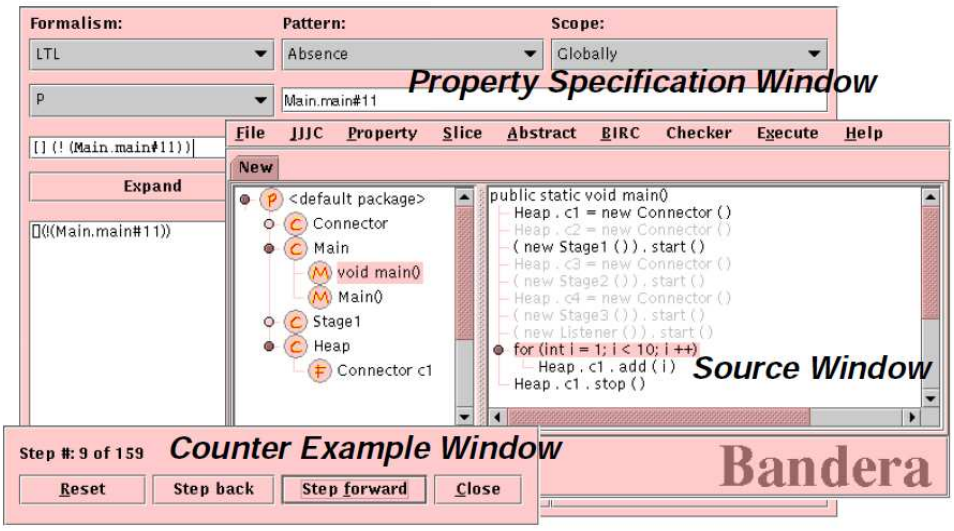

Figure 1: Bandera's User Interface, Version 0.3
(borrowed from *Bandera: Extracting Finite-state Models from Java Source Code* [4])

2.2 Crystal/Fluid

Crystal/Fluid is a model-checker that includes a robust annotation-based concurrency verifier (including an Eclipse GUI), and a framework for the creation of other analyses based on the Java abstract syntax tree. While this tool was considered for evaluation, it has already been used to check Log4J for concurrency errors. Rather than repeat this task, it was felt it would be more beneficial to assess JPF's ability to detect similar errors as found with Crystal/Fluid.

3. Installation of Java PathFinder (JPF)

In this section we discuss the installation of JPF for both the advance version and the open source version since they both presented a challenge. We conveyed these challenges to our contact at NASA Ames, Willem Visser, and sent him our step-by-step directions so that he may make them available.

3.1 Installation of the Advance Copy of JPF

Before the open source release of JPF, we were provided with an advance copy that could be built from the command line using the Ant tool from apache.org. Several prerequisite libraries had to be downloaded in order to successfully install JPF. JPF requires Java SDK 1.4.*, as well as the following third-party libraries [1]:

- BCEL – the Bytecode Engineering Library to load classfiles;
- Xerces – the XML parsing library to parse execution paths stored in XML, and;
- The MD5 library (from Timothy W. Macinta) to efficiently build MD5 state hash codes.

All of these libraries must be included within the CLASSPATH, even though they can reside outside of the JPF directory tree. In addition, the following build tools must be installed [1]:

- JUnit – the Java regression test framework to run unit tests, and;
- Ant – the Java build system to resolve dependencies and compile JPF sources.

After completing installation prerequisites, we used Ant to manually build JPF. We appended the location of the Ant script to our PATH, and then added ant.jar to our CLASSPATH. (The other prerequisite libraries were already in our CLASSPATH.) Then, we switched to the directory where we installed Java PathFinder, and ran Ant to compile all JPF sources and run the regression tests.

3.2 Installation of the Open Source Version of JPF

We decided to install JPF from within the Eclipse IDE since we already had some of our test environments setup within the IDE. However, setting up the open source version of JPF to run within the Eclipse platform ended up being quite a challenge. The usually bundled *.project* file was removed from the package (according to an entry in their open forums) because it referenced libraries that were not prerequisites of the current JPF version. It should have been easy for us to simply create our own project and checkout the CVS project for JPF using the Eclipse IDE; however, the *pserver* connection to the repository could not be established from within the IDE.

