

Advanced Debugging with JPF Inspector

Pavel Jančík¹, Pavel Parízek^{1,2}, and Jan Kofroň¹

¹ Department of Distributed and Dependable Systems, Charles University

² School of Computer Science, University of Waterloo

Abstract. Debugging is mostly manual and very tedious work. It might take a long time to analyze the cause of a bug. Debugging of multi-threaded programs is especially difficult due to non-determinism in the thread scheduling, which is out of control of the developer. In this paper, we present JPF-Inspector—a tool for debugging Java programs, which is an extension of the Java PathFinder model checker. JPF-Inspector addresses some limitations of existing tools. In particular, it supports reverse stepping of the program execution, modification of a reversed program state, and re-execution from a modified program state. Furthermore, at each non-deterministic branching point of the state space, the user can choose the branch to be taken.

1 Introduction

Testing and debugging are important parts of software development process. Via testing, bugs can be found (code locations where the program fails in some sense), but it yields no information about the causes (code locations where the bug actually is). Even though automatization is heavily applied on testing, debugging still remains mostly manual, time-consuming work. Different techniques to ease the debugging process like “edit and continue”, “reverse-stepping”, and “record and replay” have been introduced. Reverse-stepping represents operations inverse to “step-over and step-in”. Reverse-stepping consists in undoing actions of the program and resuming the state of the program just before the actions were performed.

Debugging multi-threaded programs is a difficult and laborious work. Due to the non-determinism in thread scheduling one can hardly see the same program state in repeated program runs. The tools like Chess [2], GDB [4], DeJaVu [1], and Replay Debugging [8] in VMware tools (VMware Workstation) are able to repeat the same execution trace several times. The Chess tool records the thread scheduling during a run and later on, during debugging, schedules the threads in the same order. The GDB records executed instructions and during backward stepping internally reverts the recorded state changes to be able to show previous program states. However, the record/replay approach (used in the GDB tool) does not allow the program state to be modified during replay. Therefore, the developer cannot easily examine if the error being debugged occurs in the case of, e.g., different parameter values as well. The tools mentioned above have several limitations, such as a limited record length in GDB, only single-CPU replays

in VMware’s Replay Debugging, and imperfect replays (due to time dependent functions, and lazy program state initialization) in Chess.

In these tools, the developer cannot specify/modify the used thread interleaving during debugging so that he/she cannot compare similar (error and error-free) traces. Only the Chess and Jinx [5] tools are able to influence the scheduling and (automatically) explore different ones. Other tools like GDB and Replay Debugging only record a single (random) scheduling.

Backward/Reverse stepping is a feature demanded by developers. It is supported in GDB (since version 7.0) and in VMware Replay debugging “extension”. Chess, in contrast, allows only re-execution of a given trace. The current GDB implementation does not handle multi-threaded programs well; actions (like writes into shared variables) of other threads are not propagated into the reversed program state while reverse stepping.

Assertions are frequently used means to check low-level properties at runtime. In the Java world, assertion checking can be enabled and disabled only at the application startup; dynamic enabling can be, however, beneficial due to the overhead and modification of thread scheduling probabilities they can introduce.

1.1 Contribution

We present the JPF-Inspector tool that addresses the aforementioned limitations of the existing tools. It allows easier debugging and bug-inspection of Java programs. JPF-Inspector offers easier exploration of similar program states by modification of a reversed program state and enabling the developer to drive thread interleaving.

The rest of the paper is organized as follows. Java PathFinder (a tool we have extended) is briefly introduced in Section 2. In Section 3, JPF-Inspector and its features are described. An example of using JPF-Inspector and the way its features help during debugging are shown in Section 4. In Section 5, our future plans are described.

2 Java PathFinder

Java PathFinder (JPF) [3] is a code model checker. In fact it is a highly configurable and extensible JVM optimized for program state inspection and (systematic) exploration of non-determinism in programs. JPF can be also used to collect run-time program characteristics like coverage metrics and generate “interesting” parameter values for tests.

