JPR: Replaying JPF Traces Using Standard JVM

Kaiyuan Wang, Sarfraz Khurshid, Milos Gligoric
Department of Electrical and Computer Engineering, The University of Texas at Austin
{kaiyuanw,khurshid,gligoric}@utexas.edu

ABSTRACT

Java PathFinder (JPF) is a backtrackable Java Virtual Machine (JVM), which is implemented in Java and runs on a standard JVM (e.g., Oracle HotSpot). Thus, a JPF developer can use off-the-shelf Java debuggers (e.g., jdb) when debugging code that makes up JPF. JPF explores all non-deterministic executions of a given target program and monitors for property violations. To facilitate debugging of the target program, JPF can capture and replay the execution trace that leads to a property violation. While the deterministic replay is invaluable, the replay with JPF does not allow the developer to attach an off-the-shelf Java debugger to the target program (e.g., step through the application code, set breakpoints, etc.).

We present a technique, dubbed JPR, to improve the debugging experience of the JPF captured traces by migrating the JPF traces to a new format that can be executed using the standard JVM. JPR annotates each JPF trace, during the capture phase, with extra data (e.g., instruction index, instruction count, etc.); the annotated trace is then used to instrument Java bytecode to enforce the same execution trace on a standard JVM. JPR is compatible with various optimizations, e.g., state matching and partial-order reduction. We evaluated JPR on all multi-threaded Java programs in the official JPF distribution. Our results show that JPR successfully replayed all JPF traces on the standard JVM with reasonable overhead during both recording and replaying.

Keywords

JPF; Debugging; Replay; Execution Traces

1. INTRODUCTION

Multi-threaded programs are notoriously hard to get right. These programs are frequently afflicted by concurrency bugs, such as data-races, deadlocks, atomicity violations, etc. Automatically discovering these types of bugs requires sophisticated static and/or dynamic program analysis techniques (e.g., [4, 8, 7]).

Java PathFinder (JPF) is an explicit state software model checker for Java programs [19]. JPF is implemented as a backtrackable Java Virtual Machine (JVM), which explores (all) non-deterministic schedules of the given *target program* while monitoring for violations of various (user-provided) properties (e.g. deadlocks). If JPF discovers a property violation, it provides an *execution trace* that includes detailed execution information for each thread, as well as the order in which the threads were scheduled.

To debug a multi-threaded program developers need, in addition to a technique that discovers a concurrency bug, a technique to capture and *replay* the execution trace that exposed the bug [2, 6, 5, 17, 16, 20]. If such capture and replay mechanism is available, a developer can attach an *off-the-shelf debugger* during the replay to inspect the buggy program state.

While JPF can capture and replay an execution trace, it is infeasible to debug the trace with an off-the-shelf Java debugger (e.g., jdb), because JPF does not support the Java Debug Interface [11]. If a developer, working on the target program, attempts to attach a debugger to the standard JVM that is running JPF code (recall that JPF itself is implemented in Java), the developer would be able to step through the code that makes up JPF, set breakpoints, and inspect variables, but could not easily do so for the code and state of the target program; the state of the target program (including the heap) is stored in integer arrays.

We present a technique, dubbed JPR, to improve the debugging experience of the execution traces (captured by JPF) by migrating JPF traces to a new format that can be executed using a standard JVM (e.g., Oracle HotSpot). This allows the application developer to attach an off-the-shelf debugger during the replay of the trace on a standard JVM.

JPR captures the thread schedule and the transition points (e.g., instruction counts and instruction indexes) via a JPF listener [19], it uses the captured information to instrument Java bytecode of the target program (via the instrumentation package [12] and the ASM bytecode engineering library [1]) to insert scheduling events, and enforces the execution with a *thread scheduler*, which is a separate thread that takes as input the JPF captured execution trace and enables the execution of one thread at a time.