Additionally, in contrast to the installation of the advance version, the required libraries were not bundled in this version. To make the installation process less cumbersome, we simply reused the libraries from the advance version. (Since then, the required libraries have been packaged and made available on the JPF SourceForge website.)

We settled on the following steps to install JPF to run within the Eclipse platform. We assigned one person to perform the following steps and export the baseline JPF project for the rest of the group to use.

1. Checkout (using the system shell) the CVS project for JPF into a location outside of the Eclipse workspace folder.
2. Create a Java project (e.g., *javapathfinder*) on Eclipse.
3. Copy the CVS folder from the root directory of the checked-out JPF (in step 1) into the Java project (in step 2).
4. Create the following source folders manually:
   a. src
   b. extensions/LTL2Buchi/src
   c. env/jpf
   d. examples
   e. test

   Note: These have to be created manually; otherwise, the CVS *update* function in Eclipse will create them as packages rather than source folders.
5. Update the Java project (using the Eclipse CVS *update* function).
6. Create a lib/ folder under the Java project.
7. Download the prerequisite JAR files (bcel.jar, xercesImpl.jar, xmlParserAPIs.jar, junit.jar, fast-md5-2.5.zip) into the lib/ folder (in step 6).
8. Build the project in Eclipse.
9. Test it using the examples.

4. Java PathFinder (JPF) Explained

JPF is a NASA-developed software model checker that explores all possible execution paths of a Java program in order to find property violations. It does this by exhaustively simulating the program in a custom Java Virtual Machine (JVM) while property-checking modules observe the execution for certain conditions. When a property violation is discovered, JPF outputs the violation information and the specific trace that lead to it. The out-of-the-box property checking modules handle deadlocks, uncaught exceptions, and assertion errors.

Because JPF performs a simulation of the program, it requires a main function. JPF is then able to use this main function as the root of all possible execution paths. This approach gives JPF advantages over other model checking approaches. Consider the following trivial example:

Code Listing 1: An example of JPF's path exploration

```
public void someFunction(boolean arg)
{
  if (arg == true) { return; /* Branch that executes normally */ }
  else             { throw new Exception("Branch that crashes the code."); }
}
```

By utilizing simulation starting from the main function, JPF is capable of analyzing only those paths that are actually executed in the program. If the only calls to `someFunction` occur using `true` as the argument, JPF will only need to analyze the `true` branch. However, if JPF is unable to determine the argument to `someFunction`, it will analyze both branches of the function.

Like other model checkers, JPF suffers from state space explosion—the model exponentially grows with respect to the size of the program—making it difficult to model large programs. The recommended size of programs to be modeled is below 10,000 lines of code, although it has been successfully executed on larger programs. In order to counter the explosion effect, JPF uses several techniques to improve efficiency.

Backtracking – Since JPF checks all paths in a program, it must be able to revisit previous branch points. Instead of re-traversing the execution path to recreate a branch point, JPF is able to "undo" previous decisions.

State Matching – In order to save unnecessary work, JPF remembers the previously visited execution states, and when it detects an already visited state, it skips it and backtracks to the next non-deterministic decision.

Heuristic Choice Generator – In order to simplify the evaluation of complex data types, simplified evaluation techniques can be added. For example, a non-deterministic floating point value that is passed into a method would require that all possible values be evaluated. This however would be very difficult to model due to the number of combination. Instead, a small number of pre-selected choices can be used to reduce the number of values to model.

Partial Order Reduction (POR) – When model checking a multithreaded program, all possible thread interleavings must be checked in order to verify all possible concurrent conditions. However, certain code segments are not preemptable by other threads, allowing the entire code segment to be interleaved instead of the sub elements. JPF uses this approach to reduce the number of thread interleavings to model.

State Abstraction – Instead of storing all heap, stack, and process information, JPF uses an abstracted model of these elements in order to reduce storage overhead and improve lookup time.

One problem with the JPF approach is that the JVM cannot observe what occurs within any native calls. Native calls are requests out of the control of the JVM to the operating system (OS) to perform OS level tasks, such as file access or network transactions. Since

JPF cannot reason about these calls it cannot model their behavior. Therefore explicit models must be created to abstract the behavior of the low-level native call. These models are created through the Model Java Interface (MJI), which is discussed in the next section.

