A Survey on the various modules of Java Pathfinder

Tahmid Nabi

June 7, 2015

1 Introduction

Java Pathfinder[1](JPF) is a highly configurable software model checker for Java bytecode programs, which was developed at the NASA Ames Research Center. It is implemented as a drop in replacement for a normal Java virtual machine (VM).

Unlike a normal VM, which only executes one path through the program, depending on input data and scheduling choices, JPF systematically explores all possible data and scheduling combinations. JPF stores program states and is capable of detecting if states are equivalent, in which case it backtracks to a previously stored state and continues execution from there.

If JPF finds a property violation (defect), it not only reports the nature of the defect, but also the complete program trace the sequence of operations leading to this defect.

JPF starts by a executing a Java Program by following one possible execution path. Once JPF reaches the end of the program, it automatically starts looking for other explored choices. If it finds any, JPF backtracks to the corresponding state by reverting variable values and program counters, picks the next choice and continues from there. This process is repeated by JPF until no choices are left, or a property violation is detected.

JPF provides configurable extensions to define:

- operations that force the execution to branch (Choice Generators)
- code that should be executed outside JPF (Native Peers, e.g., for abstracting native libraries)
- code that allows changing the semantics of execution for bytecode instructions (Instruction Factories)

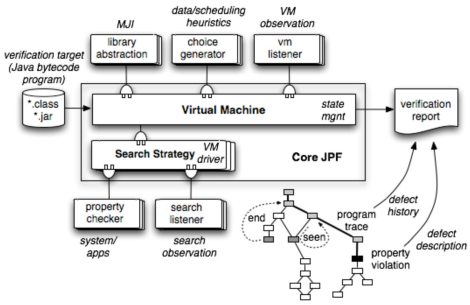

Figure 1: Structure of the JPF Model Checker

- code that non-intrusively monitors and controls JPF program execution (Listeners)
- properties to check for (such as no unhandled runtime exceptions)

Over the years, researchers have used these configurable extensions to build several additional modules of JPF that allows for verification and model of a variety of JVM-based programs, like concurrent programs[8], GUI-based applications[5], Actor programming model based JVM programs[3] and Android applications[9]. Researchers have also used the extensibility of JPF for other purposes like regression model checking[10] and UML state chart verification[4] as well. The rest of this survey paper briefly describes several such modules, their purpose, and how they utilized JPF to realize that purpose. The paper concludes by commenting upon some of the common patterns observed among these modules.

2 Regression Model Checking

Yang et al.[10] developed a new technique for **Regression Model Checking**(RMC), which applied model checking incrementally to new version of systems.

2.1 Methodology

Considering a program P, a modified version of it, P', and T, a test suite developed for P, regression testing can be defined as the process of using T to validate to validate P'.

Reusing all of T can be expensive, so regression test selection techniques(RTS) [6] use data on P, P' and T to select a subset of T, T', with

which to test P'. *Safe* RTS techniques guarantee that under certain conditions, test cases which were not selected could not have exposed faults in P'[6].

The main idea behind Yang et al.'s technique, RMC, is to reduce effort while still preserving error detection. RMC achieves this by avoiding checking some safer sub-state spaces determined using the data gathered in the preliminary safe period.

At the first step of RMC, a recording phase of RMC (implemented by the algorithm `recordingRMC`) gathers data from model checking an earlier version of the program.

At the second step of RMC, Yang et al. use a *safe* RTS technique, `Dejavu`[7]. `Dejavu` constructs control-flow graph(CFG) representations of the procedures in in P and P'. It utilizes information recorded by `recordingRMC`) to find *dangerous*edges that can lead to code that can cause program executions to exhibit different behavior.

In the third step of RMC, the `pruningRMC` algorithm takes these dangerous edges computed by `Dejavu` and information computed by `recordingRMC` as input, and uses these to determine which sub-state spaces are safe and prunes them. This mechanism thus reduces the effort to model check P'.

2.2 Implementation

RMC is implemented on top of JPF.

From section 1, we know that JPF provides **Listeners** to customize JPF execution. Yang et al. implement two listeners, `recordingRMCListener` and `pruningRMCListener` to implement RMC. The former computes a reachable elements map during JPF verification. `Dejavu` then uses this information to compute the dangerous elements information, the `pruningRMCListener` loads the stored reachable elements map and dangerous elements, and controls the verification to prune states by backtracking during the search.

