Advanced Debugging with JPF-Inspector

Pavel Jančík
Charles University, Czech Republic

Jan Kofroň
Charles University, Czech Republic

Pavel Parízek
University of Waterloo, Canada

***Abstract*—Debugging in the context of JPF relates both to analyzed Java programs and the JPF itself. In the first case, the main challenge is debugging concurrent programs. In the case of JPF itself, the main challenge is to find what is happening inside. We present the JPF-Inspector — a tool for monitoring and control of JPF during traversal of Java program's state space. It makes debugging easier by addressing some limitations of existing tools. We describe main features of JPF-Inspector and show on a small example how it can be used for debugging of concurrent programs. Finally, we discuss our vision regarding JPF-Inspector and future plans.**

***Keywords*-debugging, concurrency, state space traversal, backtracking, Java Pathfinder**

I. Introduction

Debugging programs is very hard. Tools like JPF provide a trace for each detected error, but then it is still hard to find the root cause (incorrect statements) of the error from the trace and fix the program code. In the context of JPF, we must specifically consider (1) debugging the Java program in which JPF found some error, and (2) also debugging the JPF itself. The latter is very important as there are many custom extensions and optimizations for JPF.

The main challenge in debugging JPF and its extensions on some Java program is to find what is happening inside JPF at any given time during traversal of the program's state space, and whether JPF is processing the given Java program correctly. The relevant information for this purpose include: what bytecode instructions were executed (in a given transition), what non-deterministic choices were generated, whether state matching works correctly, and what is the current program state. In some cases, it might be useful to know also the current state of internal data structures of JPF.

Current approaches to debugging Java programs analyzed by JPF include manual inspection of the program code and the error trace provided by JPF, reading huge log files, adding code for printing debug messages, using debuggers (e.g., GDB or the debugger available in a given IDE), and writing special purpose JPF listeners for tracking relevant operations and exploring program state. For debugging the JPF implementation, which involves fixing the bug indicated by a failing unit test, the prevailing approaches include manual inspection of the code, printing debug messages, and reading log files. All these tasks are manual to a large degree and very time consuming.

We present the JPF-Inspector — a tool for monitoring and control of JPF during traversal of the program state space. It supports common features of debuggers like breakpoints and single-step execution, and it also provides means for inspecting and changing the state of the analyzed Java program at any point during the state space traversal. Developers can use Inspector to interactively find the necessary information about program state and JPF execution, instead of adding debug messages into the JPF codebase and manually reading huge log files. In general, the JPF-Inspector makes debugging of Java programs in the context of verification with JPF much easier, and it also greatly simplifies the process of developing custom JPF extensions, properties, and optimizations — especially when considering more complex Java programs as test inputs.

The rest of the paper is organized as follows. We describe all features of JPF-Inspector in Section II. Then we illustrate how to use Inspector for easier debugging of concurrent Java programs and general inspection of their behavior on a small example (Section III), and we also discuss how Inspector could be used for debugging JPF and its extensions (Section IV). Finally, we provide a list of planned features and our long-term vision regarding Inspector in Section V.

II. JPF-Inspector

The JPF-Inspector is a standard JPF extension that allows the user to interactively control the process of state space traversal and explore program state. More specifically, Inspector currently supports the following: breakpoints at specific events, program state inspection and limited modification, single step traversal of program state space both in forward and backtrack direction, selection of a particular option at a non-deterministic choice point, and replay of a previously recorded command sequence. In our opinion, this is the most important set of features useful for debugging. The tool is available from the Mercurial repository through URL that is mentioned at the web page http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-inspector. Now we describe the main features of JPF-Inspector in more detail.

Breakpoints can be defined at these events: property violation, execution of a specific line of code, execution of a specific instruction, transition boundary, access to some field of some heap object, invocation of a specific method, and scheduling of a particular thread. However, the set of events is not fixed — it can be extended by the user.

Single step execution is currently supported at these levels of granularity (in both directions): instruction, transition, and source code line (with and without stopping at nested and enclosing method invocations). Moreover, it is possible to run Inspector in a mode where it prints all options (enabled transitions) at each non-deterministic choice and the user must select one of them. When the user performs some backtrack

steps from the state s, the tool remembers taken choices as defaults such that it is then possible to go forward into the original state s (in which the manual inspection started).

