Omen+: A Precise Dynamic Deadlock Detector for Multithreaded Java Libraries

Malavika Samak Murali Krishna Ramanathan
Indian Institute of Science, Bangalore
{malavika, muralikrishna}@csa.iisc.ernet.in

ABSTRACT

Designing *thread-safe* libraries without concurrency defects can be a challenging task. Detecting deadlocks while invoking methods in these libraries concurrently is hard due to the possible number of method invocation combinations, the object assignments to the parameters and the associated thread interleavings. In this paper, we describe the design and implementation of OMEN+ that takes a multithreaded library as the input and detects *true* deadlocks in a scalable manner. We achieve this by automatically synthesizing *relevant* multithreaded tests and analyze the associated execution traces using a precise deadlock detector. We validate the usefulness of OMEN+ by applying it on many multithreaded Java libraries and detect a number of deadlocks even in documented thread-safe libraries. The tool is available for free download at http://www.csa.iisc.ernet.in/~sss/tools/omenplus.html.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Metrics—*Testing and Debugging,* testing tools; D.2.4 [**Software Engineering**]: Software/Program Verification

Keywords

deadlock detection, dynamic analysis, concurrency

1. INTRODUCTION

Thread-safe [4] libraries are desirable because they abstract the nuances of multithreading in their implementation from their users. However, designing such libraries that are free of concurrency defects is a challenging task. Traditional testing techniques are impractical because of the number of possible concurrent method invocations and the parameters to these invocations. Hence, employing randomized testing [4] is not scalable. The problem is further exacerbated by the large number of possible thread schedules for a given combination of concurrent method invocations and parameters. While the problem of thread schedules is partially addressed by dynamic analyses [6, 2] that abstract the states resulting from different schedules, the ability of these analyses to detect defects and the resulting precision is dictated by the quality of the analyzed tests. Unfortunately, designing *good* multithreaded tests is non-trivial.

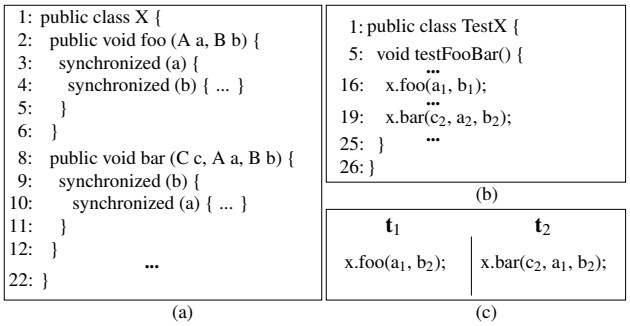

Figure 1: (a) Library under test. (b) Optional sequential (seed) testsuite. (c) Method invocations by two threads that can result in a deadlock.

Figure 1 presents an illustrative example. If the implementation of library X, as shown in Figure 1(a), is to be declared thread-safe, multiple threads should be able to *safely* invoke the methods in class X concurrently without performing any additional synchronization. A tool that enables a developer to detect concurrency defects in her implementation of a thread-safe library can be valuable. More specifically, the tool should be able to take just the implementation of a library as input and automatically detect possible scenarios leading to concurrency defects. *Optionally*, a set of sequential test cases can also be given as input to the tool. This is because sequential tests are usually available to test the various features of the library implementation. For example, the test shown in Figure 1(b) that tests foo and bar can be considered as an optional input to the tool.

In this paper, we describe the implementation of such a tool for detecting deadlocks. Our tool, named OMEN+, takes as input a Java multithreaded library implementation and automatically detects possible deadlocks. Our implementation of OMEN+ integrates our *recent* techniques for deadlock test synthesis [5] and detection [6]. Initially, we synthesize multithreaded tests for deadlock detection [5]. Subsequently, we use WOLF [6] to automatically detect *true* deadlocks by analyzing the traces obtained by executing the synthesized tests. Applying OMEN+ on the example given in Figure 1(a) detects the presence of a deadlock. It synthesizes one multithreaded test where the methods foo and bar are invoked by two threads with appropriate parameters as shown in Figure 1(c) and reports the possible deadlock.

