Multithreaded Test Synthesis for Deadlock Detection

Malavika Samak and Murali Krishna Ramanathan

Indian Institute of Science, Bangalore

{malavika,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. The usual practice involves employing software testing and/or dynamic analysis to detect deadlocks. Their effectiveness is dependent on *well-designed* multithreaded test cases. Unsurprisingly, developing multithreaded tests is significantly harder than developing sequential tests for obvious reasons.

In this paper, we address the problem of automatically synthesizing multithreaded tests that can induce deadlocks. The key insight to our approach is that a subset of the properties observed when a deadlock manifests in a concurrent execution can also be observed in a single threaded execution. We design a novel, automatic, scalable and *directed* approach that identifies these properties and synthesizes a deadlock revealing multithreaded test. The input to our approach is the library implementation under consideration and the output is a set of deadlock revealing multithreaded tests.

We have implemented our approach as part of a tool, named `OMEN`[1]. `OMEN` is able to synthesize multithreaded tests on many multithreaded `Java` libraries. Applying a dynamic deadlock detector on the execution of the synthesized tests results in the detection of a number of deadlocks, including 35 real deadlocks in classes *documented* as thread-safe. Moreover, our experimental results show that dynamic analysis on multithreaded tests that are either synthesized randomly or developed by third-party programmers are ineffective in detecting the deadlocks.

Keywords Deadlock detection; dynamic analysis; concurrency

[1] The tool derives its name due to its ability to predict ominous deadlocks.

OOPSLA '14, October 20–24, 2014, Portland, OR, USA.

http://dx.doi.org/10.1145/2660193.2660238

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

1. Introduction

Thread-safe [22] 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. A library implementation is considered thread-safe, if any *valid* concurrent invocation of its methods is free of concurrency bugs without requiring additional synchronization. To provide this guarantee, the library implementation needs to be tested for possible concurrency bugs, including deadlocks. There are multiple ways of detecting deadlocks including use of software testing [6, 11, 16, 17], dynamic analysis [3, 14, 23] and static analysis [5, 15, 18, 27] approaches.

Traditional software testing techniques are inadequate to detect deadlocks because the defects may manifest only on rare thread schedules [29]. As a result, a number of dynamic analysis approaches are designed to detect and reproduce deadlocks [3, 14, 23, 28]. The dynamic analyses operate by analyzing the execution of the program on a given test case and attempt to infer whether *any* other thread interleaving on the same test case can result in a deadlock. Unsurprisingly, the ability of the dynamic analysis techniques to detect deadlocks is *critically* dependent on the quality of the analyzed executions. Transitively, the quality of the test cases becomes the key to detecting deadlocks. But, in practice, even writing test cases for sequential programs is considered arduous and ineffective resulting in the design of automatic test generators [9, 20, 25].

Pradel and Gross [22] designed an interesting approach that *randomly* generates method sequences which are then executed concurrently by different threads to automatically detect thread safety violations. While an important first step, the use of randomization as a substrate can result in the generation of many *uninteresting* (defect unrevealing) test cases. To detect a concurrency bug, the execution needs to satisfy certain properties. The properties include the code that is covered by individual threads, objects operated on by different threads, the lifetimes of the threads, etc. These

properties can play a significant role in defect detection. Any approach that does not consider these factors into account while generating test cases will likely suffer from the usual limitations associated with the size of the search space.

To illustrate this problem further, 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.

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

Figure 1. Illustrative example.

If `testFoo` is executed by a single thread, a dynamic deadlock detector 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 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.

In this paper, we address the problem of synthesizing multithreaded test cases to enable deadlock detection in multithreaded libraries. 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. The input to our approach is the library that needs to be tested. The output is a set of multithreaded tests, along with a list of deadlocks detected by each test.

Our approach works by initially generating a seed testsuite, `I`, using `Randoop` [20]. Optionally, if a manually developed testsuite already exists for the library, then it can be used as the seed testsuite[2]. We analyze the traces obtained by executing the tests in `I` and construct a *lock dependency relation*, D, which is a collection of *lock nodes*. Because the traces are from different executions, the lock instances observed in the traces for different tests are different. Therefore, we use the type of the lock instance to represent a lock node in D so that the partial identity of locks across runs can be maintained. A cycle in D points to a potential deadlocking scenario. We identify the method invocations, `M`, that are responsible for the creation of the lock nodes present in the cycle. These methods need to be invoked concurrently across multiple threads.

Executing the methods concurrently alone may not be sufficient because objects used in the invocations need to be shared or linked properly. Therefore, we derive an *invocation context*, i.e., the parameters (and receivers) to be used while invoking the methods in `M`. We analyze the execution trace associated with the test case involving each method in `M`. From the location of the lock acquisition in the trace, we search backwards to identify the data dependence of the lock on the method's parameter. We utilize this information along with the lock nodes in a cycle, to generate an assignment of objects to parameters of the methods in `M`. Subsequently, we generate a test that spawns multiple threads where each thread will invoke a method from `M` with the appropriate invocation context. We use `iGoodLock` [14] to analyze the execution of the multithreaded test to detect deadlocks.

We have implemented a tool, named `OMEN`, on top of the `soot` [26] bytecode analysis framework that incorporates our approach. 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` [20] as the seed testsuite and are able to generate 26 multithreaded tests from a total of 3500 sequential tests. Analyzing the execution traces of the synthesized tests detects 61 deadlocks across all libraries, including 45 true positives. In comparison, ConTeGe [22] 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` [1], 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 is seven minutes approximately.

[2] These tests may cover typical usage scenarios of the library and direct our approach towards generating multithreaded tests for these scenarios.

To study the difficulty associated with detecting the deadlocks that are detected by OMEN, we obtained manually written tests from four volunteers including two graduate students (unaffiliated to our lab), a researcher and a software engineer. We observe that application of iGoodLock [14] on the execution of their multithreaded tests did not reveal *any* deadlock. However, we were able to use their sequential tests as seed testsuite to our approach and detected 63 potential deadlocks (42 true positives). We also observed that the detection ability of our approach is influenced by the basic coverage provided by the tests. Therefore, the first author developed a sequential seed testsuite with the objective of invoking distinct methods to provide good coverage. Significantly, even using such a naive seed testsuite, our tool detects 81 deadlocks (60 true positives). In other words, OMEN is able to leverage even uninteresting sequential tests to synthesize deadlock revealing multithreaded tests.

We make the following technical contributions:

1. We propose an elegant approach to automatically synthesize multithreaded tests that can enable deadlock detection in multithreaded libraries.
2. We address the challenges associated with identifying the combination of the methods that need to be invoked concurrently and the invocation context before invoking the methods.
3. We provide a detailed design and implementation of our tool, named OMEN, that incorporates our proposed algorithms.
4. We demonstrate the usefulness of OMEN by applying it on many multithreaded Java libraries and detect a number of previously unknown (and known) deadlocks.

2. Motivation

We motivate the problem addressed in this paper by providing an example from a widely used multithreaded library, colt [1]. The documentation for colt suggests that some classes defined in it are thread safe. The example demonstrates the various challenges in synthesizing effective multithreaded tests that can help detect violations of this property.

The implementation of method sampleBootstrap that is defined in class DynamicBin1D is given in Figure 2. The method is used for statistical computations. A software tester unfamiliar with the domain may find it daunting to write effective sequential tests to validate the various features. Furthermore, the documentation for class DynamicBin1D suggests that it is thread-safe [1]. Helpfully, the documentation also provides a sample usage of the method and we present the relevant parts of the usage in Figure 3. The code fragment constructs two instances of DynamicBin1D, X and Y, and also creates other relevant objects that are used while invoking sampleBootstrap on instance X at line 25. Because the code corresponding to the sample usage is single threaded, a

```
581 : public synchronized DynamicBin1D sampleBootstrap
          ( DynamicBin1D other, ... ) {
582 :   if (randomGenerator==null)
          randomGenerator = Uniform.makeDefaultGenerator();
585 :   int maxCapacity = 1000;
586 :   int s1 = size();
587 :   int s2 = other.size();

590 :   DynamicBin1D sample1 = new DynamicBin1D();
591 :   DoubleBuffer buffer1 = sample1.buffered (...);

593 :   DynamicBin1D sample2 = new DynamicBin1D();
594 :   DoubleBuffer buffer2 = sample2.buffered (...);

596 :   DynamicBin1D bootstrap = new DynamicBin1D();
597 :   DoubleBuffer bootBuffer = bootstrap.buffered (...);

600 :   for (int i=resamples; --i >= 0; ) {
601 :     sample1.clear();
602 :     sample2.clear();

604 :     this.sample (s1, true, randomGenerator, buffer1);
605 :     other.sample (s2, true, randomGenerator, buffer2);

607 :     bootBuffer.add(function.apply(sample1, sample2));
608 :   }

611 : }
```

Figure 2. Implementation of sampleBootstrap from colt.

dynamic deadlock detector will not report any defects. However, we will illustrate a few scenarios involving the invocation of sampleBootstrap across multiple threads that will result in a potential deadlock.

```
Example usage:
 2 : double[] v1 = { 1, 2, 3, ...};
 3 : double[] v2 = { 10, 11, 12, ...};
 4 : DynamicBin1D X = new DynamicBin1D();
 5 : DynamicBin1D Y = new DynamicBin1D();
 6 : X.addAllOf(new DoubleArrayList(v1));
 7 : Y.addAllOf(new DoubleArrayList(v2));
 8 : RandomEngine random = new MarsenneTwister ();
11 : BinBinFunction1D diff = new BinBinFunction1D { ... }
25 : X.sampleBootstrap (Y, 1000, random, diff);
```

Figure 3. Sample usage of sampleBootstrap.

Since sampleBootstrap is synchronized (at line 581), a lock on the object instance invoking it is acquired before executing any other instruction within the method body. This lock is released only when the method exits. Between the method entry and exit, a number of locks are acquired and released. For example, the invocations of the methods size (at lines 586 and 587), buffered (at lines 591, 594 and 597), clear (at lines 601 and 602) and sample (at lines

604 and 605) acquire and release locks on their receivers respectively while holding a lock on the receiver of the current invocation of `sampleBootstrap`. The code for the implementation of these methods that cause the nested acquisition spans multiple classes and files. Clearly, manually parsing the implementation and writing test cases to detect deadlocks becomes impractical.

Moreover, all the nested acquisitions are not equally interesting from the perspective of deadlock detection. For example, the nested acquisition associated with the *reentrant* locks at lines 586 and 604 can never cause a deadlock. Similarly, the nested acquisitions at lines 591, 594, 597, 601 and 602 are also safe because the associated objects on which the lock is acquired (`sample1`, `sample2` and `bootstrap` respectively) are created locally. The caller of `sampleBootstrap` does not have access to these objects when the nested lock acquisitions happen. This leaves us with nested acquisitions at lines 587 and 605 that can create some interesting deadlocking scenarios given a suitable invocation context.

