Java Trace Guide

Project Proposal

Tomáš Poch
poch@dsrg.mff.cuni.cz

A software model checker allows automatic detection of flaws in software systems by inspecting all states a running program can reach. In the context of the Java development, Java Pathfinder (JPF) [3], state of the art model checker developed by NASA, is available and capable of detecting assertions, race conditions, and other errors in Java projects of a reasonable size [5]. However, the communication with the user is command line based and lacks user-friendliness, efficiency, and interactivity of todays mature development tools. The goal of this project is to improve usability of JPF by its integration into a modern Java IDE.

The code in the grey box on the right (taken from [8]) is a typical example of race condition behavior. There are two threads and the result of computation depends on their interleaving. If line (2) is executed before line (4), division by zero occurs.
Such code is hard to test and debug in an IDE, since the erroneous behavior can be experienced just once in many executions or not at all.

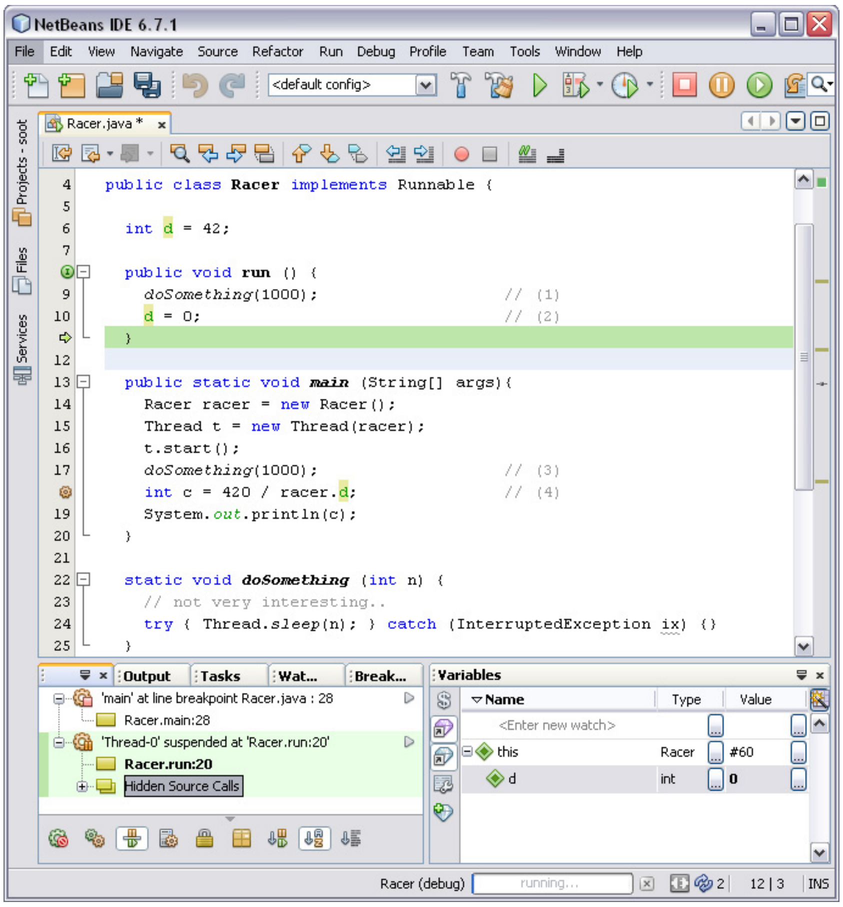
```java
public class Racer implements Runnable{
  int d = 42;
  public void run () {
    doSomething(1000); // (1)
    d = 0; // (2)
  }

  public static void main (String[] a)
  {
    Racer racer = new Racer();
    Thread t = new Thread(racer);
    t.start();
    doSomething(1000); // (3)
    int c = 420 / racer.d; // (4)
    System.out.println(c);
  }

  static void doSomething (int n) {
    // not very interesting..
    try { Thread.sleep(n); }
    catch (InterruptedException ix) {}
  }
}
```

```
> bin/jpf Racer
JavaPathfinder v4.1 - (C) 1999-2007 RIACS/NASA Ames
========================================== system under test =
application: /Users/pcmehlitz/tmp/Racer.java
========================================== search started:   =
error #1
gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty
java.lang.ArithmeticException: division by zero
        at Racer.main(Racer.java:20)
===================================================== trace #1
-------------------------------------- transition #0 thread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>main}
      [282 insn w/o sources]
 Racer.java:15                : Racer racer = new Racer();
 Racer.java:1                 : public class Racer
     [1 insn w/o sources]
 Racer.java:3                 : int d = 42;
 Racer.java:15                : Racer racer = new Racer();
 Racer.java:16                : Thread t = new Thread(racer);
     [51 insn w/o sources]
 Racer.java:16                : Thread t = new Thread(racer);
 Racer.java:17                : t.start();
-------------------------------------- transition #1 thread: 0
...
-------------------------------------- transition #2 thread: 1
...
-------------------------------------- transition #3 thread: 1
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {main,>Thread-0}
 Racer.java:11                : d = 0;                // (2)
 Racer.java:12                : }
-------------------------------------- transition #4 thread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>main}
  Racer.java:20               : int c = 420 / racer.d;// (4)
```