FSE'14, November 16–21, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3056-5/14/11...$15.00
http://dx.doi.org/10.1145/2635868.2661670

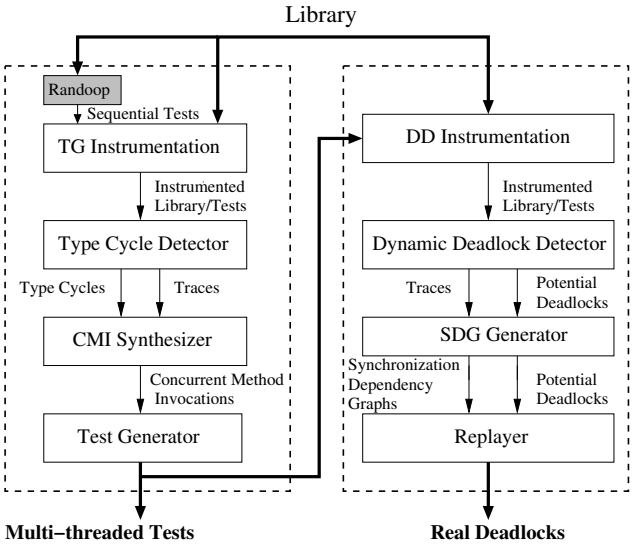

Figure 2: Architecture of OMEN+

In the rest of the paper, we provide a high level architecture overview and the implementation details of OMEN+. We refer the reader to [6, 5] for more details.

2. ARCHITECTURE

The overall architecture of OMEN+ is given in Figure 2. We divide the functioning of OMEN+ into two phases – (a) test synthesis and (b) deadlock detection. We briefly describe the various components in each phase below.

2.1 Test Synthesis

The input to this phase is the implementation of the library and optionally a set of sequential tests. The output of this phase is a set of multithreaded tests that can potentially expose deadlocks. In the absence of sequential tests, OMEN+ uses RANDOOP [3] to generate the sequential tests and uses it for further processing. Henceforth, we refer to the set of sequential tests as the seed testsuite. We now describe the various components involved that help achieve the synthesis of multithreaded tests.

Test Generation (TG) Instrumentation: The classes in the library and the seed testsuite are instrumented to track the synchronization operations (lock acquires and releases), method entries, exits and the loads and stores of variables.

Type Cycle Detector: The test cases are executed and the trace associated with the execution is captured. The execution trace comprises of all the aforementioned instrumented events. Simultaneously, a *lock type* dependency relation representing the lock acquisitions on objects of various types is maintained. A node in the relation is uniquely identified by a tuple that includes the type of the lock object, the source location where the lock is acquired and the context representing the locks currently held by the thread. For example, on executing `testFooBar` in Figure 1(b), the following nodes are created: η_1 = (A, 3, {}), η_2 = (B, 4, η_1), η_3 = (B, 9, {}) and η_4 = (A, 10, η_3).

Subsequently, the detector detects cycles in the relation iteratively. In each iteration, cycles of length one higher than that of the previous iteration are detected. The maximum length of the cycles that needs to be detected is a tunable parameter and is set by the user. The default value is two. For the example relation, the cycle is detected as (η_2, η_4) because the type of η_4 and the type of the held lock in η_2 is A, and the type of η_2 and the type of the held lock in η_4 is B. For a better intuition underlying the detection of a type cycle, we refer the reader to [5].

Concurrent Method Invocation (CMI) Synthesizer: The goal of this component is to synthesize the combination of methods that need to be invoked concurrently, such that the type cycles detected in the previous step can manifest. Additionally, for the cycles to lead to deadlocks, a set of constraints on the parameters are necessary to constraint the lock acquisition on specific object instances. Therefore, for a detected type cycle, the component analyzes the execution trace to identify the method invocations (CMI) that cause the lock acquisitions involved in the cycle. Moreover, it performs a backward flow analysis to identify the parameters to the method invocations that influence the lock acquisition. Subsequently, it generates constraints on the identified parameters of various methods so that the deadlock can manifest in a real execution.