In summary, the following barriers exist to detect deadlocking scenarios involving concurrent invocation of *one* method:

- finding possible nested acquisitions spread across multiple methods and files,
- identifying relevant acquisitions that can expose deadlocking scenarios, and
- synthesizing a new test case that invokes the method with appropriate parameters from different threads.

`DynamicBin1D` contains 35 public methods. Therefore, if the deadlocking scenarios involve invoking multiple methods in `DynamicBin1D`, then an additional barrier exists in the form of identifying the relevant methods that need to be invoked among 2^{35} combinations. With parameters to be appropriately set for each of these invocations, the number of tests that need to be run becomes impractical quickly. This is without even accounting for the specific schedules that will result in a deadlock.

Sl.No	Deadlocks
1	{(t_1: 581→ 587), (t_2: 581→ 587)}
2	{(t_1: 581→ 587), (t_2: 581→ 605)}
3	{(t_1: 581→ 605), (t_2: 581→ 587)}
4	{(t_1: 581→ 605), (t_2: 581→ 605)}

Table 1. Detected deadlocks.

In this paper, we overcome all these barriers. The input to `OMEN` includes the implementation of `DynamicBin1D` and optionally a seed testsuite that includes the sample uses of different `API`s of the class. Figure 4 shows one of the automatically synthesized multithreaded tests output by `OMEN`. Here, there are invocations to `sampleBootstrap` from two different threads and the invocation context is setup such that a deadlock can manifest.

```
public class TestDriver   {
  // Initialize variables X, Y, random and diff
  Thread t1 = new Thread() {
        void run() {
              X.sampleBootstrap(Y, 1000, random, diff);
        }
  }
  Thread t2 = new Thread() {
        void run() {
              Y.sampleBootstrap(X, 1000, random, diff);
        }
  }
  // start the threads t1 and t2
}
```

Figure 4. Test case synthesized by `OMEN`.

After analyzing the execution of the synthesized test in Figure 4, `iGoodLock` [14] detects four deadlocks as shown in Table 1. For example, the first deadlock indicates that t_1 acquires a lock on `X` at line 581 and waits for a lock on `Y` at line 587, while t_2 acquires a lock on `Y` at line 581 and waits for a lock on `X` at line 587. A manual analysis of the reported deadlocks results in identifying the first three deadlocks as true positives.

3. Design

The overall architecture of our tool, `OMEN`, for synthesizing multithreaded test cases to enable deadlock detection is given in Figure 5. There are four major components in our design: `Logger`, `Cycle Detector`, `Synthesizer` and `Generator`.

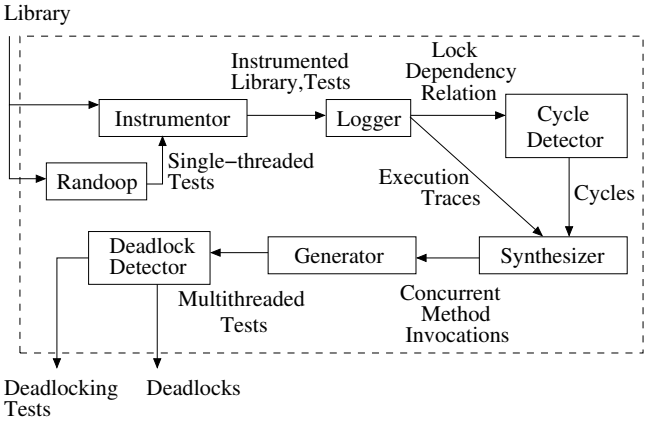

Figure 5. Architecture of `OMEN`.

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` [20]. 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 a dynamic deadlock detector which reports the detected deadlocks along with the corresponding multithreaded tests. We now provide a detailed description of the working of the major components.

3.1 Logger

The primary goal of the `Logger` is to monitor the execution of multiple test cases and track data appropriately so that the collected data can be effectively used by the other components. We define two data structures, an *execution trace* and a *lock dependency relation*, to collect the required data for our analysis and describe them here. The data maintained in the lock dependency relation and the execution trace is used to address two fundamental issues to synthesize a multithreaded test:

- Identify lock acquisitions that point to a potential deadlock.
- Identify the appropriate invocation context for the methods (in the seed testsuite), such that their invocation from different threads will lead to a real deadlock.

We provide a detailed description of the former in Section 3.2 and the latter in Section 3.3.

3.1.1 Execution trace

An execution trace, σ, is a sequence of events generated for every test case execution. The events include:

- `alloc(x)` : Represents an allocation to `x`.
- `load(x,f,y)` : If `f` is `null`, then the event represents the assignment `x := y`. Otherwise, represents `x.f := y`.
- `store(x,f,y)` : Represents the assignment `x := y.f`. Here, `f` is never `null` to avoid redundancy with `load`.
- `lock(x)` : Acquires lock on the object represented by `x`.
- `unlock(x)` : Releases the lock on the object represented by `x`.
- `enter(iid,m,plist)` : Invocation of method `m` at invocation index `iid`. `plist` gives the sequence of parameters to `m`.
- `exit(m)` : Return from method `m`.

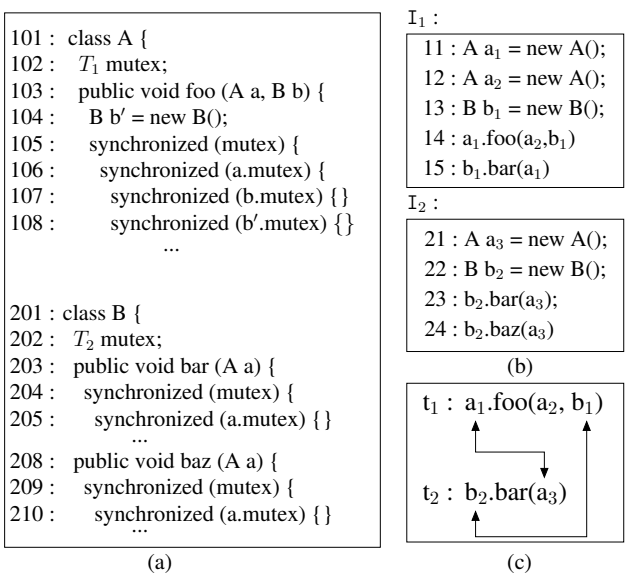

Figure 6. (a) Running example. (b) Seed testsuite: $\{\mathtt{I}_1, \mathtt{I}_2\}$. (c) Synthesizing a potential deadlock.

Here, x and y are variable names, and f is a field name. We maintain the above events to track the methods in which the lock acquisitions are made and the flow of data to the lock acquisitions. The collection of the traces for all tests is represented by Σ.

We use a running example to explain the different stages of our approach. Figure 6 gives the implementation of two classes, `A` and `B`. It also provides the implementation of the seed testsuite, $\mathtt{I}_1$ and $\mathtt{I}_2$, that are used to test the methods in these classes. If the methods in `A` and `B` are invoked concurrently, then there are many possible deadlocks. One of the possible concurrent method invocations leading to a deadlock is shown in Figure 6(c). If methods `foo` and `bar` are invoked from different threads and the objects connected by a line are shared across the threads, then a deadlock may manifest on some schedule. The lock acquisitions that correspond to this deadlock are at lines `105`, `107`, `204` and `205`. Sequential tests $\mathtt{I}_1$ and $\mathtt{I}_2$ given in Figure 6(b) will not be able to detect the possible deadlocks. Given the implementation in Figure 6(a), the goal is to synthesize multithreaded tests to detect possible deadlocks in its usage.

The partial trace associated with the execution of the test cases in Figure 6(b) is given in Table 2. The intermediate temporary variables are represented by $v_1, \ldots, v_6$. The `enter` events show that the methods have one additional element in the `plist` compared to the number of parameters in the original method invocation in the source because we insert the receiver object of the invocation as the first parameter. For example, $\mathtt{e}_4$ has three items in the `plist` whereas `foo` takes two arguments and the first item in `plist` corresponds to $\mathtt{a}_1$.

$\sigma(\mathtt{I}_1)$	$\sigma(\mathtt{I}_2)$
...	...
$\mathtt{e}_1$: load(v_1,null,$\mathtt{a}_1$)	$\mathtt{e}'_1$: load(v_1,null,$\mathtt{b}_2$)
$\mathtt{e}_2$: load(v_2,null,$\mathtt{a}_2$)	$\mathtt{e}'_2$: load(v_2,null,$\mathtt{a}_3$)
$\mathtt{e}_3$: load(v_3,null,$\mathtt{b}_1$)	$\mathtt{e}'_3$: enter($\mathtt{e}'_3$,bar,(v_1,v_2))
$\mathtt{e}_4$: enter($\mathtt{e}_4$,foo,(v_1, v_2, v_3))	$\mathtt{e}'_4$: store(v_3,mutex,v_1)
$\mathtt{e}_5$: alloc(b′)	$\mathtt{e}'_5$: lock(v_3)
$\mathtt{e}_6$: store(v_4,mutex,v_1)	$\mathtt{e}'_6$: store(v_4,mutex,v_2)
$\mathtt{e}_7$: lock(v_4)	$\mathtt{e}'_7$: lock(v_4)
$\mathtt{e}_8$: store(v_5,mutex,v_2)	.
$\mathtt{e}_9$: lock(v_5)	.
$\mathtt{e}_{10}$: store(v_6,mutex,v_3)	.
$\mathtt{e}_{11}$: lock(v_6)	
...	

Table 2. Partial execution trace associated with Figure 6.

3.1.2 Lock dependency relation

In this subsection, we provide the rationale underlying the design of the lock dependency relation before defining it.

Traditional object-sensitive cycle detection cannot be employed to detect problematic regions in the code. This is because we propose to monitor the executions of single threaded tests and the object instance identifiers across test executions do not necessarily match. Therefore, the structure of the relation should be such that it can help decipher the location of possible deadlocks even with data obtained from multiple single-threaded tests. We propose to use type of the locks to help achieve our goal.

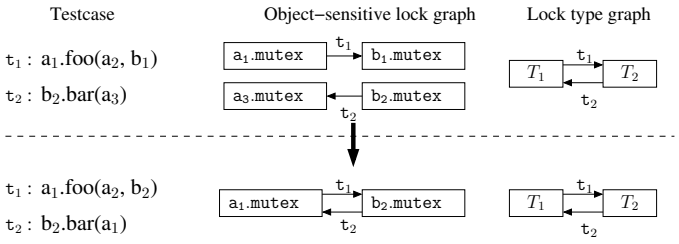

Figure 7. Example illustrating the benefits of using lock types.