JPR is similar to an earlier work by Schuppan et al. [17], which enabled an execution of a JPF trace using a standard JVM. One of the key differences is the scheduling mechanism. The previous work modifies the target program, such that each thread, prior to the end of a transition, notifies the next thread to be executed; this approach requires modifications to the target program and substantial *modifications* to the invocations of `wait`, `notify`, and `notifyAll` primitives. On the other hand, JPR does *not* modify any method invocation, but only inserts schedule events at the end of each transition; a separate thread is used to enforce the schedule and enables one thread at a time.

We evaluated JPR on all multi-threaded examples distributed with JPF. We measured the overhead to capture additional information needed to migrate the JPF trace, as well as the overhead to replay the execution using a standard JVM. Our results show that JPR introduces negligible overhead in both phases.

2. BACKGROUND AND EXAMPLE

This section provides a brief background on the key technology utilized by JPR and illustrates JPR on an example program.

```
1  class Fork {}
2  class Philosopher extends Thread {
3    Fork left, right;
4    Philosopher(String name, Fork left, Fork right) {
5      super(name);
6      this.left = left;
7      this.right = right;
8    }
9    @Override
10   public void run() {
11     // think
12     fireEvent(4)
13     synchronized(left) {
14     fireEvent(5)
15     synchronized(right) {
16     // eat
17 }}}}
```

```
1  class DiningPhil {
2    static int nPhilosophers = 2;
3    public static void main(String[] args) {
4      if (args.length > 0) {
5        nPhilosophers = Integer.parseInt(args[0]);
6      }
7      Fork[] forks = new Fork[nPhilosophers];
8      for (int i = 0; i < nPhilosophers; i++) {
9        forks[i] = new Fork();
10     }
11     for (int i = 0; i < nPhilosophers; i++) {
12       Philosopher p = new Philosopher(
13         "Phil" + i,
14           forks[i],
15           forks[(i + 1) % nPhilosophers]);
16       p.start();
17 }}}
```

Figure 1: Dining Philosophers examples from the official JPF repository

2.1 Java PathFinder

JPF is an explicit-state model checker that implements a backtrackable Java Virtual Machine (JVM). JPF systematically executes Java programs by exploring all non-deterministic choices (e.g., thread interleavings). Each *execution trace* is a sequence of transitions and each *transition* is a sequence of bytecode instructions (executed by a single thread) that takes the program from one *state* to another; each transition ends with an observable instruction [19].

JPF *listeners* is an extension mechanism to either observe or modify a state of a program. JPF listeners are used to implement various properties that a program should (not) satisfy, e.g., deadlock detection. Listeners are executed in the same JVM as JPF, which provides an easy access to the entire program state (e.g., state of each thread). As described in Section 3.1, we implement a JPF listener to collect execution traces, including thread schedule for each trace and transition points. This is the first step towards enabling the replay of JPF execution traces on standard JVM.

2.2 ASM Bytecode Engineering Library

ASM is an all purpose Java bytecode manipulation and analysis framework [1]. It can be used to modify existing classes or dynamically create new classes. ASM provides common transformations and analysis algorithms, which allow to easily assemble custom complex transformations and code analysis tools.

In this paper, we implement a schedule event injector using the ASM library. The event injector inserts special *"fire"* event at interesting points (e.g., end of transitions) in the target classfiles according to the extracted JPF execution traces.

2.3 JPR Through An Example

We use a traditional example – dining philosophers – to illustrate synchronization issues, execution traces, and replay. A group of philosophers are at a round table with bowls of spaghetti in the middle. Forks are placed between each pair of adjacent philosophers. Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when holding both (left and right) forks. Each fork can be held, at any given time, by only one philosopher. After a philosopher finishes eating, he/she needs to put down both forks so that the forks become available to others.