JPF is an explicit state model checker. It means JPF stores the whole program states with concrete values of all variables. However, JPF does not provide means for easy exploration of stored states to users. If an error is reported, an error trace (list of executed instructions) does not contain modified values (and/or dumps of program state), so that it is extremely hard for the user to understand what went wrong.

JPF incorporates various optimizations such as program state matching (stateful search), on-the-fly partial order reduction, and class-loading symmetry and heap-symmetry detection to reduce the number of program states to be visited. The program is executed in transitions—a sequence of instructions executed in one thread with at most one change visible to other threads. JPF also records the changes made by each transition to be able to restore the previous states. The way JPF traverses the state space is driven by the “search strategy” such as depth-first search; various heuristics can be incorporated.

Not every action can be easily undone—output to the console, sending network packets, native methods in Java, etc. are in principle not undo-able. In such cases, models to handle these situations have to be created. JPF contains models for the most common native methods from the Standard Java Libraries, like file operations and networking. However, if a program calls native methods (or uses third-party libraries which call them), one will probably have to write own model classes to be able to apply JPF his/her program.

JPF has a built-in race detector and is able to explore all thread interleavings; it is thus eligible for detection of concurrency bugs. JPF also checks assertion violations, deadlocks, uncaught exceptions, and null dereferences. The checked properties can be easily extended by the user created “listeners”.

As stated before, JPF is optimized for extensibility, state exploration and not for the fastest execution of a single program trace (in contrast to standard JVMs). JPF is (order of/roughly) 200 times slower than common JVMs, which is still acceptable slowdown when applied on unit tests and other small parts of programs. The slowdown introduced by “record and replay” implementations (in GDB) is similar to JPF; in both approaches executed instructions and their side effects are logged.

3 JPF-Inspector

JPF-Inspector is an JPF extension which enables debugging of the system under test (SuT) executed in JPF. It runs on top of JPF and instruct JPF how to traverse the state space, gather and modify representation of SuT state (variables) and thus introduce “user friendly” interface to controlling JPF.

JPF-Inspector supports various breakpoints, basic debugging commands like single stepping of program execution, and inspection and modification of a program state.

While debugging, the key point is to specify when to break the execution and start the inspection of the current program state. JPF-Inspector offers various kinds of breakpoints—at a given program location, at a field access, and at the states when an object of a given type is created or released (garbage collected). JPF-Inspector also supports specific breakpoints to ease the handling of non-determinism in SuT. The execution can be stopped if a given thread is scheduled or when a value (of a variable) has to be non-deterministically selected.

JPF-Inspector supports forward and backward single stepping of the program execution at various granularities. A backward step completely restores

the program state (even in the case of a multithreaded program). This involves restoration of the heap, program stacks, and the states of all the threads with their corresponding instruction pointers. It also allows to modify the restored program state and continuing the re-execution from the modified state to observe if misbehavior manifests itself for other situations (e.g., different data).

Due to the nature of JPF, any possible interleaving of the threads can be observed. The developer can control the non-determinism hidden in the program; this means to specify a thread to schedule or a value to be used (e.g., the return value of the *Random.nextInt()* method). This is especially helpful when debugging a data race to see and compare to the “standard” execution when the race does not occur. Execution on a multi-core CPU can be simulated by a proper thread interleaving. This is achieved by scheduling particular threads in an alternating way (another thread is scheduled after a single transition).

JPF-Inspector allows introspection of a program state. Threads, call-stacks, values of parameters, local variables, and object fields can be printed-out. It is also possible to change the value of each variable of a primitive type.

By using JPF-Inspector it is possible to enable and disable assertion checking on demand at runtime. Furthermore, new assertions can be also dynamically added. The newly added assertions can just observe the current state of the program and it is guaranteed that the program state is not modified by them.