Figure 7 illustrates the benefits of using lock types in a dependency relation. For the test case shown above the dotted line, two threads, $\mathtt{t}_1$ and $\mathtt{t}_2$, invoke `foo` and `bar` respectively. If we construct a traditional object-sensitive lock graph, it will not have a cycle because the objects used in the respective invocations are *different*. On the other hand, the type of $\mathtt{a}_1$`.mutex` and $\mathtt{a}_3$`.mutex` is T_1 and the type of $\mathtt{b}_1$`.mutex` and $\mathtt{b}_2$`.mutex` is T_2. Therefore, the lock type graph that maintains the type identifier instead of the object instance identifier has a cycle. We observe that this points to a potential deadlock if the invocation context is setup appropriately. This is confirmed by the example given below the dotted line, where there is a cycle in both the graphs due to the appropriate invocation context.

If the two threads in the test case shown above the dotted line were to be considered individual single threaded tests, we can still obtain a cycle on the lock type graph. Subsequently, we can use the detected cycle to identify the relevant method invocations and the invocation context to synthesize a test as shown by the multithreaded test given below the dotted line.

The lock dependency relation is an elaboration of the lock type graph to help synthesize a deadlock revealing multithreaded test. A lock dependency relation D is a set of lock nodes, where each lock node is an abstraction of a lock acquisition, and is built to facilitate detection of cyclic acquisitions. To facilitate the detection, we define the lock node η = (τ,s,H,TI), where each element of the tuple is described below:

- **Lock type (τ) :** *Represents the data type of the lock object.* Lock type is chosen to represent the lock acquisition as we detect cyclic acquisition on types as discussed previously. This facilitates detection of possible deadlocks across test cases even when there is no object sharing.
- **Source location (s):** *Uniquely identifies the source location where the lock is acquired.* This enables differentiation of two locks of the same type acquired at different source locations as they can correspond to different defects. For example, we want to differentiate between the acquisitions at lines 204 and 209 in Figure 6, where the locks acquired correspond to the same type T_1.
- **Held locks (H):** *Represents the set of locks that are currently held based on the order of acquisitions and is represented as a set of lock nodes.* The entire history of locks that are held, when the current lock is acquired, is encoded into H. This enables detection of lock acquisitions made with distinct contexts. When it is not empty, it represents a nested lock acquisition.
- **Test and trace location identifier (TI):** *Set of ordered pairs (*$\mathtt{I}_i$,index*), where* $\mathtt{I}_i$ *identifies the test case in the testsuite and* index *gives the index of the lock acquisition in the execution trace of* $\mathtt{I}_i$. If $\mathtt{I}_i$ acquires a lock on object with type τ at source location s while holding the locks corresponding to the lock nodes in H, then an ordered pair ($\mathtt{I}_i$,index) will be added to TI. This ensures that redundant tuples representing the *same* locking scenarios from different tests are not added as separate lock nodes to D. Consequently, this helps improve the processing time due to fewer lock nodes in D. However, maintaining a set as opposed to just the first test that created η provides the flexibility for our analysis to choose any one of the test cases from the set for concurrent execution.

For ease of presentation, we will also use $\tau(\eta)$, $s(\eta)$, $\mathtt{H}(\eta)$ and $\mathtt{TI}(\eta)$ to refer to the four elements of tuple η.

We now explain the construction of the lock dependency relation using the illustrative example from Figure 6. Initially D is empty. The invocation of method `foo` at line 14 by $\mathtt{I}_1$ results in multiple nested lock acquisitions. Initially, a lock on $\mathtt{a}_1$`.mutex` is acquired at location 105. Accordingly,

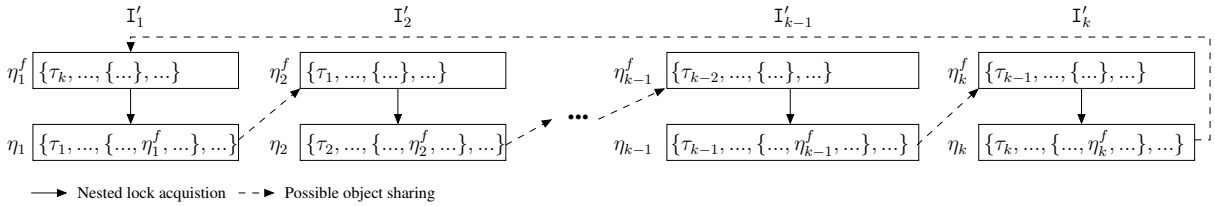

Figure 9. Illustration of cycle $(\eta_1, \eta_2, \ldots, \eta_k)$.

$$\eta_1 = \{T_1, \mathtt{105}, \{\}, \{(\mathtt{I_1}, \mathtt{e_7})\}\}$$
$$\eta_2 = \{T_1, \mathtt{106}, \{\eta_1\}, \{(\mathtt{I_1}, \mathtt{e_9})\}\}$$
$$\eta_3 = \{T_2, \mathtt{107}, \{\eta_1, \eta_2\}, \{(\mathtt{I_1}, \mathtt{e_{11}})\}\}$$
$$\eta_4 = \{T_2, \mathtt{108}, \{\eta_1, \eta_2\}, \{(\mathtt{I_1}, \mathtt{e_{13}})\}\}$$
$$\eta_5 = \{T_2, \mathtt{204}, \{\}, \{(\mathtt{I_1}, \mathtt{e_{20}}), (\mathtt{I_2}, \mathtt{e'_5})\}\}$$
$$\eta_6 = \{T_1, \mathtt{205}, \{\eta_5\}, \{(\mathtt{I_1}, \mathtt{e_{22}}), (\mathtt{I_2}, \mathtt{e'_7})\}\}$$
$$\eta_7 = \{T_2, \mathtt{209}, \{\}, \{(\mathtt{I_2}, \mathtt{e'_{14}})\}\}$$
$$\eta_8 = \{T_1, \mathtt{210}, \{\eta_7\}, \{(\mathtt{I_2}, \mathtt{e'_{16}})\}\}$$

Figure 8. D after executing $\mathtt{I}_1$ and $\mathtt{I}_2$.

a new lock node, η_1 = $\{T_1$,105,$\{\}$,$\{(\mathtt{I_1}$,$\mathtt{e_7})\}\}$, is added to D. The lock node corresponds to index $\mathtt{e}_7$ of the trace (shown in Table 2) associated with $\mathtt{I}_1$ and hence ($\mathtt{I_1}$,$\mathtt{e_7}$) is added. Subsequently, when a lock is acquired on `mutex` fields of parameters `a` and `b`, tuple η_2 = $\{T_1$, 106 , $\{\eta_1\}$,$\{(\mathtt{I_1}$,$\mathtt{e_9})\}\}$ and η_3 = $\{T_2$, 107 , $\{\eta_1$, $\eta_2\}$, $\{(\mathtt{I_1}$, $\mathtt{e_{11}})\}\}$ are added to D. Similarly, tuple η_4 = $\{T_2$, 108 , $\{\eta_1$, $\eta_2\}$, $\{(\mathtt{I_1}$, $\mathtt{e_{13}})\}\}$ is added to capture the synchronization at location 108.

Figure 8 shows the set of all tuples that constitute D after the execution of $\mathtt{I}_1$ and $\mathtt{I}_2$. Observe that η_5 and η_6 have $\{(\mathtt{I_1}$,$\mathtt{e_{20}})$, $(\mathtt{I_2}$,$\mathtt{e'_5})\}$ and $\{(\mathtt{I_1}$,$\mathtt{e_{22}})$, $(\mathtt{I_2}$,$\mathtt{e'_7})\}$ as the fourth element respectively. Essentially, this means that the respective lock nodes can be created by executing either $\mathtt{I}_1$ or $\mathtt{I}_2$.

3.2 Cycle detector

Identifying the combination of lock acquisitions that point to a potential deadlock can be achieved by performing cycle detection on the lock dependency relation, D. We describe the process of cycle detection in this subsection.

We define a *path*, ρ = $(\eta_1, \eta_2, ..., \eta_k)$, as a sequence of lock nodes, where for each $i \in [2, k]$, there exists some $\eta_i^f \in \mathtt{H}(\eta_i)$[3] such that $\tau(\eta_{i-1}) = \tau(\eta_i^f)$. If there exists a $\eta_1^f \in \mathtt{H}(\eta_1)$ such that $\tau(\eta_k) = \tau(\eta_1^f)$, then we consider the path to be a *cycle* represented by θ.

We elaborate on these definitions using Figure 9. It shows the nested acquisitions made by single threaded test cases $\mathtt{I}'_1, \mathtt{I}'_2, \ldots, \mathtt{I}'_k$. For example, $\mathtt{I}'_1$ makes a nested acquisition by acquiring a lock on an object of type τ_1 while holding a lock on an object of type τ_k. The lock nodes corresponding to the first and second acquisitions made by $\mathtt{I}'_1$ are represented by nodes η_1^f and η_1 respectively. $\mathtt{H}(\eta_1)$ contains η_1^f to specify the nested acquisition. Similarly, other test cases make nested acquisitions represented by η_2, η_3 ... η_k respectively. We can now construct paths to show the possibility of object sharing. For example, observe the possibility of object sharing between test cases, $\mathtt{I}'_1$ and $\mathtt{I}'_2$, because the types of η_2^f and η_1 are equal to τ_1. This possibility of sharing creates a path from η_1 to η_2. Extending this further, we are able to construct a path $(\eta_1, \eta_2, ...\eta_{k-1}, \eta_k)$. We declare this path to be a cycle because $\tau(\eta_k) = \tau(\eta_1^f) = \tau_k$.

Algorithm 1 Cycle Detector

Input: Lock dependency relation (D), Max cycle length (κ)
Output: Set of cycles (Θ)

```
1:  i ← 1
2:  for every η ∈ D do
3:      if H(η) ≠ ∅ then D_1 ← D_1 ∪ {(η)}
4:          ρ ← concat((η), η)
5:          if cycle(ρ) then Θ ← Θ ∪ {ρ} end if
6:      end if
7:  end for
8:  while (i < κ and D_i ≠ ∅) do
9:      for every pair ρ ∈ D_i , η ∈ D s.t. η ∉ ρ do
10:         /* concatenate a node to the path */
11:         ρ' ← concat(ρ, η)
12:         if cycle(ρ') then
13:             if unique(ρ', Θ) then Θ ← Θ ∪ {ρ'}
14:             end if
15:         else if path(ρ') then  D_{i+1} ← D_{i+1} ∪ {ρ'}
16:         end if
17:     end for
18:     i ← i + 1
19: end while
```

The length of the path is given by the number of elements in the sequence. We define D_i as a set of paths with length i and bound the maximum length of the cycle by a user-tunable parameter κ. Having an upper bound on the length of the cycle is appropriate from a practical perspective to bind the running time of the algorithm.