NASA successfully used JPF to detect defects within their software and recently (April 26th, 2005) open-sourced the project with the hopes that the open source community will adopt JPF and create new uses for it [7]. By putting JPF in the hands of the community, more developers can create interesting model-checking add-ons, as well as continue to create the models required for the many unsupported native calls.

5. Extending JPF

"One can think of JPF as an execution system framework for all kinds of dynamic, runtime oriented verification purposes." [2] JPF Website

One of JPF's main qualities is its ability to be extended with new functionality. New functionality is not limited to verification of properties, but can be anything that desires to use JPF's systematic program execution approach. For example, it could be used to produce coverage metrics in order to show code that is traversed often.

Search- and VMListeners

JPF uses a Listener pattern that allows modules to subscribe to events in the virtual machine. When the event occurs the module is called allowing it to query the execution and states as well as allowing it to change the successive behavior of the JVM. This allows for powerful extensions to be added without the need to modify the internals of JPF.

This is the primary means of adding new evaluations to the JPF framework. New modules for checking conditions like race-conditions would use this listener approach in order to track the execution and to determine when a condition is met.

Model Java Interface (MJI)

JPF cannot reason about native calls and therefore must use models instead. These models are created using the MJI and are used instead of the actual native method. The model developer is responsible for abstracting the important characteristics of the native call and retaining it through the model. JPF can then reason about the behavior of the native call through the use of the model.

6. "Hello, World!" for JPF

One of the first example programs we tested using JPF is the test program shown below. This is an adapted test program from one of the sample tests of our Exceptions Analysis project (i.e., Project #1) for our course, *Analysis of Software Artifacts*. (Note that we implemented a `main(…)` function.)

Code Listing 2: Proj1Ex.java

```
import java.io.*;

public class Proj1Ex {
    public void foo(int i) throws FileNotFoundException {
        if (i>0)
            throw new RuntimeException();
        throw new FileNotFoundException();
    }

    public void bar(int i) throws RuntimeException {
        try {
            foo(i);
        } catch (IOException e) {
        }
    }

    public static void main(String[] args) {
        (new Proj1Ex()).bar(0);
    }
}
```

JPF does not report any unhandled exceptions in the program above. This is because the program does not execute the line of code that throws the unhandled `RuntimeException` due to the zero value passed to function `bar(…)`. Changing the parameter to `bar(…)` from `0` to `1` allows JPF to find the bug.

In an effort to have JPF report all possible unhandled exceptions for non-deterministic input, we changed the parameter passed to `bar(…)` from absolute values to randomized values, as shown below.

Code Listing 3: An attempt at non-determinism

```
bar((new java.util.Random(System.currentTimeMillis())).nextInt())
```

This, however, does not cause JPF to report all unhandled exceptions since it does not ask JPF to explore all paths. What this statement implies, rather, is that JPF will check a run of the test program using the specific randomized integer returned by the `Random(…).nextInt()` function. Therefore, for different verification runs of the program, JPF may or may not find the unhandled exception.

We emailed the question of how to enable JPF to explore all paths to our contact at NASA Ames, Willem Visser. He replied within a few hours with the answer: we must use the JPF function `Verify.random(int n)` method to instruct JPF to simulate execution using all values from 0 to n.

Code Listing 4: Proj1Ex.java Using Verify.random()

```
import java.io.*;
import gov.nasa.jpf.jvm.Verify;

public class Proj1Ex {
```

```
    public void foo(int i) throws FileNotFoundException {
        if (i>0)
            throw new RuntimeException();
        throw new FileNotFoundException();
    }

    public void bar(int i) throws RuntimeException {
        try {
             foo(i);
        }
        catch (IOException e) {}
    }

    public static void main(String[] args) {
        (new Proj1Ex()).bar(Verify.random(1));
    }
}
```

After making this change, JPF was able to find the unhandled `RuntimeException` thrown within function `foo(…)` as shown below.