The authors compared RMC with full model checking, and reported that while the recording phase led to increased cost, the cost savings in the pruning phase were more than enough to compensate for this.

3 jpf-concurrent

3.1 Problems with model checking Concurrent Programs

In multi-threaded programs, the concurrent actions can be executed in any order. Model checking such programs means considering all possible inter-

leavings of these concurrent actions. This can lead to a very large state space that requires too large memory and processing time. For concurrent applications the JDK provides concurrency utilities, the `java.util.concurrent` package[2]. Due to thread non-determinism from many Java Concurrency constructs, if JPF is used to model check java programs using these constructs, it leads to very large state spaces.

Another difficulty arises from the fact that different JDK vendors use native methods to implement `java.util.concurrent` package, which are tied to specific JVMs.

To address these issues, Ujma and Shafiei[8] created jpf-concurrent which abstracts classes from the Java Concurrency Utilities, minimizing the state-space explosion problem.

3.2 Implementation of jpf-concurrent

The Model Java Interface(MJI) component of JPF is used to transfer the execution from JPF down to the host JVM. All methods that execute on the host JVM using MJI are atomic and are not model checked by JPF and thus do not contribute to the state space explosion problem.

JPF also has model classes, which replace actual Java classes and forces JPF to model check the model classes themselves.

These two features are used by jpf-concurrent. Ujma and Shafiei[8] abstract the state of concurrent objects in JPF environment and keeps the actual state in the host JVM. by introducing model classes corresponding to the Java Concurrency Utilities classes. These JPF-level model classes use one field, `version` to represent state, reducing state space size.

Next, methods invoked on concurrent objects in JPF are delegated to the methods on the host JVM. For each concurrent class modeled in jpf-concurrent, there exists a subclass of `Model` (created by the authors) in the host JVM level, which implement the actual operations of the concurrent class. For each JPF-level model class, jpf-concurrent includes its corresponding native peer.

Each JVM-level model class contains a static map which maps a JPF-level model class to its corresponding JVM-level model class.

So, when a method is called on the JPF-level model class, MJI enforces the calling of the native peer method. The native peer then uses the static mapping to find the appropriate JVM-level model class and execute the actual method in the host JVM level. A `VersionManager` class is used to update the state of the JPF-level class based on the state of JVM-level class.

The methods of these classes operate on actual state of concurrent objects in the host JVM. As a result, JPF does not consider their execution with any transitions. This reduces number of thread interleavings explored by the model checker which results in a smaller state space. Fig 2 displays an example of the entire process.

The authors[8] also use a set of benchmarks, to show that in all cases jpf-concurrent provides a speedup for different types of classes included in Java Concurrency Utilities.

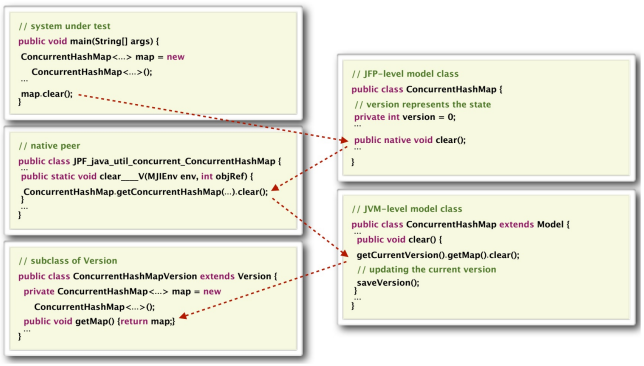

Figure 2: Transferring execution from JPF to the host JVM by jpf-concurrent

4 JPF-AWT

It is difficult to verify Graphical User Interface (GUI) applications for the following GUI-specific aspects:

- GUI applications are event-driven systems which engage in long-running interactions with users. So the number of possible interaction sequences can be very large.
- GUI applications are typically implemented on top of frameworks so, a large part of the behavior to be verified is handled by the framework instead of the program itself.
- GUI applications are inherently concurrent.