We define auxiliary functions `path` and `cycle` which determine whether a sequence of lock nodes is a path and cycle respectively. Also, a cycle of length k can be represented as k different lock node sequences (by rotating). Therefore, given a set of cycles Θ and a candidate cycle θ, we define a

[3] f represents the first lock in a nested lock acquisition.

function unique(θ, Θ) which checks whether cycle θ is not already represented by some cycle in Θ. We define function concat that takes a sequence and an element as input and returns the concatenated sequence as output.

We present the approach for cycle detection in Algorithm 1. It takes the lock dependency relation, D, and maximum cycle length, κ, as input and uses bottom-up dynamic programming to find cycles. D_1 is initialized to contain sequences of unit length where each sequence represents one nested acquisition. Further, each node that has a path to itself forming a self loop is added to the set of detected cycles (lines 4-5). Subsequently, in each iteration (lines 9-18), we construct a path of length one higher than the path constructed in the previous iteration. For this purpose, each path in D_i is *concatenated* with a node contained in set D using function concat. If the concatenated sequence forms a *cycle*, then it is added to the set of detected cycles, Θ, as long as none of the existing cycles in Θ represent the same cycle. This is achieved by using the unique function. If the sequence only qualifies as a path, then it is added to the set D_{i+1}. The process terminates when no new paths are constructed in an iteration or the number of iterations exceed κ.

D_1	$\{(\eta_2), (\eta_3), (\eta_4), (\eta_6), (\eta_8)\}$
D_2	$\{(\eta_2, \eta_3), (\eta_2, \eta_4), (\eta_6, \eta_2), (\eta_8, \eta_2)\}$
Θ	$\{(\eta_2, \eta_2), (\eta_3, \eta_6), (\eta_3, \eta_8), (\eta_4, \eta_6), (\eta_4, \eta_8)\}$

Table 3. Paths and cycles generated by executing Algorithm 1 with D from Figure 8 and κ = 2.

We will use the D shown in Figure 8 to demonstrate the working of Algorithm 1. D_1 is initialized to a set of unit length sequences as shown in Table 3. Every node is also checked for a self loop. Node η_2 forms a self loop as it acquires a lock on object of type T_1 while holding a lock on object of type T_1 (specified by node η_1). Therefore, sequence (η_2, η_2) that forms a path as well as a cycle is added to Θ. Subsequently, other paths of length two are constructed. For example, sequence (η_3, η_6) satisfies the property of a cycle. This is because there exists $\eta_5 \in \mathtt{H}(\eta_6)$ such that $\tau(\eta_3) = \tau(\eta_5)$, and there also exists $\eta_2 \in \mathtt{H}(\eta_3)$ such that $\tau(\eta_6) = \tau(\eta_2)$. Similarly, other paths of length two that also form a cycle are constructed and added to Θ. Paths that do not qualify as cycles are added to D_2 as shown in Table 3. At the end of the loop, D_2 contains four paths and Θ contains five cycles. The execution terminates as κ is set to 2.

3.3 Synthesizer

Armed with the detected cycles and the logged execution traces, the Synthesizer attempts to transform the cycles into potential deadlocks. To accomplish this, it detects the method invocations in the test cases responsible for the creation of the input cycle. It also outputs the necessary additional constraints that need to be satisfied by these method invocations to create a possible deadlock. Therefore, for every input cycle, the Synthesizer detects the sequence of method invocations along with a list of test cases that make these invocations. It also outputs constraints on the parameters used in these method invocations. If these methods are invoked concurrently by distinct threads satisfying the generated constraints, then a deadlock may manifest.

3.3.1 Overview

For each cycle, $\theta = (\eta_1, \eta_2, ..., \eta_k)$, the lock nodes represent nested lock acquisitions. Each nested lock acquisition, η_i, is associated with invoking some method m from one of the tests in the seed testsuite (I). We construct the initial sequence of methods (M) for θ by collecting all the invoking methods. Subsequently, for each method $\mathtt{m} \in \mathtt{M}$, we generate constraints such that invoking m while satisfying the constraints will create a cyclic lock acquisition on lock objects. Broadly, we perform the following steps for a detected cycle:

1. For each $\eta \in \theta$, identify the associated method invocation (represented by m) in I.
2. Identify the parameters of m, on which the lock acquisitions represented by the nodes in θ are data dependent. This will enable our analysis to alter the lock acquisitions to manifest the cycle.
3. Infer constraints on the parameters of all $\mathtt{m} \in \mathtt{M}$ such that invoking the methods in M while satisfying the constraints on the parameters can lead to a deadlock.

Consider the cycle (η_3, η_6) detected in the running example from Figure 6. To transform the input cycle (η_3, η_6) into a deadlock, the goal of the Synthesizer is to derive the following:

- foo and bar are the two methods that need to be invoked concurrently by different threads.
- The first[4] and third parameters ($\mathtt{a}_1$,$\mathtt{b}_1$) of foo and the first and second parameters ($\mathtt{b}_2$,$\mathtt{a}_3$) of bar are interesting as they can influence the lock acquisitions that create the cycle. This is because a lock is acquired on the mutex fields of objects represented by these parameters.
- The object sharing across threads must be such that $\mathtt{a}_1$.mutex and $\mathtt{b}_1$.mutex need to be equal to $\mathtt{a}_3$.mutex and $\mathtt{b}_2$.mutex respectively.

In the subsequent subsections, we describe the process of deriving the above information.

3.3.2 Identifying method invocations

Algorithm 2 provides an approach to identify method invocations that can lead to a potential deadlock. The algorithm takes as input a cycle, $\theta = (\eta_1, \eta_2, ..., \eta_k)$, and the execution traces. For each node η in the cycle, it obtains the test that caused the node to be constructed and finds the corre-

[4] Recall that we refer to the receiver as the first parameter of a method.

Algorithm 2 Method Invocation Identifier

Input: Cycle (θ), Execution Traces (Σ)
Output: Method Invocation Index sequence (MI), Test sequence (I_M), Invoked Method Sequence (M)

```
1:  for every η ∈ θ do
2:      I_η ← test case retrieved from TI(η)
3:      σ ← Execution trace of I_η
4:      index ← trace index of nested lock acquisition in σ
5:      for index > 0 do /* Find required method invocation*/
6:          if σ[index] is enter and is invoked in I_η then
7:              m ← method invoked at σ[index]
8:              MI ← concat(MI,index)
9:              I_M ← concat(I_M,I_η)
10:             M ← concat(M,m)
11:             index ← 0 /* exit loop */
12:         end if
13:         index ← index−1
14:     end for
15: end for
```

sponding execution trace σ_η. The location of the nested acquisition in the execution trace is obtained from $\texttt{TI}(\eta)$ and is represented by index (lines 2-4). We analyze the trace backwards and identify the closest enter method event that corresponds to a method invocation from the test case $\texttt{I}_\eta$ (lines 5-15). The parameters can be manipulated for this method as the invocation of this method is within the test case and is not in the library implementation. The index of this method invocation is added to a method invocation index sequence MI, the test case responsible is added to the test sequence $\texttt{I}_\texttt{M}$ and the signature of the invoked method is added to the invoked method sequence M. When all the nodes in the cycle are processed, the algorithm outputs the sequences MI, $\texttt{I}_\texttt{M}$ and M.

We illustrate the process of applying Algorithm 2 on cycle (η_3, η_6) shown in Table 3. From $\texttt{TI}(\eta_3)$ shown in Figure 8, we find that the test responsible for creating η_3 is $\texttt{I}_1$. Our approach performs a backward search on the execution trace associated with $\texttt{I}_1$ (see Table 2) and obtains the invocation of method foo at index $\texttt{e}_4$. The method invocation index sequence MI is updated to ($\texttt{e}_4$), $\texttt{I}_\texttt{M}$ is updated to ($\texttt{I}_1$) and the sequence M is updated to (foo). For node η_6, the algorithm can choose either one of the test cases, $\texttt{I}_1$ or $\texttt{I}_2$ (see Figure 8). Assume the algorithm picks $\texttt{I}_2$ and searches the execution trace corresponding to $\texttt{I}_2$, it will identify the invocation of bar at index $\texttt{e}'_3$ (see Table 2) and updates the output sequences MI, $\texttt{I}_\texttt{M}$ and M appropriately. As all nodes in the cycle are processed, the output of the algorithm for the cycle will be MI = ($\texttt{e}_4$, $\texttt{e}'_3$), M = (foo,bar) and $\texttt{I}_\texttt{M}$ = ($\texttt{I}_1$, $\texttt{I}_2$).

3.3.3 Identifying relevant parameters

For cycle $\theta = (\eta_1, \eta_2, ..., \eta_k)$, we already have the sequence of method invocations (MI), the sequence of methods (M) and the sequence of tests that make the invocations ($\texttt{I}_\texttt{M}$). Until now, for ease of explanation, we just used the second lock node of a nested acquisition while defining θ without specifying the first lock node. A nested acquisition η_i can be decomposed into two lock nodes and can be represented as $\eta_i^f \rightarrow \eta_i$ as shown in Figure 9, where η_i^f and η_i are the first and second lock nodes respectively. We define function firstlock for a given cycle θ and node η_i, which returns *a set of nodes* that can be potentially used as η_i^f.

For every nested lock acquisition, we can pick one of the nodes from the set of nodes returned by $\texttt{firstlock}(\theta, \eta_i)$ as the first lock node and use η_i as the second lock node. For example, if we consider the cycle $\theta = (\eta_3, \eta_6)$ and consider the nested lock acquisition η_3, the $\texttt{firstlock}(\theta,\eta_3)$ will return $\{\eta_1,\eta_2\}$ because the first lock in the nested acquisition can correspond to either of these nodes as they have the same type $\texttt{T}_1$. The second lock node is η_3. Given the first and second lock nodes, the goal now is to determine the data dependence of the lock acquisitions to the appropriate parameters in the method invocation.