Based on the cycle (η_2, η_4) detected in the previous step, the lock acquisitions at lines 3, 4, 9 and 10 are interesting. By analyzing the execution trace, we obtain (`foo`, `bar`) as the CMI associated with the cycle. The parameters relevant to the method invocations in the CMI are the first and second parameters for method `foo` and second and third parameters for method `bar`. For a deadlock to manifest, the locks should be acquired on the same object instance. Therefore, the generated constraints are that the first parameter of `foo` and second parameter of `bar` must share the same object instance, and the second parameter of `foo` and third parameter of `bar` must share the same object instance.

Test Generator: The generator uses the output from the previous step to synthesize a set of multithreaded tests. For each identified CMI, associated parameters and constraints, it generates a test case that spawns distinct threads where each thread invokes a method in the CMI. The parameters to the methods in the generated test case are provided as follows: (a) the tests corresponding to the methods in the CMI in the seed testsuite are executed and the objects that form the parameters to the methods in the execution are collected and (b) a mapping from the collected objects to the parameters in the method invocation in the synthesized test case is performed such that the constraints are satisfied. More details on the synthesized test is given in Section 3.

2.2 Deadlock Detection

In this phase, the synthesized multithreaded tests are executed and the execution trace is analyzed to detect a set of *true* deadlocks. The deadlock detection is done by finding cycles in the *traditional* object sensitive lock graph. The correctness of the detected deadlocks is ensured by reproducing the deadlock after manipulating the schedule appropriately.

Deadlock Detection (DD) Instrumentation: The library class files and the synthesized multithreaded tests are instrumented to track the lock acquisitions and releases.

Dynamic Deadlock Detector: The detector executes the instrumented test cases and constructs a lock graph. The nodes in the lock graph are uniquely identified by the following attributes – (a) the thread that acquired the lock, (b) the object instance on which the lock was acquired, (c) the dynamic execution index of the instruction where the lock was acquired and (d) the context specifying the set of locks held by the thread. An edge exists between two nodes representing lock acquisitions l_1 and l_2, if l_2 is acquired while holding l_1. Cycle detection on the lock graph outputs a set of potential deadlocks.

The cycle detected by analyzing the execution trace of the synthesized multithreaded test (shown in Figure 1(c)) consists of two nodes representing the object instances associated with a_1 and b_2.

```
 1: public class TestDriver {
 2:   public static void cycle() {
 3:     Initializer.initialize(0);
 4:      Thread thread0 = new Thread() {
 5:        public void run() {
 6:          List<Parameter> paramList = Initializer.collectedObjects.get(0);
 7:          // Initializes objects from paramList
 8:          X receiver = ...;  A par1 = ...; B par2 = ...;
 9:          /* par1: class TestX, line 16, parameter1 to method foo, run1
10:             par2: class TestX, line 19, parameter3 to method bar, run2
11:             foo : TestX, line 16,  run1
12:           */
13:          receiver.foo(par1, par2);
14:        }
15:     };
16:     Thread thread1 = new Thread() {
17:       public void run() {
18:         List<Parameter> paramList = Initializer.collectedObjects.get(1);
19:         // Intializes objects from paramList
20:         X receiver = ...; C par1 = ...; A par2 = ...; B par3 = ...;
21:         /* par3: class TestX, line 19, parameter3 to method bar, run2
22:            par2: class TestX, line 16, parameter1 to method foo, run1
23:            bar : TestX, line 19, run2
24:         */
25:         receiver.bar(par1, par2, par3);
26:       }
27:     };
                  ...
30:   }
31: }
```