To address these problems, Mehlitz et al. [5], implement a JPF extension called JPF-AWT which can verify a GUI application efficiently using a script specifying user input sequences.

JPF-AWT replaces the low-level framework libraries while leaving the application unmodified. JPF-AWT also replaces classes like `java.awt.Graphics` with its own modeled version. Fig 3 gives a high-level architecture of JPF-AWT.

The main component of JPF-AWT is a replaced `java.awt.EventDispatchThread` which models non-deterministic user input. It does so by using JPF's provided NativePeer mechanism to interface to a scripting engine.

User interaction is modeled by allowing specification of input sequences by means of a simple scripting language which uses events as its building block. There are three special constructs in the scripting language: **ANY**, which represents non-deterministic input choices to be explored by model checker, **REPEAT** which represents loops, and **NONE**, which simply advance to the next event.

The scripts are processed by an interpreter. The *NativePeer* of JPF's modeled class `java.awt.EventDispatchThread` class executes at the host VM level, and is used to obtain the next event from the script interpreter.

The `processScriptAction()` method in this native peer class obtains the next input event from the interpreter and maps the event to the corresponding method in the target component class. The JPF VM then executes this method.

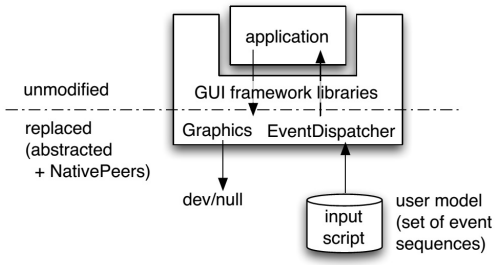

Figure 3: JPF-AWT architecture

Thus, JPF-AWT can execute complex user input sequences and leverage JPF's model checking capabilities to find defects caused by complex user interaction. The authors report that they successfully applied JPF-AWT to a large NASA ground data system and found defects which escaped conventional testing. They also state to enhance the scripting language as future work.

5 JPF-Android

Visser et al.[9] describe JPF-Android, a JPF extension that can be used to verify Android applications.

Since the Android framework is very large, one challenge of JPF-Android is to decide which parts of system to model. Modeling too much of the framework can result in the scheduling possibilities becoming exponential and causing a state space explosion.

So, JPF-Android focuses on verifying a single application with multiple components. To reduce scheduling possibilities, only the necessary components of the system service like the **ActivityManager, ActivityThread, Application components, Window structure, Message Queue and Intent** are implemented.

Like AWT applications, Android applications have a single-threaded design. So, JPF-Android's structure in based on JPF-AWT[5]. Just like JPF-AWT, user interactions with the UI is modeled as a set of scripted input sequences. JPF's **MJI** is used to model Android's message queue. A native `JPF_ActivityManager` class is used to model Android's Activity Manager. This class keeps a map of Intents provided by input script file and resolves an Intent to the relevant component to be scheduled to the application's message queue.

Android applications can contain multiple windows - one for each Activity. The application flow can switch between these activities at any point of execution. To model this, the authors introduced sections to the scripting language. Each section groups the input events of a specific activity.

However, transitioning from one section to another can lead to infinite event sequences. To handle this, the authors make JPF-Android keep record of its current activity and the current position in a visited activity's section. If an execution returns to a previously visited activity, its current position is looked up and execution continues from there.

The authors state that currently JPF-Android can detect deadlocks, race conditions and other property violations in applications. They state that their future work would be to model the extra Android libraries they left in their initial implementation and also to add a coverage testing extension to JPF-Android.

6 Verifying UML Statecharts with JPF

Mehlitz[4] published about an approach to verify UML statecharts using JPF. The three steps in his described approach were:

1. translate the UML model in a Java program
2. choose model properties to verify
3. optionally provide a guidance script

The translation from UML to Java program is generated from a list of rules. Some rules are: translating states into a Java class extending a library class(provided by the author), translating each trigger into a method, and handling transition by calling *setNextState* method from inside trigger methods.

The resulting program consists of two layers: the model generated from the diagram, and the UML library which connects the model with JPF.

Three types of properties can be verified: built-in JPF properties, generic properties implemented by the UML library, and application specific properties(checking these are implemented using JPF listeners).