$b : Var \cup \{\perp\}$ $\quad$ $\texttt{F} : field \times field \times \cdots \times field$
$e_\texttt{m} : int$ $\quad$ $p : int \cup \{\perp, \texttt{err}\}$

$$\frac{b = \perp \quad \texttt{F} = \emptyset \quad p = \perp \quad b' = x}{\{b, \texttt{F}, e_\texttt{m}, p\} \xrightarrow{\texttt{lock(x)}} \{b', \texttt{F}, e_\texttt{m}, p\}} \text{(LOCK)}$$

$$\frac{b = x \quad \texttt{F} = f \oplus \texttt{F}' \quad p = \perp \quad b' = y}{\{b, \texttt{F}, e_\texttt{m}, p\} \xrightarrow{\texttt{load(x,f,y)}} \{b', \texttt{F}', e_\texttt{m}, p\}} \text{(LOAD)}$$

$$\frac{b = x \quad \texttt{F}' = f \oplus \texttt{F} \quad p = \perp \quad b' = y}{\{b, \texttt{F}, e_\texttt{m}, p\} \xrightarrow{\texttt{store(x,f,y)}} \{b', \texttt{F}', e_\texttt{m}, p\}} \text{(STORE)}$$

$$\frac{b = x \quad p = \perp \quad p' = \texttt{err}}{\{b, \texttt{F}, e_\texttt{m}, p\} \xrightarrow{\texttt{alloc(x)}} \{b, \texttt{F}, e_\texttt{m}, p'\}} \text{(ALLOC)}$$

$$\frac{p = \perp \quad \texttt{plist}[p'] = b \quad e_\texttt{m} = \texttt{iid}}{\{b, \texttt{F}, e_\texttt{m}, p\} \xrightarrow{\texttt{enter(iid,m,plist)}} \{b, \texttt{F}, e_\texttt{m}, p'\}} \text{(ENTER)}$$

$$\frac{}{\{b, \texttt{F}, e_\texttt{m}, p\} \xrightarrow{*} \{b, \texttt{F}, e_\texttt{m}, p\}} \text{(OTHER)}$$

Figure 10. Rules for identifying the relevant parameters. $\oplus$ is concatenation of an element to a sequence.

We perform a backward analysis of events in the execution trace starting from the location corresponding to the lock acquisition under consideration to derive the parameter that can *influence* the lock object. The set of rules for this processing is given in Figure 10. For each lock node, we maintain a parameter state, $P = \{b, \texttt{F}, e_\texttt{m}, p\}$, where

- b is a variable representing the base object. It is initialized to $\perp$.
- F is a sequence of field variables representing the list of field dereferences on b.

- $e_{\mathtt{m}}$ is the index of the method invocation `m` that caused the nested acquisition η and is obtained from Algorithm 2.
- p is the index of the parameter in the `plist` associated with the method invocation at index $e_{\mathtt{m}}$. It is also initialized to $\perp$. A value of `err` for p represents that the lock cannot be manipulated from the method invocation.

Intuitively, the tuple P represents the object obtained when b is dereferenced with fields present in `F`.

The typing rules are straight forward. Initially, the lock event corresponding to creation of the lock node under consideration is processed using the `LOCK` rule. Subsequent `lock` events will not match the rule because either b or p will not be $\perp$. The `LOAD` operation updates the base and field dereferences appropriately. If the event is created by assignment $x := y$, then the base alone is updated. Otherwise, if assignment $x.f = y$ is responsible for the event, base and field dereferences are updated. The `STORE` is very similar to `LOAD`. If an `alloc` event on a variable that is also currently the base is reached, then it implies that there is no relevant parameter to be found. This points to a local allocation within the implementation of the library that cannot be influenced by the client program. Therefore, p is updated to `err`. The `ENTER` rule is applied when the event associated with the method of interest ($e_{\mathtt{m}}$) is reached where the parameter index is obtained based on b.

We now illustrate the application of the typing rules on cycle $\theta = (\eta_3, \eta_6)$ from the running example (see Figures 6 and 8). Both η_3 and η_6 are nested lock acquisitions. We have $\mathtt{firstlock}(\theta, \eta_6) = \{\eta_5\}$ and $\mathtt{firstlock}(\theta, \eta_3) = \{\eta_1, \eta_2\}$. Because the latter has two elements, we can select any one of the two elements and let us assume that η_1 is selected as the first lock in the nested acquisition η_3. Therefore, we need to identify the parameters *influencing* the lock acquisitions represented by η_1 and η_3 for the nested acquisition η_3, and the lock acquisitions represented by η_5 and η_6 for the nested acquisition η_6. We will describe the application of the typing rules to track the parameter for η_3. The rules can be applied for η_1, η_5 and η_6 similarly.

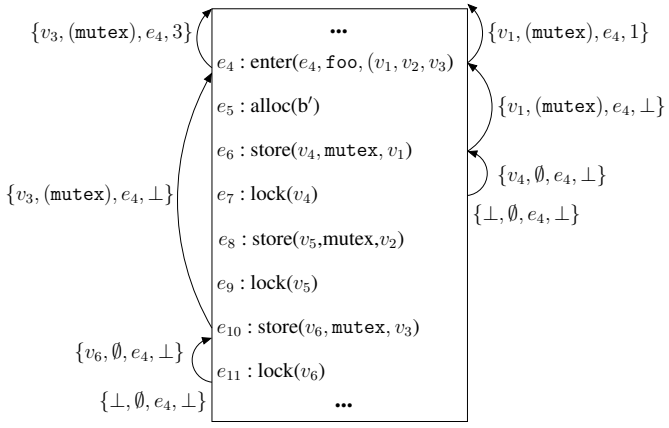

Figure 11. Parameter tracking for η_3 and η_1 using $\sigma(\mathtt{I}_1)$.

Figure 11 illustrates the application of typing rules on η_3. The associated execution trace is given in Table 2. For lock node η_3, the index in the execution trace is $\mathtt{e}_{11}$ and is the first element to be processed. Initially, the state is $\{\perp, \emptyset, \mathtt{e}_4, \perp\}$. We obtain the method invocation index as $\mathtt{e}_4$ for the nested acquisition η_3 from the previous phase, where relevant method invocations are identified. The `LOCK` rule matches the current state and hence is applied to transition the state to $\{v_6, \emptyset, \mathtt{e}_4, \perp\}$. We process the trace in the reverse order and will analyze the element at index $\mathtt{e}_{10}$. `STORE` is applied on this element to obtain the state $\{v_3, \{\mathtt{mutex}\}, \mathtt{e}_4, \perp\}$. For the rest of the elements up to (and including) $\mathtt{e}_5$, there is no change in the state. When $\mathtt{e}_4$ is processed, `ENTER` is applied to obtain the output state $\{v_3, \{\mathtt{mutex}\}, \mathtt{e}_4, 3\}$ for η_3, where 3 is the index into the `plist` of $\mathtt{e}_4$. This essentially means that we detect that the third parameter of `foo` in Table 2 as influencing η_3. The figure also depicts the transitions for η_1 given by the arrows on the right side and is self explanatory. The final parameter states for all the relevant nodes in the cycle under consideration is given in Table 4.

Nested Acquisition ($\eta^f \rightarrow \eta$)	$P(\eta^f)$	$P(\eta)$
$\eta_1 \rightarrow \eta_3$	$\{v_1, \mathtt{(mutex)}, \mathtt{e}_4, 1\}$	$\{v_3, \mathtt{(mutex)}, \mathtt{e}_4, 3\}$
$\eta_5 \rightarrow \eta_6$	$\{v_1, \mathtt{(mutex)}, \mathtt{e}'_3, 1\}$	$\{v_2, \mathtt{(mutex)}, \mathtt{e}'_3, 2\}$

Table 4. Output of parameter tracking for cycle (η_3, η_6).

3.3.4 Constraint Generation

For the detected cycle, we have described the approach to identify the relevant method invocations and the appropriate parameters that can influence the lock acquisition in the previous two subsections. The goal of this subsection is to generate constraints on the parameters of these method invocations so as to manifest a deadlock. Intuitively, for a deadlock to manifest, the second lock acquired by one thread needs to be the same as the first lock acquired by another thread.

For a cycle, $\theta = (\eta_1, \eta_2, ..., \eta_k)$, where each nested lock acquisition η_i can be represented as $\eta_i^f \rightarrow \eta_i$, the following constraint set ($\mathcal{C}$) is generated for all $i \in [1, k]$:

- $P(\eta_{i-1}) \equiv P(\eta_i^f)$, if $i \in [2, k]$
- $P(\eta_k) \equiv P(\eta_i^f)$, otherwise

A constraint $P(\eta_i) \equiv P(\eta_j)$ indicates that $P(\eta_i)$ and $P(\eta_j)$ must represent the *same* object. In other words, the parameter b_i passed to invocation at $e_{\mathtt{m}_i}$ when dereferenced with fields contained in $\mathtt{F}_i$ and the parameter b_j passed to invocation at $e_{\mathtt{m}_j}$ when dereferenced with fields contained in $\mathtt{F}_j$ must represent the same object.

For the cycle (η_3, η_6) from the running example, we generate $\mathcal{C} = \{P(\eta_3) \equiv P(\eta_5),\ P(\eta_6) \equiv P(\eta_1)\}$. Table 4 already gives the final states of P for the relevant lock nodes. The generated constraints imply that the object corresponding to the `mutex` field of the first parameter of `foo` must be

equivalent to the object corresponding to the mutex field of the second parameter of bar. Similarly, the objects corresponding to the mutex fields of the third parameter of foo and the first parameter of bar must be equivalent.

After achieving the goals mentioned in Section 3.3.1, we now use the obtained information to describe the process of generating a deadlock revealing test case.

3.4 Generator

For each cycle, the Generator uses the data generated from the previous phases to synthesize a multithreaded test. It takes the sequence of test cases, method signatures and method invocation indices ($\mathtt{I_M}$, M, MI obtained from Algorithm 2) and a set of constraints ($\mathcal{C}$ derived in Section 3.3.4) as input and constructs a test case which spawns concurrent threads, where each thread invokes a method in M.

```
Input:
    Sequence of test cases (I_M), Sequence of methods (M),
    Sequence of invocation indices (MI), Constraint set (C).
Synthesized test:
1   O ← ∅;
2   for every test case I_i in I_M do
3     O_i ← collectobjects(I_i, MI[i]);
4   done
5   for every constraint C_i(P_i ≡ P_j) in C do
6     d ← p(P_i); s ← p(P_j);
7     F_d ← F(P_i); F_s ← F(P_j);
8     o' ← enforce(O_i[d], F_d, O_j[s], F_s);
9      if o' ≠ null then reinitialize O_i[d] with o';
10     else declare infeasible and exit;
11     end if
12  end for
13  for every method m_i in M
14     invoke m_i with O_i as parameters from a new thread;
15  end for
```

Figure 12. Skeleton of synthesized multithreaded test.

