Visualization of Execution Paths for Concurrent Programs

Andrei Eleshevich
Peter the Great St. Petersburg Polytechnic University
29 Polytechnicheskaya st.
195251 St. Petersburg Russia
ordronus@gmail.com

Marat Akhin
Peter the Great St. Petersburg Polytechnic University
29 Polytechnicheskaya st.
195251 St. Petersburg Russia
akhin@kspt.icc.spbstu.ru

ABSTRACT

Understanding concurrent program behaviour is very hard even for experienced developers, let alone students, because of different possible thread interleavings, which are often not so obvious prima facie. In this paper we present a visualization system intended to help students in this difficult task. It collects all possible execution traces with the help of Java PathFinder and visualizes them as UML sequence diagrams, thus allowing one to discern possible execution schedules for a given concurrent program. We believe this kind of visualization could be of great use in teaching concurrent programming.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*computer science education*; D.2.5 [**Software Engineering**]: Testing and Debugging—*monitors, testing tools, tracing*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*

Keywords

Education, Visualization, Concurrency

1. INTRODUCTION

It is quite difficult to teach programming to students [4], and it is even more so with concurrent programming. At the same time, concurrency is one of the most required skills for a developer nowadays, as more and more problems stop fitting on a single core every year.

The main problem with teaching concurrent programming is that in a single-threaded program execution order is easily defined and clear to the naked eye, whereas in a multi-threaded program, because of context switches which can happen arbitrarily, there are many possible thread interleavings, some of which might lead to bugs such as race conditions and deadlocks.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IWAIT '15, Oct. 8–10, 2015, Aizu-Wakamatsu, Japan.
Copyright 2015 University of Aizu Press.

In many cases concurrency errors stay hidden most of the time and surface only in several specific thread interleavings (so called Heisenbugs). When teaching, this can lead to students becoming overcautious and starting to abuse synchronization, which in turn nullifies the advantages of concurrent programming.

We propose to enhance concurrent programming teaching by providing a visualization tool which captures all possible execution orders and shows them to the student, so that she can explore them and reason about side effects of different thread interleavings on the program results. The prototype implementation is based on Java PathFinder (JPF) [5] and uses it to collect interesting execution traces which are later rendered as UML sequence diagrams; if JPF finds a possible concurrency error, it is shown to the student together with the schedule that caused it.

The rest of the paper is organized as follows. Section 2 talks about related work in the area of visualization of concurrent program. We present our approach, the architecture of our prototype tool and show the diagram of a program containing a race condition in sections 3 and 4. Possible future work and conclusions are discussed in section 5.

2. RELATED WORK

There have been a lot of research in the area of concurrent program visualization for program understanding. JThreadSpy [3] uses instrumentation to collect execution traces and renders them as UML sequence diagrams, in which it is very similar to our approach. The differences are that it only instruments class methods and uses runtime information (therefore cannot explore all reachable thread interleavings). Our approach captures all information about program execution and explores every possible execution trace due to the power of JPF.

Atropos [1] also uses instrumentation for trace collection, but it records all operations and data being manipulated by them. It then creates a data dependence graph (DDG) for visualization; the user can analyze DDG to find which data dependencies caused an error in the program. Atropos does not allow the user to analyze different possibilities, as it uses runtime information to build the DDG, i.e. captures only one concrete execution.

Thread Interleaving Explorer (TIE) [2] is a tool most similar to ours and actually was the main inspiration behind this work. It is a debugger for erroneous execution JPF traces that allows the user to select a threading schedule and study its influence on the program execution. Unlike our approach, it focuses on exploring one (erroneous) trace at a time and