Figure 1 shows the source code for the dining philosophers available in the official JPF source code repository; we ignore the blue part for now (i.e., `fireEvent` invocations). For simplicity, we execute this program with only two philosophers. Figure 2 shows a part of the state space exploration if we run JPF against the exam-

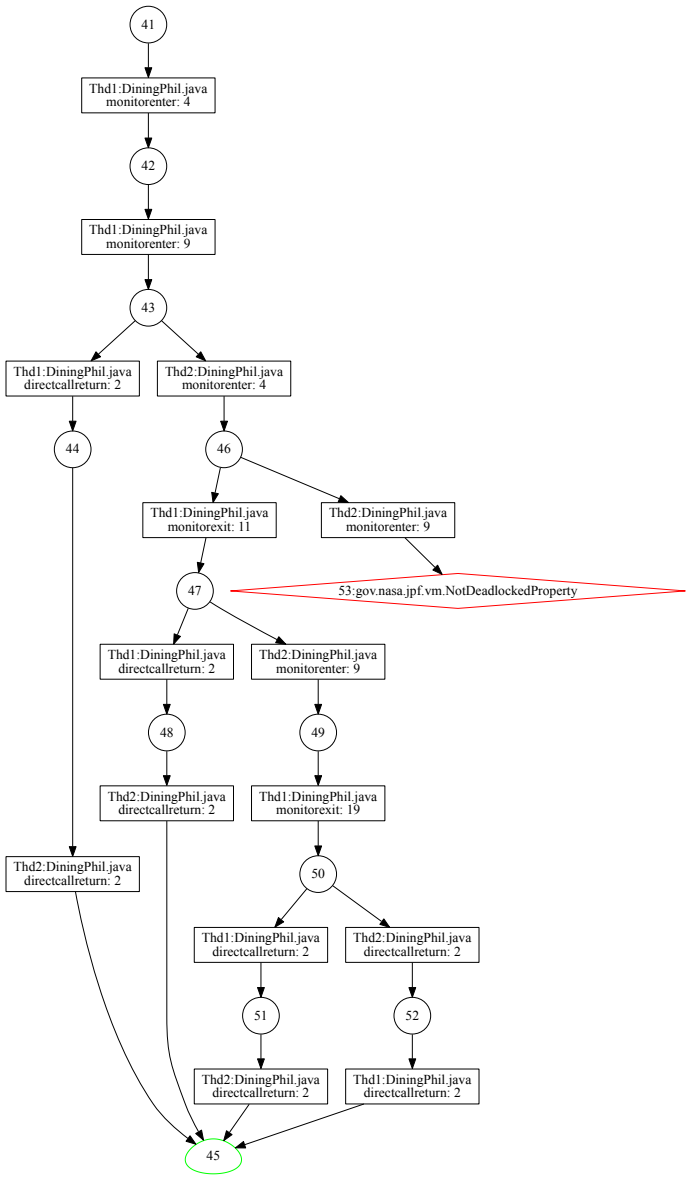

Figure 2: Part of JPF state space graph for DiningPhil

ple. Each rectangle represents a transition and each circle represents a program state. The circle (at the very bottom) highlighted in green denotes termination, and the diamond node highlighted in red denotes a state which violates the JPF built-in properties. The second line in each transition shows the bytecode instruction

```
1  // Start of Main...
2  // Start of Thread Phil0.
3  Philosopher,run()V,monitorenter,4,Phil0,1,4 // fireEvent(4)
4  // Start of Thread Phil1.
5  Philosopher,run()V,monitorenter,4,Phil1,1,4 // fireEvent(4)
6  Philosopher,run()V,monitorenter,9,Phil0,1,5 // fireEvent(5)
7  Philosopher,run()V,monitorenter,9,Phil1,1,5 // fireEvent(5)
```

Figure 3: Part of Dining Philosophers deadlock schedule

which leads to a choice generator (potential context switch point, but not the instruction to be executed next). The first line in each transition gives the current executing thread and the source file of that bytecode instruction. JPF explores five paths according to the state space graph and only one of the execution paths leads to a deadlock. The deadlock happens when both Phil0 and Phil1 have started and entered the `synchronized(left)` block but cannot enter the `synchronized(right)` block. We configured JPF to use state matching and partial order reduction, which impacted the shape of the explored state space.