We need to obtain appropriate objects that can be passed as parameters to the method invocation which also satisfy the constraints. Because, we already have the sequential test cases that construct the objects for method invocations, we execute the sequential test cases as part of the multithreaded test case to collect the objects. Subsequently, constraints are enforced on the collected objects before passing them as parameters to the method invocations in the spawned threads.

Figure 12 presents the generic structure of the synthesized test. Initially, it collects the objects from tests in $\mathtt{I_M}$ with the help of method collectobjects (lines 2-4). Method collectobjects executes the sequential test input to it and allows the test to proceed until the required objects are collected. After collecting the objects, it terminates the execution of the test to ensure that the state of the collected objects are unmodified. The constraints in $\mathcal{C}$ are enforced between lines 5-12 by the enforce algorithm presented in Algorithm 3 and will be discussed subsequently. Due to the availability of the necessary invocation context, multiple threads (depending on the number of edges in the cycle) are spawned (lines 13-15) and each thread invokes relevant methods with the appropriate objects. Analyzing the execution of the synthesized multithreaded test using a dynamic deadlock detector (e.g., iGoodLock) will enable detection of potential deadlocks. OMEN reports the synthesized multithreaded test and the detected deadlocks.

We now explain the working of method enforce presented in Algorithm 3 which is used to enforce constraints. The goal of this algorithm is to take two object instances and construct an object which is shareable across multiple threads. The method takes source and destination objects along with the field dereferences as input and outputs a modified object (or null). As the first step, it dereferences the source object (o_s) with fields in $\mathtt{F}_s$ and makes an assignment to the pointer obtained by dereferencing the destination object (o_d) with fields in $\mathtt{F}_d$. It is not necessary that such an assignment is always feasible. For example, private fields in the library cannot be assigned from a client. If the assignment is feasible, it performs the relevant object assignments (line 2). Otherwise, it checks whether the object assignment can be performed for the owner objects (lines 4-8) by invoking enforce with modified $\mathtt{F}_s$ and $\mathtt{F}_d$ recursively. This may also be infeasible as the types of the owner objects may be incompatible. This process continues until a feasible assignment is made and the modified destination object o_d is returned subsequently. If a feasible assignment is not possible, a null is returned suggesting the infeasibility of synthesizing a test.

The underlying reason for checking the feasibility of assignments recursively is to be less disruptive. For example, consider a lock is obtained on $o_1.f_1.f_2$ in one test and a lock is obtained on $o_2.f_1.f_2$ in another test and the constraint is that these objects need to be shared. We can perform any one of the following three assignments (if they are feasible): (a) $o_1.f_1.f_2 = o_2.f_1.f_2$, (b) $o_1.f_1 = o_2.f_1$ and (c) $o_1 = o_2$. Among these assignments, the first assignment is the least disruptive change that can be made to satisfy the necessary constraint. If assignment (a) is infeasible, then (b) is considered. If assignment (b) is also infeasible, then (c) is considered.

We explain Figure 12 for cycle (η_3, η_6) from the running example. From the output of the Synthesizer, we get MI = ($\mathtt{e}_4$, $\mathtt{e}'_3$), $\mathtt{I_M}$ = ($\mathtt{I}_1$, $\mathtt{I}_2$) and the constraint set $\mathcal{C} = (P(\eta_3) \equiv P(\eta_5), P(\eta_6) \equiv P(\eta_1))$. Initially, $\mathtt{I}_1$ is executed until the invocation of foo at $\mathtt{e}_4$ and the objects $(\mathtt{a}_1, \mathtt{a}_2, \mathtt{b}_1)$ associated with the invocation are collected into $\mathcal{O}_1$, as shown in Figure 13(a). Similarly, $\mathtt{I}_2$ is executed and the objects associated with the invocation of bar at $\mathtt{e}'_3$ are collected into $\mathcal{O}_2$, as presented in Figure 13(b). We then need to enforce the generated constraints. For the constraint $P(\eta_3) \equiv P(\eta_5)$, the ap-

Algorithm 3 enforce

Input: o_d : Destination object, $\mathtt{F}_d : (f_d^1, f_d^2, ...f_d^m)$
o_s : Source object, $\mathtt{F}_s : (f_s^1, f_s^2, ...f_s^n)$
Output: modified object or `null`

1: **if** $o_d.f_d^1.f_d^2...f_d^m \leftarrow o_s.f_s^1.f_s^2...f_s^n$ is feasible **then**
2: $o_d.f_d^1.f_d^2...f_d^m \leftarrow o_s.f_s^1.f_s^2...f_s^n$; Return o_d;
3: **else if** $f_d^m \neq$ `null` & $f_s^n \neq$ `null` **then**
4: **if** `typeOf(`f_d^m`) = typeOf(`f_s^n`)` **then**
5: $\mathtt{F}'_d \leftarrow (f_d^1, f_d^2, ...f_d^{m-1})$
6: $\mathtt{F}'_s \leftarrow (f_s^1, f_s^2, ...f_s^{n-1})$
7: Return `assign` $(o_d, \mathtt{F}'_d, o_s, \mathtt{F}'_s)$
8: **end if**
9: **end if** Return `null`

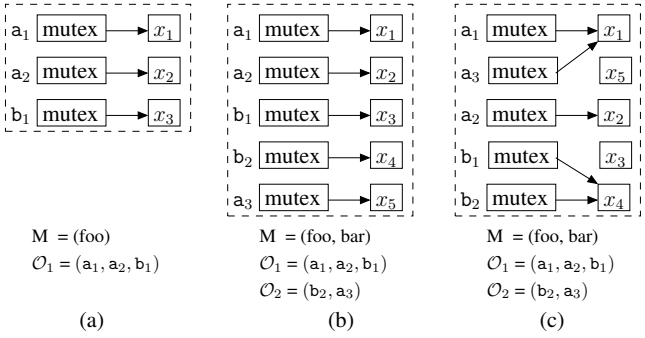

Figure 13. Object assignments for the parameters

proach calls `enforce`($\mathtt{b}_1$, (`mutex`), $\mathtt{b}_2$, (`mutex`)) resulting in the assignment of x_4 to $\mathtt{b}_1$.`mutex`, as shown in Figure 13(c). Similarly, the other assignment for $P(\eta_6) \equiv P(\eta_1)$ is performed. Subsequently, the test spawns two threads which invoke methods `foo` and `bar` with the appropriate objects as parameters. Since the objects are now shared across threads, a dynamic deadlock detector can analyze the execution and detect a potential deadlock.

4. Implementation

We have implemented `OMEN` in `Java` to synthesize multithreaded tests for `Java` libraries. The implementation uses the `soot` [26] bytecode analysis framework. We use `iGoodLock` [14] to detect deadlocks on the synthesized test executions. Many of the implementation choices in our tool are driven by practical considerations. We discuss a few of them in this section.

Instrumentation: We instrument the program using `soot` to generate the execution traces. We identify the relevant operations including lock, unlock, method entry, method exit, loads and stores of the variables and fields and insert hooks to generate the trace elements as defined in Section 3.1.1. We define the load and store events in the trace with a maximum of one level of field dereference. Therefore, we emit multiple level of dereferences in the source program as multiple loads or stores involving temporary variables. For example, a dereference $a.f_1.f_2$ is considered as two operations ($v_1 = a.f_1, v_2 = v_1.f_2$) and appropriate loads and stores are emitted.

Cycle detection:

- We define a path and a cycle based on the equality of the corresponding types. In our implementation, we also consider the sub-typing relation and consider the types to be equal even if one is a subtype of the other.
- Each cycle $\theta = (\eta_1, \eta_2, ...\eta_k)$ can encode multiple deadlocks depending upon the number of lock nodes present in each set returned by `firstlock`(θ, η_i). For ease of explanation, we used one lock node from the set of first lock nodes and explained one possible deadlock. In our implementation, we search for all potential deadlocks encoded by the cycle. The count of the potential deadlocks given by a cycle is a product of the number of elements in each set returned by `firstlock`(θ, η_i), where i ranges from 1 to k.
- If the nested acquisitions are guarded by the same static or shared lock, then such *guard* locks can be a source of imprecision for our analysis. Our implementation does not handle them and may synthesize unnecessary tests. However, when `iGoodLock` [14] analyzes the synthesized tests, it will not report any potential deadlocks and the associated imprecision is eliminated.
- Multiple test cases can generate the same lock node (e.g., η_3 from Figure 8) and our analysis picks a random test case to identify the method invocation corresponding to the creation of the lock node. A poor choice of the test case may result in `OMEN` not being able to synthesize a test even in the presence of a possible deadlock. Our implementation can easily be extended to explore other choices of test cases when it fails to synthesize a test.

Tracking parameters: The locks under consideration may be influenced by other method invocations instead of the current invocation. For example, a constructor may set a field before the lock is acquired in a method invoked from the client. In this context, our analysis will be unable to synthesize a test. Furthermore, our implementation does not handle aliasing and collections. This may result in imprecision of the analysis where `OMEN` will fail to synthesize multithreaded tests. However, our experimental results do not show any imprecision due to these issues.

5. 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. The information pertaining to the benchmarks

	ConTeGe			OMEN									
Class name				$\|D\|$	$\|\Sigma\|$	Time (in secs)			$\|\Theta\|$	$\|$CMI$\|$	Final Output		
	Tests	Time(s)	DL			S	G	DD			Tests	DL	TP
DynamicBin1D	4400	4036	0	64	1025K	352	16	32	36	36	6	36	21
CharArrayWriter	4269	2425	0	12	94K	12	2	2	1	1	1	1	1
ClosableByteArrayOutputStream	3105	2287	0	25	188K	25	2	5	1	1	1	1	1
ClosableCharArrayWriter	2758	2100	0	25	158K	20	2	4	1	1	1	1	1
HashTable	4218	2355	0	28	553K	79	6	26	28	21	15	20	19
Stack	4094	3139	0	46	219K	38	4	11	1	1	1	1	1
ByteArrayOutputStream	3901	2002	0	15	146K	20	2	2	1	1	1	1	1
MonitoredObjectImpl	-	-	-	-	-	-	-	-	-	-	-	-	-
Total			0								26	61	45

Table 6. Experimental results with $\mathtt{I_R}$ as the seed testsuite. D: Lock Dependency Relation, Σ: Execution traces, S: Cycle Detection and Parameter Synthesis, G: Generator, DD: Deadlock Detection, Θ: Detected cycles, CMI: Concurrent Method Invocations, DL: Deadlocks, TP: True Positives.