JPF Output Listing 1: Project 1 Unhandled Exception

```
java.lang.RuntimeException
  at examples.Proj1Ex.foo(examples\Proj1Ex.java:15)
  at examples.Proj1Ex.bar(examples\Proj1Ex.java:22)
  at examples.Proj1Ex.main(examples\Proj1Ex.java:28)

---------------------------------- path to error (2 steps):
Step #0 Thread #0
  examples\Proj1Ex.java:28        (new Proj1Ex()).bar(Verify.random(1));
  examples\Proj1Ex.java:9 public class Proj1Ex {
  examples\Proj1Ex.java:28   (new Proj1Ex()).bar(Verify.random(1));
Step #1 Thread #0 Random #1
  examples\Proj1Ex.java:28   (new Proj1Ex()).bar(Verify.random(1));
  examples\Proj1Ex.java:22      foo(i);
  examples\Proj1Ex.java:14      if (i>0)
  examples\Proj1Ex.java:15                  throw new RuntimeException();
---------------------------------- end error path



==================================
  1 Error Found: uncaught exception
==================================

---------------------------------- thread stacks
---------------------------------- end thread stacks
```

7. Log4J Bugs to be checked

In our evaluation of JPF we decided to examine 6 known bugs in various versions of Log4J. As Log4J is a very mature project, we were not hopeful that we would be able to find new bugs in the project; thus, we located the bugs by examining the Log4J bug database for verified and resolved bugs. We then attempted to use JPF to detect these

bugs by obtaining the appropriate Log4J source code both before and after the bug was fixed. Log4J is very specific in that it provides the user with the following contract: "Log4J will not throw unexpected exceptions at run-time potentially causing your application to crash." [8] This means that Log4J should never allow an exception that is not explicitly declared as being thrown to escape from its code. This is important to our discussion as one of the selected bugs is a violation of this contract.

7.1 Bug Types

We identified 3 different types of bugs in the Log4J sources, these were:

1. Unclosed Resource Bugs - 1 bug;
2. Exception contract violations - 1 bug, and;
3. Threading issues (both deadlock and race-conditions) - 4 bugs.

7.2 Bug 11186

7.2.1 Bug Description

This bug is a result of violating the Log4J exception contract. It arises when a user extends a Logger class, and inserts code that might throw a runtime exception. While this is a bug that arises from user behavior, it still violates the contract of Log4J, and is something that might easily occur in practice.

7.2.2 Experiences

Initially when we ran JPF upon the code, JPF would crash and report an `ArrayIndexOutOfBoundsException`. This actually turned out to be coming from the lack of an MJI for code within `java.net`; however, no error message was given to indicate this was the source of the error, and the stack trace was from seemingly unrelated code. To be truly useful JPF must provide better errors for such "easily" determined things as a missing MJI. After commenting out the code that referred to elements of the `java.net` package, JPF correctly reported an uncaught exception error.

7.3 Bug 7793

7.3.1 Bug Description

This bug occurs when a stream is opened on a `java.net.URL` object in one piece of code, and this is then sent to a `PropertyConfigurator` that uses the stream but never closes it.

7.3.2 Experiences

As this bug uses the `java.net` package, and thus cannot currently be checked with JPF, we attempted to create a similar example using an `InputStream`. In attempting to verify the bug in this manner, we actually inadvertently stumbled across a bug in JPF.

Code Listing 5: JPF Bug

```
public class StringError {

  public static void main(String[] args) {
        byte buff[] = {97, 98, 99, 65, 66, 67};
```

```
        String s = new String(buff);        //Error!!
        System.out.println("Ouput: " + s);
  }
}
```

When running JPF on the preceding code, JPF crashed throwing another `ArrayIndexOutOfBoundsException`. We submitted this bug to Willem Visser and he replied indicating that there is a known bug in JPF relating to its use of `ThreadLocal`. He also told us that our example was the most succinct code that exhibited this behavior and would be very helpful in resolving the bug.

Eventually though, we were able to create a simple example of an unclosed resource, and were able to use JPF and its ability to do assertion checking to locate the bug.

Code Listing 6: Unclosed Resource Bug

```
import gov.nasa.jpf.jvm.Verify;

public class OpenClose {
  public static void main(String[] args) {
        Closeable c = new Closeable();
        int rand = Verify.random(2000);

        if (rand == 1327) {
              //do nothing
        } else {
              c.close();
        }
        assert (c.isClosed == true) : "It is not closed";
  }
}

class Closeable {
  public boolean isClosed = true;

  public Closeable() {
        isClosed = false;
  }

  public void close() {
        isClosed = true;
  }
}
```

While this is certainly a contrived example, it demonstrates one of the uses of JPF: checking for assertion errors. We again used the `Verify.random(int n)` method to instruct JPF to simulate execution using all values from 0 to n, which simulates non-deterministic behavior such as user-input. In this example we see that we close the Closeable resource on every possible execution path except for one. This type of defect is very difficult to locate during testing (assuming a non-trivial example). Although there is only one instance where the assertion will fail, we are easily able to detect this using JPF.