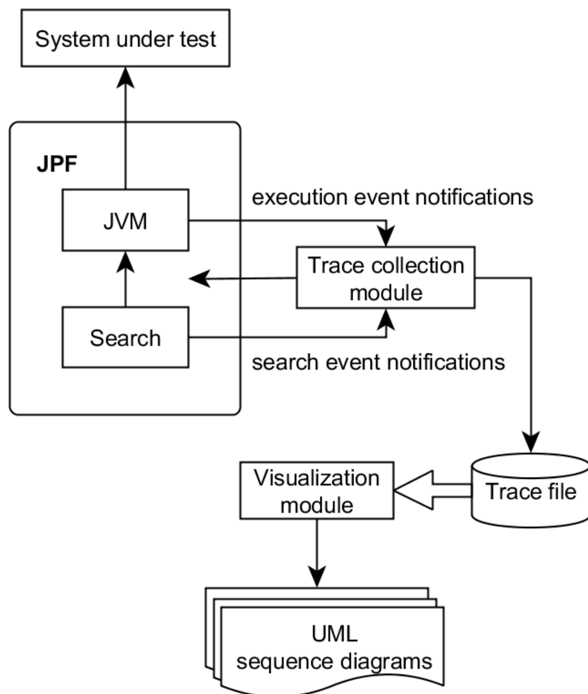

Figure 1: Prototype architecture

cannot visualize interactions between context switches (i.e. show several executions at the same time in detail).

3. APPROACH

Our main idea is to exhaustively exercise all possible thread interleavings using JPF[1]. By employing various model checking techniques, JPF can efficiently backtrack to already visited program states (should they appear during exploration) without the need to re-execute the program. We collect the information about these unique program states and combine them later to create different thread schedulings. These schedulings are presented to the user, so that she can study them to gain insights to how her concurrent program actually works.

We utilize well-known UML sequence diagrams (USD) to visualize possible program executions and follow their standard notation. USD objects correspond to program objects, USD lifelines — to different object activities. Static methods and fields are represented via special dummy objects (one per class).

Method calls are mapped to USD messages from the caller to the callee objects; synchronized calls are represented with solid arrows, open arrows are used otherwise. Object allocations and thread starts are shown as special method calls (which maps nicely to their actual semantics).

Fields are viewed as properties with dummy `get`/`set` methods that are also visualized as special method calls (for which we track the current field values). At the moment, our prototype supports only fields of primitive types and does not handle `volatile`/`final` fields w.r.t. concurrency quirks; this

[1]Here JPF is only a tool, our approach can be based on any other system that provides information about possible concurrent executions.

Listing 1: Racer program

```
public class Racer extends Thread {

  int d = 42;

  public void run() {
    doSomething(1001);
    d = 0;
  }

  public static void main(String[] args) {
    Racer racer = new Racer();
    racer.start();

    doSomething(1000);
    int c = 420 / racer.d;
    System.out.println(c);
  }

  static void doSomething(int n) {
    try {
      Thread.sleep(n);
    } catch (InterruptedException ix) {}
  }
}
```

is one of the possible areas for future work.

To represent thread information, we extended USD by adding color annotations to the activation boxes — every thread is assigned a unique color that is used to mark methods run by the corresponding thread. If a thread execution is paused, it is shown by a darker shade of the thread's color.

Another USD extension is to support execution branches. If a method execution creates several interesting schedulings, its activation box is labeled with a number of branch options JPF found during program exploration. From this label the user can open another possible scheduling in a separate window.

4. PROTOTYPE ARCHITECTURE

Our prototype implementation consists of two modules: trace collection module and visualization module 1. Let us discuss them in more detail.

4.1 Trace Collection

We collect possible execution traces from JPF using a custom JPF listener and capture such information as method invocations, field accesses and object allocations. We also record different execution branches and their unique identifiers (UID) and context switches possible in the program. An execution branch in our approach roughly corresponds to JPF choice during its model checking phase; if several executions lead to the same JPF state, we consider them to be the same branch (though there would be several different thread interleavings leading to it).

4.2 Visualization

After collecting all execution traces, visualization is done using a stand-alone GUI program. It carefully aggregates possible execution branches (by unpacking branch combinations possible in the program to execution paths) and builds UML sequence diagrams from them. The first diagrams represent correct (w.r.t. JPF) executions, the last one shows an execution with possible errors. The user can open additional diagrams by selecting a branching point and picking one of the possible branching options there.