A complete description of all supported commands is provided in the user guide [7].

4 Debugging with JPF Inspector

This section illustrates the way JPF-Inspector features help while searching for bug causes. It should sketch the debugging process while using JPF-Inspector, possible conjectures of the developer and in particular reasons why given command has been used.

Consider the following situation. One of the unit tests in a project non-deterministically fails—probably contains a race. It is time to use race detection tools such as JPF to identify the bug source. Figure 1 shows a code snippet, which is used throughout this section. The code fragment shows a textual file cache with asynchronous file loading (prefetching). It contains *CacheManager* which holds the content of cached files, *FileLoader* which loads files in background and *TestClass* which represents a failing test.

The developer runs JPF in order to find a thread schedule which results in an assertion violation. JPF really finds the error, however the provided error trace is not of much help. To be able to analyze the error trace the test is re-executed in JPF-Inspector. Figure 2 contains an extract of the JPF-Inspector command line (user commands and JPF-Inspector responses) described in this section. First, the developer has to instruct JPF-Inspector to stop after JPF discovers the error (assertion violation, null dereference, uncaught exception, etc.) by the `create breakpoint property_violated` command and then starts JPF by `run`.

```

1  class CacheManager {
2      private FileLoader loader;
3      private Map<File, String> cache;
4
5      // Asynchronously load file to cache.
6      synchronized void loadFile(
7          String fileName, String charset) {
8
9          if (!fileInCache(fileName)) {
10             loader.setFileName(fileName);
11             loader.setCharset(charset);
12         }
13     }
14
15     boolean fileInCache(String);
16     String getCachedFile(String);
17 }
18
19 public class TestClass {
20     public static void main (...) {
21         CacheManager cm;
22         cm = new CacheManager();
23         cm.loadFile("Test1.ini", "UTF-8");
24         String content =
25             cm.getCachedFile("Test1.ini");
26         assert (content.equals(
27             "\#TestData-UTF8-áéíóüÿ-end"));
28     }
29 }
30 class FileLoader extends Thread {
31     private File file = null;
32     private String charset = "US-ASCII";
33
34     FileLoader () {
35         ...
36         this.start(); // Start loading thread
37     }
38
39     synchronized void setFileName
40         (String fileName) {
41         file = new File(fileName);
42         notify(); // Wake up loader thread
43     }
44     synchronized void setCharset (String);
45
46     synchronized void run () {
47         this.notify(); // Wake up constructor
48         while (true) {
49             wait(); // Wait for file to process
50             // Load file to buffer
51             // Convert charset
52             Charset cs = Charset.forName(charset);
53             ...
54             // Store to CacheManager
55         }
56     }
57 }

```

Fig. 1. Code extract of the failing test

4.1 Use Case 1: Dynamic Assertions

After a while JPF-Inspector finds the error and breaks its execution. Now it is possible to inspect the reached program state as well as the yielded error trace. The `print` command can be used to print-out the loaded cached content of the file. The malformed non-ascii characters ("□" characters in Figure 2 at line 11) in the `content` variable point to the charset conversion in the `FileLoader` class. Either the charset field holds an incorrect value or there is a bug in the conversion routine. A new assertion is added (`assert pos=TestClass.java:52 this.charset == "UTF-8"`) at runtime to check the first hypothesis. Then, backward steps (`back_step_over 3`) are used to restore the program state before the `loadFile` call (Figure 1 at line 23) and the execution is started in the forward direction by the `run` command to check the newly added assertion. Now, JPF follows the backtracked trace and checks the assertion. The `charset == "UTF-8"` assertion violation is found and the execution is stopped at line 52.