Our customized listener extracts scheduling information for each execution path and saves it to a file. We collect the first instruction in each transition and the number of times it has been executed on the current path before leading to the transition. Our schedule injector uses these transition information to insert "fire" events before each bytecode instruction that leads to a transition. These fire events block threads, which are then enabled (one at a time) by the thread scheduler. The thread scheduler enables a thread when all threads are blocked.

Figure 3 shows a part of the schedule that leads to a deadlock. Each line contains information about the first instruction in a transition, obtained from Figure 2. Each line includes the following information:

`ClassName`, `MethodName`, `InstructionMnemonic`, `InstructionIndex`, `ThreadName`, `NumOfExecutedTimes`, `EventId`

where `ClassName` and `MethodName` are the class and method (name + signature) the bytecode instruction belongs to. `InstructionMnemonic` is a mnemonic of the bytecode instruction (e.g., `monitorenter`). `InstructionIndex` is the offset of the instruction in the current method. `ThreadName` is the name of a thread that executes the instruction; we assume that each thread is given a unique name that is constant across several runs. `NumOfExecutedTimes` is the total number of times the instruction (uniquely identified with the class name, method name and index) has been executed on the current execution path. We use this information to enforce orders of instructions if some of them are executed multiple times in a single run (e.g. in a `for/while` loop). `EventId` is the unique id assigned to the bytecode instruction. For example, the `monitorenter` instruction at index 4 (class: `Philosopher`; method: `run`) represents `synchronized(left)` and `monitorenter` (same class and method) at index 9 represents `synchronized(right)`.

To reproduce the deadlock in Figure 2 on a standard JVM, JPR inserts an invocation to the "fire" event before each `monitorenter` instruction according to the schedule in Figure 3. Note that we insert these invocations on the bytecode level, although we show these calls in the source code in Figure 1. Each thread will block at the fire event when the instruction is executed the appropriate number of times based on the recorded schedule. Our thread scheduler iterates over each line in the captured trace and unblocks only the relevant thread until no more events are available in the trace. In our running example, Phil0 first starts and blocks at `fireEvent(4)` (line 3 in Figure 3), then Phil1 starts and blocks

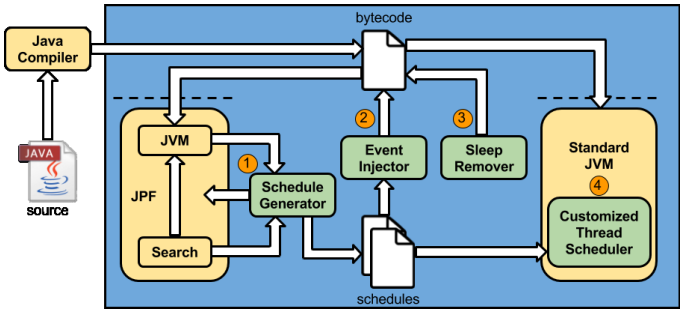

Figure 4: JPR overview

at `fireEvent(4)` (line 5). Both threads are blocked before `synchronized(left)`. Our thread scheduler then enables Phil0, because that is the next event in the captured trace. Phil0 tries to grab its `left` lock and blocks again at `fireEvent(5)` (line 6). The thread scheduler then enables Phil1. Phil1 tries to grab its left lock and blocks again at `fireEvent(5)` (line 7). At this point, our scheduler detects that no thread is alive and throws an instance of the `DeadlockException` type.

3. TECHNIQUE

This section presents the details behind JPR, which captures JPF execution traces and replays those traces on a standard JVM. Figure 4 gives the high-level overview of our tool. The blue area shows the scope of our tool. The yellow areas show the existing tools. We implemented the green areas to capture and replay the execution traces. The input to JPR is a compiled Java program; the compiled program should either have a main method or include a test method written in the JUnit style. The following subsections describe the key components of JPR.

