OMEN: A Tool for Synthesizing Tests for Deadlock Detection [*]

Malavika Samak [†]
Indian Institute of Science, Bangalore
malavika@csa.iisc.ernet.in

Murali Krishna Ramanathan [‡]
Indian Institute of Science, Bangalore
muralikrishna@csa.iisc.ernet.in

Abstract

Designing and implementing *thread-safe* multithreaded libraries can be a daunting task as developers of these libraries need to ensure that their implementations are free from concurrency bugs, including deadlocks. Developing multithreaded tests for this purpose is significantly challenging. In this demo, we will demonstrate our tool (OMEN) for synthesizing deadlock-inducing multithreaded tests for Java libraries. The input to OMEN is the library implementation under consideration and the output is a set of deadlock revealing multithreaded tests.

1. Introduction

Thread-safe [3] libraries are beneficial as the developers of the client programs need not consider the intricacies of the issues pertaining to multithreading and yet accrue the benefits of multithreading. However, designing such libraries can be challenging.

```
class A {                           class Test {
    synchronized foo (A a) {            void testFoo(A a1, A a2) {
      synchronized (a) {}                  a1.foo(a2)
    }                                    }
}                                   }
```

Figure 1. Illustrative example.

Consider the simple example shown in Figure 1. It presents the implementation of method `foo` in class `A`. When a client, `testFoo`, invokes `foo` as shown in the figure, a lock on a_1 is acquired followed by a lock on a_2. The implementation of `A` is *not* thread-safe because a deadlock can occur under certain scenarios when `foo` is called without holding appropriate lock(s). For example, if two threads invoke `testFoo(`a_1,a_2`)` and `testFoo(`a_2,a_1`)` concurrently, then a deadlock may manifest in some execution. This is because the first thread may attempt to acquire a lock on a_2 while holding a lock on a_1 and the second thread may attempt to acquire a lock on a_1 while holding a lock on a_2.

If `testFoo` is executed by a single thread, a dynamic deadlock detector[1, 5] will not detect any deadlock in the corresponding execution. If we synthesize method sequences that can be executed concurrently in a *random* manner and have the deadlock detector analyze the corresponding execution, it will not necessarily be helpful either. For example, invoking a_1`.foo(`a_2`)` from different threads cannot help because the threads do not acquire the locks in opposite order. For the deadlock to manifest, it is essential that different threads invoke `foo` as explained in the previous paragraph. Unfortunately, even for such a simple example, the sophisticated machinery of deadlock detectors fail to detect any problems, unless a suitable test case exists.

In general, deadlocks can occur if a combination of certain methods are invoked by different threads. A brute force analysis of concurrent execution of different possible combination of methods [3] is impractical. Even assuming that the relevant combination of methods to be executed concurrently is provided by an oracle, the *invocation context* becomes vital to detect any issues.

2. Architecture

We address the problem of synthesizing multithreaded test cases to enable deadlock detection in multithreaded libraries [4]. Our key insight is that a subset of properties (e.g., nested lock acquisitions) that are exhibited when a deadlock manifests in a multithreaded execution can also be observed in a single threaded execution. Subsequently, we use the observed properties to enable the synthesis of a deadlock revealing multithreaded test case. Based on this insight, we propose a novel, directed and scalable approach for synthesizing multithreaded test cases. We have implemented a tool, named OMEN, on top of the `soot` [6] bytecode analysis framework that incorporates our approach.

[*] An extended version is published at OOPSLA 2014 [4].

[†] 3rd year PhD student and the lead developer of OMEN.

[‡] Assistant Professor. Formerly, a member of the core analysis team at Coverity, San Francisco, USA.

SPLASH '14, October 20–24, 2014, Portland, OR, USA.
Copyright is held by the owner/author(s).
ACM 978-1-4503-3208-8/14/10.
http://dx.doi.org/10.1145/2660252.2664663

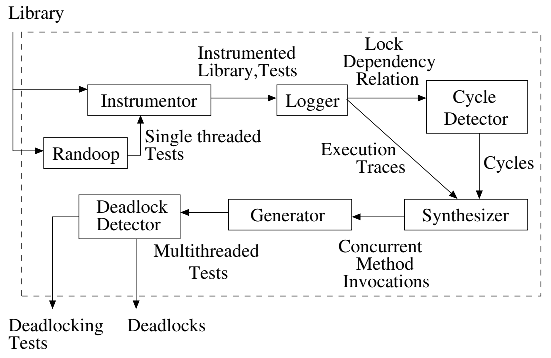

Figure 2. Architecture of `OMEN`.