Program state inspection is supported through commands that print: status and call stack of each thread, field values of each heap object, and values of local variables in any stack frame of any thread. Navigation over fields of the reference type (i.e., over the tree of heap objects) is possible through a custom expression language. Modification of the program state is currently restricted to changing values of local variables and object fields. For example, it is not possible to add new frames to call stack of some thread, to remove existing frames, or to create new heap objects.

Inspector allows the user to save into a text file all commands executed in the current session, and replay them later in another session. It is not possible to save only a particular subsequence of commands through any Inspector command.

A complete description of all supported commands — their syntax and output — is provided in the user guide [8].

The user interface has the form of a JPF Shell panel that contains an embedded simple textual console for writing commands and printing output.

From the implementation perspective, Inspector consists of (1) a server module that provides all the functionality and (2) a client module that manages the user interface and reads user commands. The server module provides an API for calling Inspector from other program analysis frameworks.

III. Debugging Concurrent Programs

The main goal of the debugging process is to find the root cause of an error detected during program execution or by some program verification tool, and fix the bug afterwards. Many useful tools that facilitate debugging have been developed in the past — for example, GDB [6], JPDA [7], Replay Debugging in VMware tool [9], and CHESS [3]. Nevertheless, each of these tools has certain limitations and debugging is still very difficult and tedious work. GDB does not support program state modification in combination with backward steps and does not allow executing the program under a different thread schedule, so the developer cannot easily check whether the error depends on the values of some variables or on the thread schedule. The CHESS tool does not support replay of an error trace in combination with program state modification.

In general, a big challenge is to support efficient debugging of concurrent programs. Tools like CHESS allow reproducing concurrency errors despite non-determinism in thread scheduling, but they do not point to root causes of detected errors.

JPF-Inspector addresses some limitations of existing debuggers and provides mechanisms for more efficient debugging of concurrent programs in the context of automated verification with JPF. Specifically, the developer can set breakpoints, explore program state, modify the program state, do backward steps, and control thread scheduling at the same time. A typical scenario not possible by existing tools is the following: create some breakpoints, run the program until it stops at one of the breakpoints, inspect program state, make several backward steps, optionally modify the program state, and execute the program in the forward direction in a stepwise manner (1) along the same path or using a different schedule (a different execution path) and (2) possibly with different values of program variables (input data). This way, the developer can use Inspector to find what conditions trigger a given error.

Single step execution allows the user to focus on a specific part of the program state space after a breakpoint was hit, and to inspect the effects of the next few instructions (transitions) on the program state and error occurrence. The ability to control thread scheduling interactively (and explore a particular thread interleaving) is especially helpful when debugging a concurrency error, since the developer can check whether the given error occurs under a different schedule and thus narrow down the set of possible root causes (buggy program code locations). Of course, it is also possible to use Inspector just for interactive observation of the program behavior under specific conditions, when no specific errors are known.

A very big advantage is that JPF behaves as a real Java virtual machine, and therefore it precisely simulates all bytecode instructions in a given program. The only exception are calls of native methods.

In the rest of this section, we show the usage of JPF-Inspector for debugging on a small multi-threaded program.

Figure 1 shows a program that contains an atomicity error and a unit test that fails non-deterministically at the assertion (line 25) because of the error. The program represents a cache for text files with asynchronous loading. An instance of the CacheManager class holds the content of cached files and an instance of the FileLoader class is responsible for loading of files using a background thread. The Test class represents the failing unit test.

When the developer runs JPF on the program, it finds the error (violated assertion). The error trace provided by JPF indicates the thread schedule that results in the assertion violation but in general does not help much in determining the root cause of the error.

One option is to use JPF-Inspector to analyze the error trace and find the root cause. Figure 2 shows the respective user session in JPF-Inspector — a sequence of user commands and their output. At first, the developer has to re-execute the failing test in JPF-Inspector such that the execution stops immediately after JPF detects the error. This can be achieved by the commands `create breakpoint property_violated` and `run`.