3.1 Schedule Generator

JPR runs the given target program on top of JPF and captures the execution traces with a custom listener. The listener internally builds a state space graph during the exploration; once the exploration is finished, the listener traverses the graph and writes each execution trace into a separate file. Note that we can easily configure JPR to capture only execution traces that lead to a property violation.

JPR builds on the idea that knowing the first instruction in each transition, and the thread that executes the instruction, is sufficient to replay the trace. Thus, JPR inserts one fire event prior to each instruction that starts a new transition. Our technique currently creates schedules that enforce the execution of one thread at a time (which closely follows the way in which JPF explores the state space); we leave it for our future work to explore generating schedules that allow the execution of independent transitions by multiple threads simultaneously.

There are three ways in which a trace may end during the JPF exploration. First, all threads successfully finish the execution. In this case, no special instructions are needed in the generated trace. Second, there was a state matching [19]. In this case, the generated trace includes a special flag, which will be interpreted by our thread scheduler (Section 3.4). Third, there was a property violation. In this case, the generated trace includes a fire event before the next instruction executed by each thread. This blocks the threads at the point when a property is going to be violated.

To avoid generating unnecessary scheduling events, we only consider instructions that belongs to user-given classes and ignore instructions from the standard Java library. Furthermore, when

JPF finishes the exploration, JPR detects and removes duplicate generated traces. (Note that JPF explores a substantial number of duplicate traces despite various optimizations.)

3.2 Event Injector

The Event Injector takes as input the set of generated traces (in the previous step) and instruments bytecode of the target program; we modify the original class file as shown in Figure 4. For each entry in the generated trace, the Event Injector finds the name of the class, the name of the method, and the index of the bytecode instruction. Next, it locates the classfile that corresponds to the class in the event and inserts a "fire" event in the method at appropriate index (before the instruction). The argument to the fire event is the unique instruction id; each instruction that starts at least one transition is assigned a unique id by JPR.

3.3 Sleep Remover

The Sleep Remover replaces `Thread.sleep(long)` method invocation with the `POP2` bytecode instruction. Since we are enforcing the thread execution, sleeps (or any other timed delays) have no impact on the schedule and therefore can be safely ignored. Note that we need to insert a bytecode instruction to remove the arguments that were intended for the sleep method invocation. In addition to sleep, we also modify timed wait invocations in a similar way. It is interesting that removing sleeps may result in an execution that is faster than non-enforced execution with sleeps [10] as we will illustrate in our evaluation.

3.4 Thread Scheduler

Finally, JPR runs the instrumented target program on a standard JVM. The instrumented code will spawn, when the instrumented classes are loaded, a thread scheduler, which is implemented as a daemon thread.

The Thread Scheduler ensures that only one thread is enabled at any given time. The Thread Scheduler loads one of the generated execution traces and enables the next thread as specified in the trace when all threads are blocked. If all threads are blocked and the next event in the trace is a special flag to indicate that state-matching was encountered during JPF exploration, the Thread Scheduler simply invokes `System.exit(0)`. If all threads are blocked, then the Thread Scheduler reports a deadlock property violation (by throwing the `DeadlockException`). Finally, if any of the threads throws an unexpected exception, then the application terminates with a property violation.

4. EVALUATION

In this section, we present our initial evaluation of JPR. Specifically, we evaluated JPR using all multi-threaded programs that are available in the official JPF distribution, as well as a couple of test cases from the Apache Commons Collections project.

Table 1 shows our experiment results. The "Subject" column lists all subjects used in the experiment. The "Total" column shows the total number of execution traces generated by the JPF listener. The "Unique" column shows the unique number of schedules out of "Total". The "Failed" column shows the number of schedules that violates a property (including assertion failures). The "JVM" column shows the time to execute a subject on Oracle's JVM without enforcing the thread schedule. The "JPF" column shows the time to run the subject on top of JPF without our listener. The "Generator" column represents the time to generate all schedules without duplicates. The "Injector" column shows the time to inject fire events, i.e., to instrument bytecode. The "TestAll" column shows the time to replay all unique traces with our Thread Scheduler on Oracle's JVM.