(a)

```
443:  public static void initialize(int cycle) {
                    ...
447:    for(int i = 0; i < nodeCount(cycle); i++) {
                    ...
454:      //Collect the objects by executing test suite
455:      initiateCollection(i, cycle);
456:    }
        // Enforce constraints on the collected objects
466:    enforceConstraints(cycle);
                  ...
475:  }
```

(b)

```
Iteration 1 (run1) :
  Testcase invoked   : testFooBar
  Line number        : 16
  Target invocation  : foo
Iteration 2 (run2) :
  Testcase invoked   : testFooBar
  Line number        : 19
  Target invocation  : bar
```

(c)

```
Deadlock 1
    Thread:  (2)
      Lock 1000 at X.java, line 3 (iid is 131) –>
      Lock 1006 at X.java, line 4 (iid is 132)
    Thread:  (3)
      Lock 1006 at X.java, line 9 (iid is 137) –>
      Lock 1000 at X.java, line 10 (iid is 138)
```

(d)

Figure 3: (a) Test synthesized by OMEN+ for the example given in Figure 1(a). (b) Implementation skeleton of `initialize` to collect object instances. (c) Data used by `initialize` to execute appropriate sequential tests. (d) Deadlock detected by OMEN+.

SDG Generator: This component builds the partial order to be followed by the execution to reproduce the required input deadlock. The schedule is built by leveraging the execution trace. The output is a synchronization dependency graph (SDG) which constrains the order of lock acquisitions by different threads.

Replayer: For each detected deadlock, the `Replayer` replays the execution so that the constraints in the SDG are satisfied. If the execution results in the required deadlock being reproduced, then the deadlock is output to the end user.

3. IMPLEMENTATION

We implement OMEN+ in `Java` and use `soot` [7] to instrument the bytecode to track the relevant operations for our analysis. To analyze a library with OMEN+, it is invoked as follows:

```
java -cp <classpath> omen.util.OmenDriver
 -randoop-arguments --testclass=<class-under-test>
 <other-randoop-args>
```

One of the interesting features of the tool is the ability to use sequential tests to synthesize multithreaded tests where the object instances obtained from running a sequential test are used in the synthesized test. We now explain the various parts of a synthesized test.

The multithreaded test case synthesized in the first phase for the example in Figure 1(a) is given in Figure 3(a). The number of threads spawned is equivalent to the number of methods in the CMI. As the CMI has (`foo`, `bar`), threads `thread0` and `thread1` are created as shown in the figure and the threads invoke the methods `foo` and `bar` at lines 14 and 25 respectively. The testcase obtains the objects to be passed as parameters to these methods by invoking method `initialize` at line 3.

The implementation skeleton of `initialize` that is part of OMEN+ is shown in Figure 3(b). This does not change for different synthesized tests but the execution changes based on the cycle input to it. It invokes `initiateCollection` within a `for` loop at line `455` where the number of iterations of the loop is equal to the number of nodes in the cycle. `initiateCollection` invokes a sequential testcase and collects (and stores) the object instances passed to the parameters of the appropriate method invocation in the test case.

For the given example, the cycle is (η_2, η_4). Therefore, the method `initiateCollection` is invoked twice. The data pertaining to the test that needs to be run and the objects that need to be collected for each cycle is maintained as an output of `CMI Synthesizer`. For the current example, this data is presented in Figure 3(c). Hence, the first invocation of `initiateCollection` collects parameters to method `foo` at line `16` in `testFooBar` and the second invocation collects parameters to method `bar` at line `19` in `testFooBar`. After the objects are collected, the constraints that are generated for the cycle is enforced by the method `enforceConstraints` at line

Benchmark	Version	$\|D\|$	$\|\Sigma\|$	TGT(s)	$\|\Theta\|$	|CMI|	Tests*	DDT(s)	DL*
DynamicBin1D	colt-1.2.0	64	1025K	368	64	36	6	279	21
CharArrayWriter	classpath-0.98	12	107K	31	1	1	1	6	1
ClosableByteArrayOutputStream	hsqldb-2.3.2	25	188K	27	1	1	1	5	1
ClosableCharArrayWriter	hsqldb-2.3.2	25	158K	22	1	1	1	4	1
HashTable	jdk1.7	28	553K	85	64	28	28	147	35
Stack	jdk1.7	46	219K	42	4	3	3	26	5
ByteArrayOutputStream	jdk1.7	15	146K	22	1	1	1	4	1