When JPF finds the error and JPF-Inspector stops its execution at the breakpoint, a user can inspect the current program state as well as the error trace. The `print` command prints the values of all local variables in the top stack frame of the current thread — this includes the loaded content of the input file (line 11 in Figure 2). The malformed non-ASCII characters ("□") in the content variable point to some problem with the charset conversion in the FileLoader class. Either the charset field contains an incorrect value or there is a bug in the conversion routine. A new assertion can be dynamically added

```
1  class CacheManager {
2    private FileLoader loader = new FileLoader();
3    private Map<File, String> cache;
4
5    // asynchronously load file to cache
6    void loadFile(String fileName, String charset) {
7      synchronized (this) {
8        if (!fileInCache(fileName)) {
9          loader.setFileName(fileName);
10         loader.setCharset(charset);
11       }
12     }
13   }
14
15   boolean fileInCache(String) { ... }
16
17   String getCachedFile(String) { ... }
18 }
19
20 public class Test {
21   public static void main (...) {
22     CacheManager cm = new CacheManager();
23     cm.loadFile("Test1.ini", "UTF−8");
24     String content = cm.getCachedFile("Test1.ini");
25     assert content.equals("#áéíóúý");
26   }
27 }
```

```
28 class FileLoader extends Thread {
29   private File file = null;
30   private String charset = "US−ASCII";
31
32   FileLoader () {
33     this.start(); // start loading thread
34   }
35
36   void setFileName(String fileName) {
37     synchronized (this) {
38       file = new File(fileName);
39       notify(); // wake up loader thread
40     }
41   }
42
43   synchronized void setCharset (String) { ... }
44
45   synchronized void run () {
46     while (true) {
47       wait(); // wait for file to process
48       // load file to buffer
49       // convert charset
50       Charset cs = Charset.forName(charset);
51       // store to CacheManager
52     }
53   }
54 }
```

Fig. 1. Program with an atomicity error

(using the command `assert pos=TestClass.java:51 this.charset == "UTF-8"`) to check the first hypothesis. Note that we did not have to recompile the program after the new assertion was added. Everything is fully dynamic and controlled from the JPF-Inspector session — new assertions are checked in a JPF listener. Then, a sequence of three backward steps (the command `back_step_over 3`) is done to restore the program state as it was before the call of the loadFile method (at line 23 in Figure 1), and program execution is restarted in the forward direction by the `run` command to check whether the newly added assertion holds. JPF finds that the assertion is violated, i.e. we know that `charset != "UTF-8"`, and the program execution is stopped at line 51.

The `print this.charset` command shows that the charset field contains its initial value (US-ASCII), although the expected value is UTF-8. The program state can be modified so that the field has the expected value (using the command `set this.charset "UTF-8"`), and the program is restarted (by the `run` command) to check whether the wrong charset is the only reason why the test failed. None of the assertions is now violated and the test successfully finishes.

By the procedure described above, we found that the charset field has not been set correctly. The next step is to inspect the error trace together with program code to find the root cause of the error (the reason why this error occurred). First, we must use the `run` command to execute the program again so that is reaches the point where the dynamically added assertion (`charset == "UTF-8"`) is violated.

Then the `enable print scheduling cg` command puts Inspector into a mode where it prints all runnable threads that can be scheduled at each transition boundary. The `back_step_transition` command is used to move the program into a previous state where a different thread can be scheduled or some other values can be assigned to any variable. In our case, a single backward step over a transition is enough to reach the loadFile method that contains the bug. Output of the `thread_pc` command (at lines 48-51 in Figure 2) shows that the main program thread is stopped just before the call of the setCharset method (line 10 in the program code) and that the file-loading thread is waiting in the run method (line 47). The JPF-Inspector session log shows at line 40 that the file-loading thread runs in the transition that we backtracked over (the ”>” character marks the thread to be scheduled), and thus we know that the setCharset method has not been called and, as a result, the charset conversion has failed. The problem — a bug in the program — has been successfully identified.