However, models can be incomplete or too large. To handle such cases, the author provides the user the capability to specify guidance scripts. The scripting language is similar to the one used in JPF-AWT[5] and JPF-Android[9].

7 Verifying Actor-based Java Programs

In the *actor* programming model, multiple autonomous agents(actors) communicate by exchanging messages. Execution may differ based on interleaving of messages exchanged.

Marinov [3] et al. built a framework called **Basset** on top of JPF for testing actor systems compiled to Java bytecode. The main goal for Basset is to efficiently explore the actor application code itself and not exploration of the actor libraries.

7.1 Basset Architecture

Basset handles *actor states*, *actor execution* and *message management*.

For modeling actor states, Basset does not create the actors by itself but instead delegate the task to particular actor library used. It only maintains generic information about the actor.

Although each actor should run on its own thread, actor libraries use thread pools to reuse threads for new actors to reduce cost. However, Basset, in order to avoid thread interleaving, models each actor having its own execution thread. Since no threads are shared, actor operations can be executed atomically.

For message management, Basset maintains a *message cloud* of undelivered messages. In the main loop of Basset, in each iteration, a message in the message cloud is non-deterministically chosen to be delivered to its receiver actor.

7.2 Implementation

Basset is implemented on top of JPF.

The key extension made by the authors of Basset to JPF is the controlling of thread scheduling. Basset's own main controller is supposed to decide which actor thread should be executed, not JPF. So, the authors utilized JPF's MJI interface to implement their own thread scheduling.

The experimental results reported by the authors showed that Basset was more efficient in verifying actor program executions instead of directly exploring the code and its libraries.

8 Conclusion

In this survey, several extensions to JPF, used for different purposes(Regression Testing, Verifying concurrent, GUI, Android programs, Verifying UML statechart diagrams and Verifying and Actor programs) were discussed. Some of the common elements from each implementation are:

1. JPF Listeners are often used to add additional verification capabilities. Many of the JPF extensions implemented Listeners to add their own verification logic to perform their intended verification task.

2. A main difficulty in directly verifying these specialized programs using core JPF is the state space explosion problem. Most of the JPF extensions focus on trying to reduce state space being explored.

3. Many of these programs are built on top of frameworks (java.util.concurrent, java.awt, Android, Actor libraries). Core JPF ends up trying to verify the behavior of these frameworks as well, leading to the state space explosion.

4. So, many of the discussed JPF extensions make use of JPF's MJI interface to bypass framework verification and ensure that only the application itself is verified by JPF.

5. Another common pattern that can be observed is the use of JPF's MJI interface to avoid interleaving of threads. This also reduces state space being explored.

6. In case of a large number of possible program executions (GUI, Android, UML), researchers address the problem by allowing the user to provide specifications of particular executions they want to verify through scripts.

The above observations can prove to be useful for researchers working with JPF and/or trying to create new extensions to it.

References

[1] Java pathfinder. `http://babelfish.arc.nasa.gov/trac/jpf/wiki`. Accessed: 2015-06-07.

[2] java.util.concurrent. `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html`. Accessed: 2015-06-07.

[3] Steven Lauterburg, Mirco Dotta, Darko Marinov, and Gul Agha. A framework for state-space exploration of java-based actor programs. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 468–479. IEEE Computer Society, 2009.

[4] Peter Mehlitz. Trust your model-verifying aerospace system models with java pathfinder. *IEEE/Aero*, 2008.

[5] Peter Mehlitz, Oksana Tkachuk, and Mateusz Ujma. Jpf-awt: Model checking gui applications. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 584–587. IEEE Computer Society, 2011.

[6] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *Software Engineering, IEEE Transactions on*, 22(8):529–551, 1996.

[7] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(2):173–210, 1997.

[8] Mateusz Ujma and Nastaran Shafiei. jpf-concurrent: an extension of java pathfinder for java. util. concurrent. *arXiv preprint arXiv:1205.0042*, 2012.

[9] Heila van der Merwe, Brink van der Merwe, and Willem Visser. Verifying android applications using java pathfinder. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, 2012.

[10] Guowei Yang, Matthew B Dwyer, and Gregg Rothermel. Regression model checking. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 115–124. IEEE, 2009.