The overall architecture of our tool, `OMEN` [4], for synthesizing multithreaded test cases to enable deadlock detection is given in Figure 2. There are four major components in our design: `Logger`, `Cycle Detector`, `Synthesizer` and `Generator`. The input to `OMEN` is the library under consideration. If the seed testsuite (manually developed single threaded tests) is not given as input, we generate the seed testsuite using `Randoop` [2]. The `Instrumentor` instruments the library and the tests. The `Logger` executes the instrumented tests and stores the execution traces. It also constructs a *lock dependency relation* across the execution of all test cases in the testsuite and inputs it to the `Cycle Detector`. The `Cycle Detector` detects the presence of cyclic chains in the dependency relation. A cycle suggests the possibility of a deadlock when the corresponding test cases are executed concurrently. However, executing the identified test cases concurrently is not enough as the threads need to acquire locks on *shared* objects in a conflicting order. The `Synthesizer` processes the detected cycles and the execution traces to synthesize possible sets of concurrent method invocations. These invocations when made by different threads may manifest a deadlock. For each set of invocations, the `Generator` constructs a multithreaded test case by spawning threads and performing each invocation in the set from a different thread. These tests are executed and analyzed by `iGoodLock` [1] which reports the detected deadlocks along with the corresponding multithreaded tests.

3. Evaluation

We analyze multithreaded `Java` libraries to evaluate `OMEN`. All the experiments are conducted on an `Ubuntu-12.04` desktop running on a 3.5 Ghz Intel Core i7 processor with 16GB RAM. We are able to detect a number of unknown (and known) deadlocks by applying `OMEN` on many multithreaded `Java` libraries. We use the automatically generated tests from `Randoop` [2] as the seed testsuite and are able to generate 26 multithreaded tests from a total of 3500 sequential tests. Table 1 presents the benchmarks along with information on the number of tests synthesized for each

Class name	Tests	DL	TP
DynamicBin1D	6	36	21
CharArrayWriter	1	1	1
ClosableByteArrayOutputStream	1	1	1
ClosableCharArrayWriter	1	1	1
HashTable	15	20	19
Stack	1	1	1
ByteArrayOutputStream	1	1	1
Total	26	61	45

Table 1. Experimental results. DL: Deadlocks, TP: True Positives.

benchmark. Analyzing the execution traces of the synthesized tests detects 61 deadlocks across all libraries, including 45 true positives. In comparison, ConTeGe [3] randomly invokes methods concurrently and generates approximately 27K multithreaded tests and is unable to detect *any* deadlock. The difference in the numbers shows the contrast between randomized and directed approaches. More interestingly, we also detected the possibility of deadlocks in classes in `colt`, a library for high performance scientific computing, that are *documented* as thread safe. The overall analysis time of `OMEN` is negligible. For example, the analysis time for a trace with one million elements (`DynamicBin1D`) is seven minutes approximately.

4. Related Work

In `ConTeGe` [3], the authors describe a design for randomly generating method invocations that can be executed concurrently. Subsequently, if a concurrent execution results in an exception and none of the corresponding linearized executions fail, then a thread safety violation is reported. As we discuss in Section 3, their approach invokes methods randomly and the search space can be significant. Many dynamic analysis approaches [1, 5, 7] are designed for detecting deadlocks. All these approaches are fundamentally dependent on the quality of the analyzed executions to efficiently detect deadlocks. This is dependent on the quality of the tests. Our approach for automatically synthesizing multithreaded test cases *complements* these techniques.

References

[1] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. PLDI ’09.

[2] C. Pacheco and M. D. Ernst. Randoop: Feedback-directed random testing for java. OOPSLA ’07.

[3] M. Pradel and T. R. Gross. Fully automatic and precise detection of thread safety violations. PLDI ’12.

[4] M. Samak and M. K. Ramanathan. Multithreaded test synthesis for deadlock detection. OOPSLA ’14.

[5] M. Samak and M. K. Ramanathan. Trace driven dynamic deadlock detection and reproduction. PPoPP ’14.

[6] R. Vallee-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? CC ’00.

[7] C. Yan, W. Shangru, and W. K. Chan. Conlock: A constraint-based approach to dynamic checking on deadlocks in multithreaded programs. ICSE ’14.