Table 1: Experimental results of analyzing the benchmarks with OMEN+. $|D|$: number of nodes in the lock type dependency relation, Σ: length of the execution trace, TGT: Test Generation time, Θ: Detected cycles, CMI: Concurrent Method Invocations, DDT: Deadlock Detection time, DL: Number of deadlocks. *Output of OMEN+.

466 (see Figure 3(b)) by reassigning the collected object instances appropriately. When `initialize` returns, the parameters to the various method invocations are setup so as to expose a potential deadlock. Each thread just gets the collection of object instances for the method that it needs to invoke and invokes the method accordingly.

If the synthesized test case is executed and analyzed, true deadlocks in the library are detected. For the example under consideration, one deadlock is detected as shown in Figure 3(d). A deadlock is detected between thread identifiers 2 and 3 corresponding to `thread0` and `thread1` respectively. In `X.java` (shown in Figure 1(a)), thread 2 attempts to acquire a lock on object instance `1006` at line 4 while holding a lock on `1000` that is acquired at line 3. On the other hand, thread 3 attempts to acquire a lock on `1000` at line 10 while holding a lock on `1006` that is acquired at line 9.

4. EXPERIMENTS

We analyzed various `Java` multithreaded libraries using `OMEN+` on a Ubuntu-12.04 desktop running on a 3.5 Ghz Intel Core i7 processor with 16GB RAM. The input to `OMEN+` is the implementation of the libraries and the initial sequential seed testsuite that is generated using `RANDOOP` [3]. The analyzed libraries and the results obtained by setting the cycle length to two are given in Table 1. Increasing the cycle length beyond two did not expose any additional defects on these benchmarks. The overall time for synthesizing the multithreaded tests is negligible. For example, the time taken for synthesizing the tests for `colt` which generates the longest trace is approximately 6 minutes. The synthesis generates a number of multithreaded tests across different benchmarks ranging from 1 to 28. Executing the synthesized tests and performing deadlock detection outputs a number of real deadlocks ranging from 1 to 35. Each deadlock is accompanied with a trace corresponding to a deadlocking execution. In [5], the multithreaded tests are synthesized. However, the reported deadlocks are manually analyzed to identify whether the reported deadlocks are true positives. In contrast, the integration of a precise deadlock detector [6] as part of implementation eliminates the need for a developer to analyze the source code and reason about the correctness of the reported deadlock.

5. RELATED WORK

`ConTeGe` [4] automatically detects thread safety violations by randomly invoking methods concurrently. This approach is not scalable because of the magnitude of the search space and it did not find *any* issues in the analyzed benchmarks [5]. `RANDOOP` [3] automatically synthesizes sequential tests and we leverage it to build the initial seed testsuite. Static analysis approaches [8] for deadlock detection can be imprecise and also cannot automatically provide proof of correctness of the detected deadlock. We showed in [6] that our approach of deadlock detection, that is employed here, outperforms other dynamic deadlock detectors [2, 1] with respect to precision and scalability.

6. CONCLUSIONS

We present the design and implementation of `OMEN+`, a dynamic analysis tool for detecting deadlocks in multithreaded `Java` libraries. The input to the tool is the implementation of the library that needs to be tested and the output is a set of *true* deadlocks. We analyze a number of existing `Java` libraries and detect potential deadlocking scenarios even in thread-safe libraries.

7. REFERENCES

[1] Y. Cai and W. K. Chan. Magicfuzzer: Scalable deadlock detection for large-scale applications. ICSE 2012.

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

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

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

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

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

[7] R. Vallee-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In *CC*, 2000.

[8] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for java libraries. In *ECOOP'05*.