Inspector can be also used to check if another condition triggers the error. The main thread can be scheduled using the `cg select 0` command, so that the setCharset method will be called in the next transition outgoing from the current state s. Afterwards, the `run` command must be used to start JPF again. It then explores all thread interleavings from the state s, and in particular checks whether the assertion is violated in any thread interleaving. Now the test passes successfully (if the main thread was scheduled in the transition outgoing from the state s), and the user can see that the error does not occur if the file-loading thread is not scheduled right after the call of the

```
1  The Inspector console: Test
2
3  cmd>create breakpoint property_violated
4  New breakpoint succesfully created with ID=1
5  cmd>run
6  ...
7  cmd>print
8  Test.main(String[]) − Test.java:25 − assert content.equals("#áéíóúý");
9    0 : args (java.lang.String[]) =[Ljava.lang.String;@a7
10   1 : cm (CacheManager) =CacheManager@140
11   2 : content (java.lang.String) =java.lang.String@1000447 − #□□□□□□
12
13 cmd>assert pos=Test.java:51 this.charset == "UTF-8"
14 New assertion succesfully created with ID=2
15
16 cmd>back_step_over 3
17 INFO: SuT is stopped
18   SuT (Thread=0) executes the Test.java:23 − aload_1 source: cm.loadFile("Test1.ini", "UTF-8");
19
20 INFO: Assertion (ID=2) violated
21   SuT enters the Test.java:51
22
23 cmd>print this.charset
24 charset (java.lang.String) =java.lang.String@156 − US-ASCII
25   0 : value (char[]) =[C@157
26 ...
27 cmd>set this.charset "UTF-8"
28 Set charset (java.lang.String) =java.lang.String@159 − UTF-8
29 cmd>run
30 # Execution terminated and no property violation is found now
31 ...
32 # Reexecute SuT to reach state where the dynamic assertion is violated again
33 cmd>run
34 INFO: Assertion (ID=2) violated
35   SuT enters the Test.java:51
36 cmd>enable print scheduling cg
37 cmd>back_step_transition
38 ChoiceGeneratorAdvance − scheduling CG − monitorEnter (1c904f75) :
39      0−ThreadInfo [name=main,index=0,state=RUNNING]
40    >1−ThreadInfo [name=FileLoader,index=1,state=UNBLOCKED]
41 Execution is halted. Specify which choice to use (0−1)
42   Hint: Use 'cg select CHOICE_INDEX' command
43
44 INFO: SuT is stopped
45   SuT (Thread=1) executes the java/lang/Object.java:−1 − executenative Object.wait_V
46
47 cmd>thread_pc
48 0 : Test.java:10: loader.setCharset(charset);
49   CacheManager:loadFile:10:invokevirtual loader.setCharset(Ljava/lang/String;)V
50 1 : java/lang/Object.java:−1:(java/lang/Object.java:−1)
51   java.lang.Object:wait:0:executenative JPF_java_lang_Object.wait_V
52
53 cmd>cg select 0
54 ChoiceGeneratorAdvance used values − scheduling CG − monitorEnter (1c904f75) : >0−Thread ...
55 ...
```

Fig. 2. User session in JPF-Inspector

setFileName method. This is an example of how comparison of failing and passing execution (thread interleaving) can help in identifying the error root cause, when the error depends on a specific interleaving. A possible fix of the atomicity error is to create a new synchronized block around the calls of methods setFileName and setCharset, where the fileLoader variable is used as the monitor.

This example debugging scenario illustrates how features of JPF-Inspector are useful for analysis of an error trace and finding the actual bug in the code. The most important aspects are: (1) the possibility to combine backward steps together with program state modification and explicit control of thread

scheduling, and (2) the ability to run JPF with breakpoints at specific code locations or transition boundaries.

IV. Debugging JPF Extensions

Existing features of JPF-Inspector can be used also for debugging JPF and its extensions. Supported commands allow the developer to see, for example, what choices are generated at each transition boundary and how state matching works (i.e., whether some program state was identified by JPF as already visited). The developer can find how a given optimization changes (1) the shape of the program state space, (2) individual transitions (e.g., their length), and (3) the content of individual program states, and whether it works correctly.

Nevertheless, many other features and new commands are needed for really efficient debugging of JPF, like support for introspection. Their implementation is a part of our future work, as described in the next section.