We run all experiments on a Intel(R) Core(TM) i7-4870HQ CPU @ 2.5GHz with 16GB of RAM, running OS X El Capitan and Java 1.8.0_121.

The official JPF distribution comes with several multi-threaded subjects, including `BoundedBuffer`, `DiningPhil`, `OldClassic` and `Racer`. The rest of the subjects used in our study include test cases from Apache Commons Collections used in prior work on multi-threaded testing [10]. The name for these subjects in the table consists of the test class name and test method name.

As shown in Table 1, the total numbers of execution traces for JPF examples are all less than or equal to 5 and every generated trace is unique. For subjects taken from Apache Collections, the total numbers of execution traces are larger (from 22 to 28). Less than half of the traces for Apache tests are duplicates. JPF reports a property violation (either a deadlock or an exception) for each of the JPF examples. No violation of properties is reported for Apache Collections tests.

For all the multi-threaded programs from the JPF distribution, the execution time on Oracle JVM is mostly less than 1 second. However, `BoundedBuffer` and `OldClassic` never terminate and their execution will always end up with deadlocks. `Racer` invokes `Thread.sleep(1000)`, therefore, it takes slightly more time than 1 second to complete. For JPF subjects, the JPF execution time is always less than 1 second. For Apache Collections tests, the JPF execution time is a slightly over 1 second. With our schedule generator listener, the JPF execution slightly increases.

In all cases, the Event Injector takes almost constant time to insert fire events in the bytecode. This is mainly because the number of unique schedules in our subjects is small. We do not list the sleep remover time because it is negligible compared to other steps.

The time to replay *all* thread schedules on the Oracle's JVM is less than 1 second for each JPF subject. For Apache tests, it takes about 3 seconds to replay. Although replay may introduce some overhead (due to the Thread Scheduler and sequential execution), this is not a concern in the envisioned use cases when a developer is trying to replay and debug the target program.

We use the exploration with no listener as a baseline to compute the slowdown for the schedule generator. We found that on average our schedule generator slows down JPF execution with a factor of 1.08. We also compared the average replay time for each schedule with the time to run the program without our thread scheduler. `BoundedBuffer` and `OldClassic` are excluded because we do not have the baseline (due to the deadlock). Interestingly, our replay engine is, on average, faster than running Oracle JVM with a factor of 0.93 because `Racer` invokes `Thread.sleep` and we safely remove that invocation (Section 3.3). To validate JPR, we manually checked all schedule replays and found that they are consistent with JPF explorations.

5. RELATED WORK

There is a vast amount of work on model checking and record and replay; here, we only mention a few projects that are closely related. Pan and Linton [15] record and replay the values of shared objects at each access. LeBlanc and Mellor Crummey implemented a scheme that records the order of accesses based on version counters in Instant Replay [13]. Tai et. al. [18] use

Table 1: Numbers of Execution Traces and Execution Time

Subject	#Traces			Time [s]				
	Total	Unique	Failed	JVM	JPF	Generator	Injector	TestAll
BoundedBuffer	1	1	1	-	0.542	0.629	0.132	0.175
DiningPhil	5	5	1	0.099	0.531	0.563	0.127	0.617
OldClassic	5	5	1	-	0.514	0.538	0.139	0.630
Racer	4	4	1	1.104	0.492	0.504	0.130	0.431
TestBlockingBuffer-testGetWithAdd	22	14	0	0.209	1.279	1.390	0.138	3.070
TestBlockingBuffer-testGetWithAddAll	25	15	0	0.198	1.298	1.417	0.129	3.293
TestBlockingBuffer-testRemoveWithAdd	25	15	0	0.216	1.280	1.395	0.132	3.262
TestBlockingBuffer-testRemoveWithAddAll	28	16	0	0.211	1.318	1.448	0.157	3.573

a source-to-source transformation of an Ada program to replay a sequence of synchronization events. Their scheme assumes that shared variables are protected appropriately. Netzer [14] presents an algorithm that reduces the amount of data generated by these approaches. Choi et. al. [3] presents DejaVu which modifies JVM and assigns a global order to all "critical events" for deterministic replay. Huang et. al. [9] proposed LEAP, a local order based replay tool that reduces run-time overhead compared to global order based replay tools.