The `print this.charset` shows that the charset field holds its initial value (`US-ASCII`) while `UTF-8` is expected. The expected value can be stored in a variable (`set this.charset "UTF-8"`) and the program is re-started (`run`) to check whether the wrong charset is the only cause making the test fail. Note that in the record/replay tools like GDB it is not possible to set the variable/field after a backward step as in JPF-Inspector. After the correction of the charset field, no assertion is violated and the test is passed.

```

1 The Inspector console: TestClass
2
3 cmd>create breakpoint property_violated
4 New breakpoint succesfully created with ID=1
5 cmd>run
6 ...
7 cmd>print
8 TestClass.main(String[]) - TestClass.java:26 - assert (content.equals(
9   0 : args (java.lang.String[]) = []
10  1 : cm (CacheManager) = CacheManager@140
11  2 : content (java.lang.String) = "#TestData-UTF8-□□□□□□-end"
12
13 cmd>assert pos=TestClass.java:52 this.charset == "UTF-8"
14 New assertion successfully created with ID=2
15
16 cmd>back_step_over 3
17 INFO: SuT is stopped
18   SuT (Thread=0) executes the TestClass.java:23 - aload_1 source: cm.loadFile("Testl..."
19 ...
20
21 INFO: Assertion (ID=2) violated
22   SuT enters the TestClass.java:52
23
24 cmd>print this.charset
25 charset (java.lang.String) = "US-ASCII"
26   0 : value (char[]) = [U, S, -, A, S, C, I, I]
27 ...
28 cmd>set this.charset "UTF-8"
29 Set charset (java.lang.String) = "UTF-8"
30 cmd>run
31 # Execution terminated and no property violation is found now
32 ...
33 # Reexecute SuT to reach state where the dynamic assertion is violated again
34 cmd>run
35 INFO: Assertion (ID=2) violated
36   SuT enters the TestClass.java:52
37 cmd>enable print scheduling choice_generators
38 cmd>back_step_transition
39 ChoiceGeneratorAdvance - scheduling CG - monitorEnter (1c904f75) :
40   0-ThreadInfo [name=main,index=0,state=RUNNING]
41   >1-ThreadInfo [name=FileLoader,index=1,state=UNBLOCKED]
42 Execution is halted. Specify which choice to use (0-1)
43 Hint: Use 'cg select CHOICE_INDEX' command
44 # The ">" marks scheduled thread
45
46 INFO: SuT is stopped
47   SuT (Thread=1) executes the java/lang/Object.java:-1 - executenative Object.waitV
48
49 cmd>thread_pc
50 0 : TestClass.java:11: loader.setCharset(charset);
51  CacheManager:loadFile:11:invokevirtual loader.setCharset(Ljava/lang/String;)V
52 1 : java/lang/Object.java:-1:(java/lang/Object.java:-1)
53   java.lang.Object:wait:0:executenative JPF_java_lang_Object.waitV
54
55 cmd>cg select 0
56 ChoiceGeneratorAdvance used values - scheduling CG - monitorEnter (1c904f75) : >0-Thre...
57 ...

```

Fig. 2. JPF-Inspector console session extract

4.2 Use Case 2: Backtracking and Choices

As the previous section showed, the *charset* field has not been set. Now the error trace will be examined to find why it happened. First, the place where the

dynamic assertion `charset == "UTF-8"` is violated is reached again by the `run` command. Then the `enable print scheduling choice_generators` command is invoked to see the threads which could be scheduled at each program state (runnable threads). The `back_step_transition` command can be used to visit a previous state where different thread can be scheduled or other values assigned. The whole program trace can be inspected by a repeated invocation of the back-step commands.

In our show case, only a single backward transition step is enough in order to reach the broken `loadFile` method where the bug takes place. As the `thread_pc` command shows (see lines 50-53 in Figure 2), after the `back_step_transition` command, the `main` program thread is stopped just before the call of the `setCharset` method (Figure 1 line 11) and the `FileLoader` thread is waiting in the `run` method. The JPF-Inspector session log (line 41 in Figure 2, the `>` character marks the scheduled thread) shows that in the current error trace, `FileLoader` thread has been executed after this state, so the `setCharset` method has not been called and (as a result) the charset conversion has failed.