V. Future Work

The current version of JPF Inspector is the first step on the way to allow (1) easier debugging of Java programs that are analyzed by JPF and (2) easier debugging of JPF extensions. Much work still has to be done to fully achieve that goal.

Our top priority is to finish implementation of commands and features that we already started working on. This includes dynamic assertions, which we already introduced in Section III, and improvements of existing commands. We plan to support dynamic adding of new assertions over the program state and code locations, and also dynamic enabling/disabling of existing assertions. Regarding existing commands, we will implement additional functionality and new variants — backward step to a previous breakpoint hit, single step execution with repetition count, breakpoints upon certain expressions over variable values, backward steps to a transition boundary that is connected with a specific event (e.g., access to a shared object), and many others.

In the long term, we plan to do the following:

- implement additional commands, especially those necessary for debugging of JPF extensions,
- automate many tasks that JPF-Inspector can be already used for in an interactive way,
- improve the user interface by adding new controls and views, and by better integration with existing IDEs,
- provide basic support for the Symbolic JPF [1], and
- compare with other debugging tools and techniques on real-world concurrent programs.

Our general goal is to improve the JPF-Inspector so that it is a mature and useful debugging tool.

The principal feature necessary for debugging JPF extensions is some form of JPF introspection. This means displaying the content of internal data structures, and the raw state of choice generators, listeners, and properties.

Some automation is certainly needed, because currently the users must do everything by hand through the interactive console. The replay feature provides only limited automation. We would especially like to partially automate the search for root causes of errors detected by JPF, and maybe also to support code completion for writing Inspector commands. To enhance the usefulness of JPF-Inspector in finding root causes of detected errors, we could implement algorithms that were proposed recently and maybe improve them in the context of JPF (using precise knowledge of program state). Delta debugging can be used to find a thread schedule that triggers a given error [2]. Another option is to use trace comparison and slicing on the passing and failing thread schedules to help finding the statements forming the given bug [4]. We could also use some ideas about finding suspicious thread interleavings from [5] and modify the Inspector GUI so that it recommends thread choices that form the suspicious interleavings. The overall goal is to automatize the identification of root causes for concurrency errors as much as possible under reasonable assumptions (constraints). Lot of information already available in JPF (e.g., program state and current trace) can be exploited for this purpose.

The new GUI elements may include these: program state explorer (tree-like), breakpoint manager, and buttons for frequently used commands. Besides improving the GUI, we will also implement a purely command-line (textual) user interface that could be used from various scripts.

Support for the Symbolic JPF would include these features: (1) commands for printing attributes of various program state elements, and (2) the possibility to define symbolic values for variables on-the-fly or change symbolic values back to concrete values.

JPF-Inspector could be used also to find why state explosion occurs for a given Java program. A feature required for this usage is state comparison (”diff”). Inspector would be able to compute differences between any two states on any paths — e.g., the names of variables that have different values in the given states. With this information, the user would be able to determine what causes the increase in the number of reachable states (e.g., a counter with ever increasing value).

Acknowledgements.

This work was supported by the Google Summer of Code program in the years 2010 and 2011.

References

[1] S. Anand, C.S. Pasareanu, and W. Visser. JPF-SE: A Symbolic Execution Extension to Java PathFinder, In TACAS 2007, LNCS, vol. 4424.
[2] J.-D. Choi and A. Zeller. Isolating Failure-Inducing Thread Schedules, In ISSTA 2002, ACM.
[3] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P.A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs, In OSDI 2008, USENIX.
[4] D. Weeratunge, X. Zhang, W.N. Sumner, and S. Jagannathan. Analyzing Concurrency Bugs Using Dual Slicing, In ISSTA 2010, ACM.
[5] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps. ConSeq: Detecting Concurrency Bugs through Sequential Errors, In ASPLOS 2011, ACM.
[6] GDB: The GNU Debugger, http://www.gnu.org/software/gdb/
[7] Java Platform Debugger Architecture, http://java.sun.com/javase/technologies/core/toolsapis/jpda/
[8] JPF Inspector user guide, 2011, http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-inspector/userguide/
[9] Replay debugging in VMware products, http://www.replaydebugging.com