Class name	Version	LoC	$\|\mathtt{M_{total}}\|$
DynamicBin1D	colt-1.2.0	8033	35
CharArrayWriter	classpath-0.98	59	13
ClosableByteArrayOutputStream	hsqldb-2.3.2	77	22
ClosableCharArrayWriter	hsqldb-2.3.2	88	22
HashTable	jdk1.7	1131	20
Stack	jdk1.7	501	5
ByteArrayOutputStream	jdk1.7	33	10
MonitoredObjectImpl	jdk1.7	189	14

Table 5. Benchmark Information. LoC: lines of code across classes covered by test cases. $|\mathtt{M_{total}}|$: number of public methods in the class.

used for our experiments is given in Table 5. colt is a high performance scientific computing library, classpath contains core class libraries for use with virtual machines and compilers, hsqldb is a leading SQL relational database software in Java, and jdk is the Java Development Kit. Among the analyzed classes, DynamicBin1D is declared as thread safe in its documentation. For the other classes, the results of our approach provide various ways that client code could deadlock, if the locks are not appropriately acquired before invoking the methods in the cycle. For our experiments, we set $\kappa = 2$.

We evaluate the effectiveness of OMEN across the following dimensions:

1. Ability to *automatically* synthesize deadlock revealing multithreaded tests compared to ConTeGe [22].
2. Ability to synthesize tests with a seed testsuite that is developed by third-party developers.
3. Ability to synthesize tests with a seed testsuite that contains sequential tests invoking all methods in the class implementation.

We now describe our experimental results for each of the above dimensions.

5.1 Automated synthesis

We provide just the class that needs to be tested as input to OMEN and generate a seed testsuite using Randoop [20]. Randoop generates sequential tests that test the various methods of the class and add them to the seed testsuite ($\mathtt{I_R}$). The total number of tests in $\mathtt{I_R}$ is limited to 500. After performing the various phases of the analysis on the input class, OMEN synthesizes multithreaded tests. The precise numbers are tabulated in Table 6.

In Table 6, we observe that a total of 26 multithreaded tests are synthesized across all the benchmarks as final output. These tests are executed and the executions analyzed using iGoodLock [14] which reports a list of deadlocks. However, OMEN eliminates redundant deadlocks and reports 61 *unique*[5] deadlocks across all benchmarks. Out of these 61 deadlocks, 45 are real deadlocks. If the client programs invoke the methods in these classes without appropriate synchronization, their executions can deadlock. We also increased the maximum length of a cycle (κ) to five. OMEN did not detect any new deadlock across all the benchmarks. We also did not notice any significant increase in the analysis time because the maximum number of lock nodes in D is 64.

In comparison, the number of tests that ConTeGe generates ranges from 2758 to 4400 and the execution time varies from 2002 to 4036 seconds. In spite of that, ConTeGe [22] is unable to detect a deadlocking scenario in *any* of these classes. Both our tools use the tests generated by Randoop as the seed testsuite. As we discussed earlier, ConTeGe is unable to make effective use of the seed testsuite as it *randomly* synthesizes concurrent executions. We attribute the effectiveness of our approach to the *directed* nature of synthesizing tests.

The number of cycles detected by the Cycle Detector varies from 1 to 36. The Synthesizer operates on these cycles and the execution traces to synthesize a set of concurrent method invocation sequences (represented by CMI), where the number of elements in the CMI ranges from 1 to 36. For most benchmarks, the number of such sequences is equivalent to the cycles detected. However, seven cycles cannot be

[5] For example, in Table 1, OMEN considers deadlocks 2 and 3 as the same deadlock and reports them as one possible deadlock.

	I_{TP}								I_{MS}							
Class name	Time (in secs)			$\|\Theta\|$	|CMI|	Final Output			Time (in secs)			$\|\Theta\|$	|CMI|	Final Output		
	S	G	DD			|T|	DL	TP	S	G	DD			|T|	DL	TP
DynamicBin1D	89	12	32	66	21	15	55	35	68	12	29	91	55	10	55	35
CharArrayWriter	2	0.4	2	1	1	1	1	1	2	0.3	2	1	1	1	1	1
ClosableByteArrayOutputStream	3	0.4	2	1	1	1	1	1	2	0.4	2	1	1	1	1	1
ClosableCharArrayWriter	3	0.4	3	1	1	1	1	1	2	0.4	2	1	1	1	1	1
HashTable	13	3	10	3	3	1	3	2	13	2	14	28	21	15	20	19
Stack	17	-	-	0	0	0	0	0	13	2	9	1	1	1	1	1
ByteArrayOutputStream	2	0.4	2	1	1	1	1	1	2	0.3	2	1	1	1	1	1
MonitoredObjectImpl	7	1	2	1	1	1	1	1	6	1	2	1	1	1	1	1
Total						21	63	42						31	81	60

Table 7. Experimental results, – S: Cycle Detection and Parameter Synthesis, G: Generator, DD: Deadlock Detection, Θ: Detected cycles, CMI: Concurrent Method Invocations, |T| : Tests generated, DL: Deadlocks, TP: True Positives.

synthesized into method invocations for HashTable. This is because of lock acquisitions on *local* objects. As a result, the client of the class cannot manipulate the parameters to the method invocation appropriately. Hence, the detected cycle can be discarded as it cannot manifest as a deadlock. The Generator uses the elements in the CMI to synthesize multithreaded tests. There is a reduction in the number of tests from the number of elements in the CMI in DynamicBin1D because many concurrent method invocations share the same constraints.

The number of nodes generated in the lock dependency relation, D, ranges from 12 to 64. The size of the execution traces is significantly high (e.g., approximately 1M for DynamicBin1D) due to the number of load and store events generated for all the 500 tests. The processing time for Cycle Detector and Synthesizer is a function of the size of D and Σ for each benchmark. The maximum time is taken by DynamicBin1D as it has the largest $|D|$ and $|\Sigma|$. The processing time for the Generator is relatively low because it just needs to synthesize a multithreaded test based on the constraints derived previously. The time to detect deadlocks is associated with executing the synthesized tests and analyzing the execution with the deadlock detector. In summary, the total time for using the tool does not exceed seven minutes for any of the benchmarks. When compared to the complexities associated with manually reasoning about the possibility of deadlocks, the ineffectiveness of ConTeGe to detect problems and the overall size of the state space, the time taken by OMEN is insignificant.

We also observe that there are 45 true positives among 61 detected deadlocks. The underlying imprecision is due to the employed deadlock detector. We are in the process of integrating OMEN with WOLF [23], a deadlock detector and reproducer designed by us, which will not only detect the deadlocks but will also reproduce the deadlocks automatically. Our approach also suffers from false negatives as it based on dynamic analysis. For example, we are unable to detect potentials deadlocks in MonitoredObjectImpl. It is necessary for our approach to analyze some executions, preferably with reasonable coverage from a sequential perspective, to synthesize deadlock revealing multithreaded tests. Randoop is unable to generate any method sequences for MonitoredObjectImpl. In such scenarios where automatic sequential program test generators like Randoop do not provide sufficient coverage, our approach can fall back on manually developed seed testsuite. We now discuss the impact of using manually developed tests with OMEN.

5.2 Manually written seed testsuite

We also evaluated our approach when the seed testsuite is developed by third-party programmers. Moreover, we also study the effectiveness when a seed testsuite with good coverage is provided by using the testsuite developed by the first author.

5.2.1 Seed testsuite from third-party developers

We obtained sequential and multithreaded test cases for the classes under consideration from developers who are not associated with our project or with the development of the libraries under consideration. The four volunteers for our experiments include two graduate students, a researcher and a software engineer.

We executed the multithreaded tests developed by the volunteers and analyzed them using iGoodLock [14]. The detector did not report *any* deadlock across all the benchmarks. Subsequently, we used the set of sequential tests as the seed testsuite (I_{TP}) and applied OMEN to synthesize multithreaded tests. Table 7 presents the corresponding data.

OMEN is able to detect more deadlocks for DynamicBin1D with I_{TP} as the seed testsuite compared to I_R. This is because the tests developed by the users cover the method sampleBootstrap (see Figure 2). Randoop had missed generating a sequential test for this method and therefore our analysis had missed detecting a few defects. Our analysis is able to leverage good sequential coverage provided by I_{TP} and is able to decipher the deadlocks quickly. Even though the volunteers were not familiar with DynamicBin1D (colt), we are able to use simple sequential tests to synthesize multithreaded tests that enable deadlock detection. Re-

call that the deadlock detector did not find any defects on the multithreaded tests developed by the volunteers.

The downside to writing manual tests is that they need not provide good coverage under all circumstances. With I_{TP} as the seed testsuite, fewer deadlocks are detected in Stack compared to I_R as the seed. In this case, the volunteers' tests could not drive the objects to the required state whereas Randoop generated sequential tests that are able to drive the executions effectively. Manually perusing the tests, we observed a significant difference in the quality of the tests (in terms of coverage) written by the volunteers. As a result, we wanted to compare the effectiveness of our approach when the seed testsuite has *good* coverage. We present the findings in the next subsection.

5.2.2 Seed testsuite from the first author

To generate a seed testsuite with good coverage, the first author took all the public methods in the classes under consideration and developed single-threaded tests invoking the methods. These tests are made part of the seed testsuite (I_{MS}) and form the input to OMEN. Table 7 presents the numbers corresponding to the synthesized multithreaded tests and the deadlocks detected. Because the testsuite has good coverage of the various methods, the results of our approach are also significantly better. OMEN is able to detect 81 deadlocks across all benchmarks and our manual analysis shows that 60 detected deadlocks are true positives.

In practice, we believe that most robust software library implementations have tests that provide good coverage of the various features. However, the tests need not necessarily be effective for detecting deadlocks (or other concurrency defects) because of the complexities associated with reasoning about them. We believe deployment of OMEN to synthesize multithreaded tests in such scenarios will significantly improve the quality of the libraries and make it a valuable tool in the software development process.

6. Related Work

To the best of our knowledge, the closest effort to automatically generate tests to detect concurrency violations is by Pradel and Gross [22]. In ConTeGe [22], 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. One of the drawbacks of their approach is in generating random method invocations to detect these violations. There can be a number of method invocations and invocation contexts which can be completely irrelevant from the perspective of detecting bugs. We address this drawback by proposing a *directed* approach for identifying the method invocations and creating an appropriate invocation context that will enable deadlock detection.

Eslamimehr and Palsberg [7] propose a technique based on integrating concolic execution with race detection [8]. This increases the number of detected races due to increased code coverage. Our tool, OMEN, differs from their approach in terms of the intended application (race detection *vs* deadlock detection) and the nature of the employed analysis. We leverage the data from dynamic analysis engines to generate tests that can expose defects.