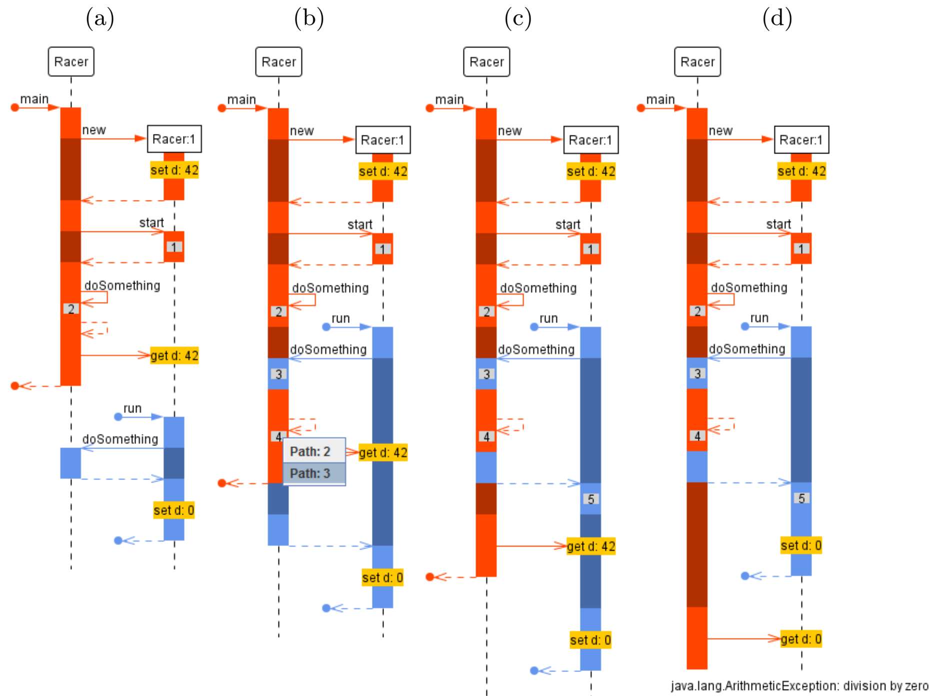

Figure 2: Execution paths for Racer program

4.3 Example

Let us consider a classic Racer example — a program with race condition shown in listing 1.

The results of thread interleaving exploration for Racer program using our prototype tool are shown in figure 2. You can see that several execution paths were explored (figures 2a–2c) before finding an error (figure 2d).

The bug is a race condition between read access to `racer.d` in `main` method and write access to it in `run` method. As they are not synchronized, `d = 0` could happen before division which would cause a division-by-zero error. As seen from figure 2, JPF exploration is depth-first as it tries to complete the execution before backtracking to a branching point. If a branching point creates non-distinct executions (up to the next branching point), we consider it to be non-interesting and does not show it in the interface.

5. CONCLUSIONS AND FUTURE WORK

We proposed an approach to the visualization of concurrent executions based on JPF for thread interleaving exploration. It creates UML sequence diagrams which allows the students to easily reason about different concurrent executions and witness the possible bugs first-hand.

Of course, our prototype leaves much to be desired. For example, visualization lacks object or method filters which would help with reducing the sequence diagram' size. The sequence diagram itself could be improved by adding explicit notation for advanced synchronization primitives, e.g. `synchronized` blocks, `wait`/`notify` calls or `volatile` accesses.

Another idea would be to employ slicing to compact the diagram leaving only variables and interactions interesting to the user.

6. REFERENCES

[1] J. Lönnberg, M. Ben-Ari, and L. Malmi. Java replay for dependence-based debugging. In *Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, PADTAD '11, pages 15–25, New York, NY, USA, 2011. ACM.

[2] G. Maheswara, J. S. Bradbury, and C. Collins. TIE: An interactive visualization of thread interleavings. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS '10, pages 215–216, New York, NY, USA, 2010. ACM.

[3] G. Malnati, C. M. Cuva, and C. Barberis. JThreadSpy: Teaching multithreading programming by analyzing execution traces. In *Proceedings of the 2007 ACM Workshop on Parallel and Distributed Systems: Testing and Debugging*, PADTAD '07, pages 3–13, New York, NY, USA, 2007. ACM.

[4] E. Pyshkin. Teaching programming: What we miss in academia. In *Software Engineering Conference in Russia (CEE-SECR), 2011 7th Central and Eastern European*, pages 1–6. IEEE, 2011.

[5] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engg.*, 10(2):203–232, Apr. 2003.