JPF Output Listing 2: Unclosed Resource Assertion Check

```
java.lang.AssertionError: It is not closed
at OpenClose.main(OpenClose.java:14)

---------------------------------- path to error (2 steps):
Step #0 Thread #0
  OpenClose.java:4     public class OpenClose {
  OpenClose.java:6     Closeable c = new Closeable();
  OpenClose.java:21    public Closeable() {
  OpenClose.java:19    public boolean isClosed = true;
  OpenClose.java:22    isClosed = false;
  OpenClose.java:23    }
  OpenClose.java:6     Closeable c = new Closeable();
  OpenClose.java:7     int rand = Verify.random(2000);
Step #1 Thread #0 Random #1327
  OpenClose.java:7     int rand = Verify.random(2000);
  OpenClose.java:9     if (rand == 1327) {
  OpenClose.java:14    assert (c.isClosed == true) : "It is not closed";
---------------------------------- end error path

---------------------------------- thread stacks
---------------------------------- end thread stacks


==================================
  1 Error Found: uncaught exception
==================================
```

The preceding is the output of JPF on the buggy code. It shows, not only the error found by JPF, but also a program trace that leads to the error.

7.4 Bug 1505

7.4.1 Bug description

Bug 1505 is a threading bug involving the implementation of an asynchronous append function within Log4J. Log4J exposes an append function that clients can call asynchronously. Inside Log4J is a background thread that performs the actual append to the log. The append function in the background thread contained the following:

Code Listing 7: Log4J's asynchronous appender thread waiting for buffer space before appending

```
if (bf.isFull()) {
  bf.wait();
}
// Append to the buffer bf below…
```

The code's intent is to check if what it is appending to, `bf`, is full or not. If `bf` is full the intent is to wait until it is no longer full. However, as this is an `if` statement rather then a `while` statement, it is not always the case that `bf` will be empty after the `wait`—the thread responsible for emptying the buffer may have been interrupted, but the thread waiting to append may have also been notified to proceed.

The example used to find this bug was a piece of code that consisted of three threads. The main thread infinitely generated asynchronous append calls; a background thread continually interrupted the main appending thread; lastly, a thread inside Log4J fulfills the asynchronous append request (a portion of this code is shown above). Running this example eventually fills `bf`. The next append call then results in Log4J's thread blocking upon `wait`. However, because of the interrupting thread, the code doing the append calls may be interrupted and the execution would then go back to Log4J, in the append, with `bf` still being full.

7.4.2 Bug experiences

To try and capture this behavior with Log4J we added an assert statement after the `if` statement.

Code Listing 8: Asynchronous appender with an assertion

```
if (bf.isFull()) {
  bf.wait();
}
assert !bf.isFull();
```

When running the code without JPF, eventually an assertion violation is raised; however the program continues to run endlessly after the assertion. The reason it continues to run endlessly is because of the interrupting thread. The Log4J thread dies from the assertion failure, the appending main thread stops (it does not have Log4J to call anymore), and because of this, the interrupting thread indefinitely continues to sleep, then interrupt the main thread, then sleep, and so on. This occurs because the interrupting thread is killed from a call in the main thread only after the bug's assertion is triggered. However, the placement of the assertion causes the main thread to never know about the bug, and thus, the main thread can never kill the interrupter thread.

Our first run with JPF raised two null pointer exceptions and an internal JPF null pointer exception. The internal null pointer exception was raised because JPF caught the 2 null pointer exceptions and raised an exception of its own. The null pointer was coming from a getMethod() call in ss.successor();. JPF could not find a method from the getMethod() call when trying to step to the next state or thread in its checking model. The specific call that caused this was `Category cat = Category.getInstance( "test_cat" );`

We debugged this and found a statement in Log4J that caused JPF to reach this null pointer error. It was a default that seemed to not be initialized when Log4J was run with JPF. Since the default was not needed it was commented out.

