Deadlock detection of Java Bytecode

Abel Garcia Elena Giachino Cosimo Laneve
Dept. of Computer Science and Engineering, University of Bologna – INRIA FOCUS

Deadlocks are a common threat of concurrent programs, which occur when a set of threads are blocked, due to each attempting to acquire a lock held by another. Such errors are difficult to detect or anticipate, since they may not occur during every execution, and may have catastrophic effects for the overall functionality of the software system.

At the time of writing this paper, the *Oracle Bug Database*[1] reports more than 40 unresolved bugs due to deadlocks, while the *Apache Issue Tracker*[2] reports around 400 unresolved deadlock bugs. These two applications are written in `Java`, a mainstream programming language in a lot of domains, ranging from system applications and databases to web and cloud applications.

The objective of our research is to design and implement a technique capable of detecting potential deadlock bugs *at compilation time* and, to this aim, we propose an end-to-end automatic static analysis tool for deadlock detection of `Java` programs. Our implementation handles all the features of `Java`, including threads, constructors, arrays, exceptions, static fields, interfaces, runtime downcasts, and dynamic data types.

Deadlock detection of `Java` programs is an hard and titanic task. It is hard because concurrency in `Java` is modelled by threads that may perform read/write operations over shared variables. The order in which concurrent operations of this kind are scheduled depends on the scheduling strategy implemented in the Java Virtual Machine (`JVM`). Therefore deadlocks may not occur during every execution. It is titanic because `Java` is a full-fledged programming language without a reference formal semantics, it is verbose, and it has a lot of libraries that are written directly in machine code.

To reduce the complexity of our work, we decided not to address the `Java` language, but to consider the `Java` bytecode, namely 198 instructions that are the compilation target of every `Java` application. `Java` bytecode is quite simple and has a reference semantics that is defined by the `JVM` behaviour. Type systems for `Java` bytecode have been studied in the past for demonstrating the correctness of the bytecode verifier [4, 1, 3].

We have defined an inference system that extracts abstract models out of `Java` bytecodes. This inference system consists of a number of rules which are identical for most of the instructions, these rules differ mainly for the operations that have effects in the synchronization process, namely invocations, locks acquisition and release, object manipulation and control flow operations.

[1] http://bugs.java.com/
[2] https://issues.apache.org/jira

Abstract models extracted from `Java` bytecodes are infinite state models that define dependencies between threads. In [2] we demonstrated that deadlock detection in these models is decidable and we also designed an algorithm. This algorithm and the inference system compose our tool, which we are experimenting on a number of `Java` libraries.

The technique is complemented by a prototype that also exhibits the executions causing deadlock and allows us to analyse false positives.

References

[1] Stephen N. Freund and John C. Mitchell. A type system for the java bytecode language and verifier. *J. Autom. Reasoning*, 30(3-4):271–321, 2003.

[2] Elena Giachino, Naoki Kobayashi, and Cosimo Laneve. Deadlock analysis of unbounded process networks. In *Proceedings of 25th International Conference on Concurrency Theory CONCUR 2014*, volume 8704 of *Lecture Notes in Computer Science*, pages 63–77. Springer, 2014.

[3] Cosimo Laneve. A type system for JVM threads. *Theoretical Computer Science*, 290(1):741 – 778, 2003.

[4] Raymie Stata and Martin Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, January 1999.