6. CONCLUSION

We presented JPR, a technique to enable replaying of JPF traces on a standard JVM. JPR records transition points for each trace explored by JPF, and it uses the recorded data to instrument bytecode of the target program to insert flags to indicate the beginning of a transition or a property violation. When executed on a standard JVM, the instrumented program sends notifications to a thread scheduler than enables one thread at a time (when all threads are blocked) to enforce the original trace. We evaluated JPR with all multi-threaded programs that are available in the JPF distribution, as well as several tests from the Apache Commons Collection project. Our results showed that we were able to successfully replay the JPF traces with reasonable overhead.

Acknowledgments

We thank the fellow students of EE 382V (Software Evolution) at The University of Texas at Austin for discussions on the material presented in this paper. This research was partially supported by the US National Science Foundation under Grants Nos. CNS-1239498, CCF-1319688, and CCF-1652517.

7. REFERENCES

[1] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[2] J.-D. Choi and H. Srinivasan. Deterministic replay of Java multithreaded applications. In *SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 48–59, 1998.

[3] J.-D. Choi and H. Srinivasan. Deterministic replay of java multithreaded applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 48–59, 1998.

[4] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[5] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded Java programs. *Concurrency and Computation: Practice and Experience*, pages 485–499, 2003.

[6] A. Georges, M. Christiaens, M. Ronsse, and K. De Bosschere. JaRec: A record/replay system for multi-threaded Java applications. In *Parallel Computing: Grids and Applications*, pages 125–146. 2002.

[7] P. Godefroid. Model checking for programming languages using VeriSoft. In *Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[8] G. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, first edition, 2003.

[9] J. Huang, P. Liu, and C. Zhang. LEAP: Lightweight deterministic multi-processor replay of concurrent Java programs. In *International Symposium on Foundations of Software Engineering*, pages 207–216, 2010.

[10] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, and D. Marinov. Improved multithreaded unit testing. In *International Symposium on Foundations of Software Engineering*, pages 223–233, 2011.

[11] Java Debug Interface, 2017. `http://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/index.html`.

[12] Java Tool Interface, 2017. `https://docs.oracle.com/javase/8/docs/technotes/guides/jvmti/index.html`.

[13] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, pages 471–482, 1987.

[14] R. H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Workshop on Parallel and Distributed Debugging*, pages 1–11, 1993.

[15] D. Z. Pan and M. A. Linton. Supporting reverse execution for parallel programs. In *Workshop on Parallel and Distributed Debugging*, pages 124–129, 1988.

[16] D. A. Popescu, E. D. Tirsa, M. I. Andreica, and V. Cristea. An application-assisted checkpoint-restart mechanism for Java applications. In *International Symposium on Parallel and Distributed Computing*, pages 190–197, 2013.

[17] V. Schuppan, M. Baur, and A. Biere. JVM independent replay in Java. *Electron. Notes Theor. Comput. Sci.*, pages 85–104, 2005.

[18] K.-C. Tai, R. H. Carver, and E. E. Obaid. Debugging concurrent ada programs by deterministic execution. *IEEE Trans. Softw. Eng.*, pages 45–63, 1991.

[19] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, pages 203–232, 2003.

[20] G. Xu, A. Rountev, Y. Tang, and F. Qin. Efficient checkpointing of Java software using context-sensitive capture and replay. In *International Symposium on Foundations of Software Engineering*, pages 85–94, 2007.