Many dynamic analysis approaches [3, 13, 14, 23, 28] are designed for detecting deadlocks. In [23], we designed a precise technique for detecting and reproducing deadlocks using dynamic analysis. Joshi *et al.* [14] proposed the concept of active testing for dynamic deadlock detection and use randomization to reproduce deadlocks. Cai and Chan [3] reduce the overhead of cycle detection by maintaining a pruned lock dependency graph. ConLock [28] identifies specific scheduling constraints that need to be maintained for deadlocks to be reproduced reliably. Generalized deadlocks involving communication patterns can be detected using the dynamic analysis designed by Joshi *et al.* [13]. All these approaches are fundamentally dependent on the quality of the analyzed executions to efficiently detect deadlocks. Our approach for automatically synthesizing multithreaded test cases *complements* these techniques. In fact, our current implementation is already integrated with iGoodLock [14].

A number of elegant techniques have been proposed [9, 12, 19, 20, 24, 25] for testing sequential programs to increase code coverage. We propose an automatic test synthesis for multithreaded libraries that can leverage these techniques and therefore stands to gain from the advances made for sequential test generation.

Static analysis has been used to detect deadlocks [5, 15, 18, 27]. The false positive rate can be significant with static analysis based approaches [14]. More significantly, the defect report needs to be manually triaged to verify its correctness. Multithreaded test synthesis makes this process less cumbersome.

Concurrency testing frameworks [6, 11] can be used to develop better tests for the seed testsuite of our approach. We believe integration of our approach with systematic state space explorers [16, 17, 30] will enable those techniques to detect more defects. Octet[2] reduces the dynamic analysis overhead significantly and we intend to investigate the performance gains achieved by integration OMEN with it. The approach based on concurrent function pairs [4] shares similar goals in reducing the bug detection effort. Reproducing the bugs [10, 21] that manifest occasionally in multi-threaded executions is orthogonal to our work as our goal is to detect defects before the software is deployed.

7. Conclusions

Designing effective multithreaded test cases is challenging and thread safety violations in multithreaded libraries can easily go undetected. We propose an approach for synthe-

sizing multithreaded tests to enable deadlock detection. Our approach is completely automatic and requires just the implementation of the library that needs to be tested. Elaborate experimentation shows the effectiveness of our tool, named `OMEN`, in detecting *real* deadlocks in thread-safe classes of popular libraries.

8. Acknowledgements

We thank the anonymous reviewers for their feedback which helped improve the presentation of the paper. We thank MHRD, Govt. of India for funding our research. We are grateful to Microsoft Research India and Google India for providing travel support for the first author. We thank Soham Ghosh for setting up `ConTeGe`, Monika Dhok and Rashmi Mudduluru for proof-reading early versions of the paper, and the third party developers for providing test cases.

References

[1] COLT API Documentation. `https://acs.lbl.gov/software/colt/api/`.

[2] M. D. Bond, M. Kulkarni, M. Cao, M. Zhang, M. Fathi Salmi, S. Biswas, A. Sengupta, and J. Huang. Octet: Capturing and controlling cross-thread dependences efficiently. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 693–712, 2013.

[3] Y. Cai and W. K. Chan. Magicfuzzer: Scalable deadlock detection for large-scale applications. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012.

[4] D. Deng, W. Zhang, and S. Lu. Efficient concurrency-bug detection across inputs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 785–802, 2013.

[5] J. Deshmukh, E. A. Emerson, and S. Sankaranarayanan. Symbolic deadlock analysis in concurrent libraries and their clients. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 480–491, 2009.

[6] T. Elmas, J. Burnim, G. Necula, and K. Sen. Concurrit: A domain specific language for reproducing concurrency bugs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 153–164, 2013.

[7] M. Eslamimehr and J. Palsberg. Race directed scheduling of concurrent programs. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 301–314, 2014.

[8] C. Flanagan and S. N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09.

[9] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05.

[10] J. Huang, C. Zhang, and J. Dolby. Clap: Recording local executions to reproduce concurrency failures. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13.

[11] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, and D. Marinov. Improved multithreaded unit testing. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 223–233, 2011.

[12] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *Proc. 22nd International Symposium on Software Testing and Analysis (ISSTA)*, July 2013.

[13] P. Joshi, M. Naik, K. Sen, and D. Gay. An effective dynamic analysis for detecting generalized deadlocks. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE, 2010.

[14] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09.

[15] S. McPeak, C.-H. Gros, and M. K. Ramanathan. Scalable and incremental software bug detection. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013.

[16] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07.

[17] S. Nagarakatte, S. Burckhardt, M. M. Martin, and M. Musuvathi. Multicore acceleration of priority-based schedulers for concurrency bug detection. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12.

[18] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09.

[19] R. Nokhbeh Zaeem and S. Khurshid. Test input generation using dynamic programming. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, 2012.

[20] C. Pacheco and M. D. Ernst. Randoop: Feedback-directed random testing for java. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, OOPSLA '07.

[21] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. Pres: Probabilistic replay with execution sketching on multiprocessors. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09.

[22] M. Pradel and T. R. Gross. Fully automatic and precise detection of thread safety violations. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12.

[23] M. Samak and M. K. Ramanathan. Trace driven dynamic deadlock detection and reproduction. In *Proceedings of the 2014 ACM SIGPLAN Conference on Principles and Practices of Parallel Programming*, PPoPP '14.

[24] K. Sen and G. Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06.

[25] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su. Synthesizing method sequences for high-coverage testing. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 189–206, 2011.

[26] 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 *In International Conference on Compiler Construction, LNCS 1781*, pages 18–34, 2000.

[27] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for java libraries. In *ECOOP 2005-Object-Oriented Programming*. Springer Berlin Heidelberg.

[28] C. Yan, W. Shangru, and W. K. Chan. Conlock: A constraint-based approach to dynamic checking on deadlocks in multi-threaded programs. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE '14.

[29] J. Yang, H. Cui, J. Wu, Y. Tang, and G. Hu. Making parallel programs reliable with stable multithreading. *Communications of the ACM*, 57(3).

[30] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: A coverage-driven testing tool for multithreaded programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12.

A. Problem Formulation

We formally present the problem of synthesizing multi-threaded test cases that is addressed in the paper.

Definition A.1. Lock operation sequence (α) is defined as a sequence of lock and unlock operations on a set of objects O such that,

$\alpha : \; l(o)\; u(o)\; \alpha \mid l(o)\; \alpha\; u(o) \mid \alpha\; l(o)\; u(o) \mid \in,$

where $o \in O$, $l(o)$ and $u(o)$ represent the acquire and release of lock on object o respectively.

The definition of α states that every lock is *always* followed by a corresponding unlock operation. It further states that if a lock is acquired on an object o' while the lock on object o is held, then the lock on o can be released only after the lock on o' is released.

Definition A.2. Cyclic Acquisition Property (CAP): A sequence of lock operation sequences, $\alpha_1, \alpha_2, ..., \alpha_n$ satisfy `CAP` if the following conditions hold:

1. $\alpha_i = {\beta_i}^1\; l(o_i)\; {\beta_i}^2\; l(o_{i+1})\; {\beta_i}^3$ and $u(o_i) \notin {\beta_i}^2$, where ${\beta_i}^1$, ${\beta_i}^2$ and ${\beta_i}^3$ correspond to a sequence of lock and unlock operations.
2. $o_n = o_1$.

The above property is a necessary condition for a deadlock to manifest. The first condition ensures the nested acquisition of o_{i+1} while holding a lock on object o_i. The second condition ensures the creation of a cycle. We now define the problem of identifying the combination of methods, with appropriate parameters, that need to be invoked concurrently across multiple threads for a deadlock to manifest.

Let `S` be the source class that needs to be tested to expose deadlocking scenarios while invoking the public methods in the class, $\mathtt{M_S}$ be the set of methods implemented in `S` and $\mathtt{M_I}$ be the set of all method invocations in the seed testsuite, `I`. A method invocation, $\mathtt{m_i} \in \mathtt{M_I}$, is defined as $o_i^0.\mu_j(o_i^1, o_i^2, ..., o_i^{n_{\mu_j}})$ where $\mu_j \in \mathtt{M_S}$, o_i^0 is the receiver object, $(o_i^1, o_i^2, ..., o_i^{n_{\mu_j}})$ are parameters to μ_j and n_{μ_j} is the number of parameters to μ_j. Let μ be a function that gives the mapping from the method invocation to the appropriate method implementation in `S` (i.e., $\mu(\mathtt{m_i}) = \mu_j$).

We define $\mathcal{O}_i = \{(o_i^0, o_i^1, ..., o_i^{n_{\mu(\mathtt{m_i})}})\}$ as the set of objects used in the invocation of $\mathtt{m_i}$. We represent the cumulative collection of the objects used in the method invocations across the testsuite as $\mathcal{O} = \bigcup_{i=1}^{|\mathtt{M_I}|} \mathcal{O}_i$. Let $\alpha(\mathtt{m_i})$ represent the lock operation sequence generated on execution of $\mathtt{m_i}$.

Definition A.3. Concurrent Method Invocation Synthesis Given $\mathtt{M_I}$ and $\mathcal{O}$, generate a set of concurrent method invocations, $\mathtt{M_C}$, such that,

1. There exists some sequence which is a permutation of $\bigcup_{k=1}^{|\mathtt{M_C}|} \alpha(\mathtt{m_k})$ that satisfies `CAP` (see Definition A.2), where $\mathtt{m_k} = o_k^0.\mu_j(o_k^1, o_k^2, ..., o_k^{n_{\mu_j}})$, $\mu_j \in \bigcup_{i=1}^{|\mathtt{M_I}|} \mu(\mathtt{m_i})$, $\mathcal{O}_k = \{o_k^0, o_k^1, o_k^2, ..., o_k^{n_{\mu_j}}\}$ and $\mathcal{O}_k \subseteq \mathcal{O}$,
2. For every pair $\mathtt{m_k}, \mathtt{m_{k'}} \in \mathtt{M_C}$, $\mathtt{m_k}$ and $\mathtt{m_{k'}}$ can be executed concurrently.

These conditions ensure that a set of method invocations is chosen from the seed testsuite, assigned parameters from the global collection of object instances used in the testsuite appropriately so that executing the methods in some order generates a sequence of lock operation sequences that satisfies `CAP`.

The problem addressed in the paper is to synthesize a test such that a different thread executes each method invocation in $\mathtt{M_C}$ concurrently. Such a concurrent execution can potentially be used to detect a deadlock in `S`. For any given $\mathtt{M_I}$ and $\mathcal{O}$, multiple sets of concurrent method invocations may exist leading to the generation of multiple multithreaded tests.