With the commented out version of JPF we got it to run on the code. However JPF was unable to identify the bug. It returned "No Errors Found." Looking into this we tried to generate a simple program that reproduced this error; eventually we found one. The program was one that looped forever in a main thread and eventually would fail an assert statement, generating an assertion error in a background thread. JPF could not find the assertion error in this case. However, if we added a print statement or sleep statement in the iteration of the loop in the main thread JPF would find it.

This bug was submitted and the reply we got was that it was a known bug, but not a documented one. "Ahh the Assertion check! Now there you stumbled on something I thought nobody would :-)" [Willem Visser, personal correspondance]. It seems that if there are only local transitions in a part of the code, the partial order reduction scheme will turn it into one mega-transition. A local transition here is anything that would not lead outside of the current thread or current function. Because of this, in the case of our code, the mega transition was an infinite loop and would only transition to itself in JPF's model. The background thread had the transition to an assertion error, but the model could never transition into it once it hit our mega-transition, as there was no way out of the infinite self loop. In the case of code with transitions that are not local, such as a system call or thread transition call, JPF will partial order reduce the code to still have transitions to other functions or threads. Thus, even though the infinite loop exists in both cases, only in the case where no local transitions exist will JPF be able to transition to the assertion error.

Our only option was therefore to try and find what code in Log4J (or in the test code for Log4J) caused the infinite loop mega transition. Using the process of elimination on the generated code, we were unable to identify the infinite loop. Next we looked inside Log4J to find the infinite loop, but the most logical places did not contain the error. The time it would have taken to test Log4J by trial and error (or examine all the parts of Log4J that were being accessed) would have been far outside our scope. Thus we stopped trying to use JPF to find this bug. We settled upon having this bug as an example of our problems with JPF.

7.5 Other Unresolved Bugs

We attempted to resolve a number of other Log4J bugs, but were unsuccessful in doing so. Attempting to locate the bugs using JPF resulted in various exceptions occurring in JPF itself. We submitted all found JPF bugs to Willem Visser and attempted to work around them as much as possible, but each inevitably led to irresolvable problems. For reference, the other Log4J bugs that we attempted to verify were: 1507, 1603, and 23912.

Although we were unable to validate the existence of all of the bugs that we set out to locate, we were able to contribute a number of JPF bugs to the JPF developers. According to Willem Visser, these bug reports will be very helpful in locating and removing the bugs, thus creating an improved version of JPF.

8. Conclusion

We evaluated three different model checking tools: Bandera, Fluid/Crystal, and JPF. We pursued the evaluation of JPF and attempted to locate known bugs in Log4J. During our evaluation we ran into a number of challenges, although none of which were insurmountable. We had difficulty installing the advance and open source versions of JPF. Some of these challenges (such as missing libraries) have since been corrected in the latest version on the JPF SourceForge website. Also, creating an Eclipse project for JPF was a complex task. Although we were hindered by a few bugs in JPF, we were able to submit useful bug reports to the project. Additionally, some of the JPF bugs were a

result of unimplemented MJI packages that the Log4J code used. The JPF developers hope that open-sourcing JPF will lead to an increased number of MJI implementations.

JPF is a very powerful framework that is likely to improve rapidly now that it is an open source project. While it still requires a little more work to become a tool for industrial use, we feel that it offers great potential.

Finally, we would like to thank Mr. Willem Visser who was extremely swift and helpful in responding to our numerous questions and bug submissions.

9. References

[1] Java PathFinder - http://javapathfinder.sourceforge.net/

[2] http://javapathfinder.sourceforge.net/$Extensibility.html

[3] http://bandera.projects.cis.ksu.edu/index.shtml

[4] *Bandera: Extracting Finite-state Models from Java Source Code*, James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, Hongjun Zheng in Proceedings of the 22nd International Conference on Software Engineering, June, 2000.

[5] http://bogor.projects.cis.ksu.edu/

[6] http://bandera.projects.cis.ksu.edu/roadmap.shtml

[7] JPF Open Source Press Release – http://www.nasa.gov/centers/ames/news/releases/2005/05_28AR.html

[8] http://logging.apache.org/log4j/docs/faq.html