Hence, the problem has been identified. Even in this trivial example, JPF-Inspector can be easily used to check if there is another condition making the bug manifest. The `main` thread can be scheduled (for the execution) by the `cg select 0` command so that the `setCharset` method will be called. Then the `run` command is used to start forward execution. Note that the `cg select` command destroys the backtracked trace if another thread is scheduled, so that the `run` command will check all possible thread scheduling if any assertion is violated. In this case, the test is passed successfully, so that it is enough to forbid the `FileLoader` thread to be scheduled after the `setFileName` method. The race can be solved by putting both setters (`setFileName` and `setCharset`) into a single critical section (wrap them by `"synchronized(fileLoader)"`). Note that the race (multi-variable synchronization problem) has not been detected by the JPF race detector which detects unsynchronized parallel accesses to fields, because each access to the field is properly synchronized.

5 Future work

Our top priority is to finish implementation of the commands and features that we already started working on. This includes dynamic assertions introduced in Section 4, and improvements of the existing commands. We will also implement additional functionality and new variants of the existing commands. Let us mention only those commands related to program debugging—backward step to a previous breakpoint hit (as supported in GDB), breakpoints upon certain expressions over variable values, and backward steps to a transition boundary that is connected with a specific event (e.g., access to a shared object).

Currently developers must write a lot of commands by hand when inspecting the bug. To enhance the usefulness of JPF-Inspector, we plan to partially automate the search for root causes of bugs detected by JPF (under reasonable assumptions).

The error trace and the program state contain a lot of interesting information about the error. The propagation phase of the error as well as the conditions for the error to manifest itself are both recorded inside the error trace. JPF-Inspector will serve as a framework to gather the detailed information from the error trace easily. The error analysis will utilize current JPF-Inspector commands as basic steps of automatic analysis algorithms.

Thanks to the fact that JPF model checker is in background of JPF-Inspector, it is possible to check different thread interleaving if the bug manifests itself in different program traces. In combination with slicing on the passing and failing thread schedules we should be able to help finding the statements contributing to the given bug [9]. Having JPF as a basis, which finds real errors only, our approach promises low false-positives rates (in contrast to static analysis).

Our long term vision is to make JPF-Inspector mature to day-to-day debugging and moving JPF-Inspector functionality into mainstream IDEs. A tight integration to IDEs includes implementation of the Java Platform Debugger Architecture [6] interface and extension of IDEs' built-in debuggers.

6 Conclusion

In this paper we presented the JPF-Inspector extension. This extension is intended to ease understanding of the error trace generated by JPF. As it has been shown in Section 4, the backward steps are useful also for finding root causes of bugs. Moreover we consider the backward stepping with state modification useful in the ordinary debugging when developer made an unwanted forward step.

Acknowledgements. This work was supported by the Google Summer of Code program and by the Grant Agency of the Czech Republic project P202/11/0312.

References

1. J. Choi and H. Srinivasan. Deterministic Replay of Java Multithreaded Applications, SPDT, 1998.
2. M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs, OSDI, 2008.
3. W. Visser, K. Havelund, G. Brat, S. Park and F. Lerda. Model Checking Programs, Automated Software Engineering Journal, Vol. 10, 2003.
4. GDB: The GNU Project Debugger, <http://www.gnu.org/software/gdb/>
5. Jinx, <http://www.corensic.com/>
6. Java Platform Debugger Architecture, <http://java.sun.com/javase/technologies/core/toolsapis/jpda/>
7. JPF-Inspector documentation, <http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-inspector/userguide/>
8. VMware - Relay Debugging, <http://www.replaydebugging.com>
9. D. Weeratunge, X. Zhang, W.N. Sumner, and S. Jagannathan. Analyzing Concurrency Bugs Using Dual Slicing, In ISSA 2010, ACM.