The result of JPF analysis is shown in the yellow box on the left. Since JPF explores all interleavings, the erroneous behavior is eventually found and reported to the user. The output contains the error description and the sequence of steps leading to the error – *error trace*. Notice transition #3, where line (2) is executed prior to line (4), which takes place in transition #4.
One can imagine that in more complex cases the traces be pretty long, contain many steps unrelated to the bug and it can be quite laborious to analyze the trace step by step.

At the same time, the modern Java IDEs (e.g. Netbeans, Eclipse) are excellent in presentation of state of a paused program. The user is allowed to inspect the call stack of each thread, variable values, etc., and execute the program slowly forward by individual steps.

The screenshot of Netbeans IDE presents the state of the example right after transition #3. There are two threads – the main thread is suspended before line (4) and Thread-0 just passed through line (2). The information presented to the user this way is identical to the information available in JPF at a certain point of the trace.
Moreover, profiler extensions of the modern Java IDE present certain metrics (e.g. time, number of invocations) related to each method declaration. Similar metrics can be easily identified in JPF, related to how much time or space JPF requires to analyze a method. Such information is not currently revealed by the JPF at all, although it can be valuable for program optimization wrt. checking (writing programs which can be analyzed by JPF).



```
public class Racer implements Runnable {

  int d = 42;

  public void run () {
    doSomething(1000);                   // (1)
    d = 0;                               // (2)
  }

  public static void main (String[] args){
    Racer racer = new Racer();
    Thread t = new Thread(racer);
    t.start();
    doSomething(1000);                   // (3)
    int c = 420 / racer.d;               // (4)
    System.out.println(c);
  }

  static void doSomething (int n) {
    // not very interesting..
    try { Thread.sleep(n); } catch (InterruptedException ix) {}
  }
```

```
'main' at line breakpoint Racer.java : 28
    Racer.main:28
'Thread-0' suspended at 'Racer.run:20'
    Racer.run:20
    Hidden Source Calls
```

Name	Type	Value
<Enter new watch>		
this	Racer	#60
d	int	0

Since the Java platform features standard debugging interface (JPDA) and profiler interface (JVMTI), the suggested approach is to implement those interfaces for JPF. Such solution allows using JPF in a Java IDE instead of a common JVM. However, to be able to navigate along the trace, additional interface has to be introduced (obviously, there is no support for "step-back" button in JPDA).

Expected team size: 3-4 students

Duration: 9 months

References:

[1] E. Clarke, O.Grumberg and D. Peled: Model Checking, ISBN-13: 9780262032704
[2] J. Gosling, B. Joy,G. Steele, G. Bracha: The Java(TM) Language Specification (3rd Edition)
[3] W. Visser, K. Havelund, G. Brat, S. Park and F.Lerda.: Model Checking Programs Automated Software Engineering Journal.Volume 10, Number 2, April 2003.
[4] A. Myatt, B. Leonard and G. Wielenga: Pro NetBeans IDE 6 Rich Client Platform Edition"
[5] G. Brat, D. Drusinsky, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, A. Venet, W. Visser, R. Washington: Experimental Evaluation of Verification and Validation Tools on Martian Rover Software. Formal Methods in System Design 25(2-3): 167-198 (2004)

[6] Java Platform Debugging Architecture: http://java.sun.com/javase/6/docs/technotes/guides/jpda/index.html
[7] Java Virtual Machine Tool Interface: http://java.sun.com/javase/6/docs/technotes/guides/jvmti/index.html
[8] Java Path Finder: http://babelfish.arc.nasa.gov/trac/jpf