Synthesizing Racy Tests

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

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

Suresh Jagannathan
Purdue University, USA
suresh@cs.purdue.edu

Abstract

Subtle concurrency errors in multithreaded libraries that arise because of incorrect or inadequate synchronization are often difficult to pinpoint precisely using only static techniques. On the other hand, the effectiveness of dynamic race detectors is critically dependent on multithreaded test suites whose execution can be used to identify and trigger races. Usually, such multithreaded tests need to invoke a specific combination of methods with objects involved in the invocations being shared appropriately to expose a race. Without *a priori* knowledge of the race, construction of such tests can be challenging.

In this paper, we present a lightweight and scalable technique for synthesizing precisely these kinds of tests. Given a multithreaded library and a sequential test suite, we describe a fully automated analysis that examines sequential execution traces, and produces as its output a concurrent client program that drives shared objects via library method calls to states conducive for triggering a race. Experimental results on a variety of well-tested Java libraries yield 101 synthesized multithreaded tests in less than four minutes. Analyzing the execution of these tests using an off-the-shelf race detector reveals 187 harmful races, including several previously unreported ones. Our implementation, named NARADA, and the results of our experiments are available at
`http://www.csa.iisc.ernet.in/~sss/tools/narada`.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—testing tools; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—program analysis

General Terms Design, Reliability, Verification

Keywords Race detection; dynamic analysis; concurrency

1. Introduction

Ensuring libraries are *thread-safe* [1, 20] is highly desirable because it would alleviate the burden on applications to deal with the complexities of multithreading. However, building thread-safe libraries is a challenging task [10]. Since libraries are intended to be used by multiple clients concurrently, they must be structured to prevent unwanted racy access to shared state. But, performance considerations dictate minimizing unnecessary synchronization. Thus, if a library is shown not to be thread-safe, applications may be able to take preventive action to avoid the conditions that trigger a thread-safety violation. For example, if it is known that invoking two methods in a library, with certain kinds of parameters, from distinct threads can result in a data race, applications can be (re)written to acquire suitable locks before invoking these methods, without requiring re-implementation of the library.

There have been many program analysis techniques that have been developed to detect races in shared-memory multithreaded programs. These efforts can be broadly classified into three main categories: (a) static analysis techniques (e.g., [15, 21]), (b) systematic testing (e.g., [13, 14]), and (c) dynamic analysis approaches (e.g., [2, 7, 18, 19, 24–26]). While static analysis techniques are execution agnostic and can be comprehensive, results can oftentimes be imprecise [12], with non-trivial false positive rates. On the other hand, systematic testing and dynamic analysis techniques are *critically* dependent on the availability of *effective* multithreaded tests to detect races precisely.

Designing useful multithreaded tests is difficult because if a race is not known to exist *a priori*, the test essentially devolves into a *blind* search for a *potential* race in the library. For a race to manifest, appropriate library methods need to be invoked from at least two different client threads. Moreover, it is also essential that the objects on which the methods operate need to be in a state conducive to trigger the race, and must be shared among these threads appropriately. The requirement to both carefully understand object sharing properties in the library and drive execution to a proper global state, makes the design of effective multithreaded tests for race detection especially challenging.

```
class Lib {
  Counter c;
  public void synchronized update() { c.inc(); }
  public void synchronized set(Counter x) {
    c = x;
  }
}

class Counter {
  int count;
  public void inc() { count++; }
}
```

Figure 1: Illustrative Example.

Figure 1 presents the implementation of two classes `Lib` and `Counter`. Because the method `update` is synchronized, one may mistakenly assume that invoking it from multiple threads without holding any additional lock will not lead to a race. Upon careful inspection, it is clear that this assumption is not true. For example, consider a scenario in which:

1. objects *p* and *q* of type `Lib`, and object *r* of type `Counter` are created,

PLDI'15, June 13–17, 2015, Portland, OR, USA.

http://dx.doi.org/10.1145/2737924.2737998

2. *p*.set(*r*) and *q*.set(*r*) are invoked, and
3. *p*.update() and *q*.update() are invoked from two threads concurrently.

If a multithreaded test performing the aforementioned operations exists, then one of the possible interleavings in the many executions of the test can expose the race on `count`. Therefore, race detection [7, 13] by analyzing the execution of the multithreaded test is also feasible. However, designing such a test requires creating objects of appropriate types, driving each thread to a suitable state so that concurrent execution of the `update` method can expose the underlying race. The goal of this paper is to *automatically* synthesize such multithreaded tests.

We propose a novel and scalable technique for automatically synthesizing racy tests to enable race detection in multithreaded libraries. The input to our technique is the implementation of the library and a sequential seed test-suite. The output is a multithreaded test-suite, where each test spawns multiple threads and invokes various methods in the library appropriately. The key insight to our approach is that the properties observed during sequential execution can be leveraged to generate constraints such that a multithreaded execution satisfying the constraints will result in a race.

Our approach executes the sequential seed test-suite to invoke various methods in the library under test. We analyze the sequential execution traces to identify *unprotected* accesses to fields. If a lock on the object before accessing its contents (fields) is not held, then the corresponding access is considered unprotected. We adopt this conservative approach to maximize the effectiveness of our search for feasible locations that can be manipulated to produce a race. Therefore, even if a lock is held before accessing a field, our definition identifies the potential for a race when the lock objects differ on a shared memory access.

We also identify various methods in the library that modify object state based on the parameters supplied by the sequential tests. For a race to manifest, we need to construct a context where the intersection of the held lock objects for any two shared memory accesses is *empty*. Interestingly, while Eraser [24] uses this property to detect races, we apply the same property to *generate* race inducing tests. We identify a composition of method invocations in the library with appropriate parameters to manifest the necessary context.

Building the necessary context requires the presence of objects upon which library operations can be performed. To do this, as part of the synthesized multi-threaded test, we execute the sequential test multiple times and collect the objects, that are used as parameters (including receivers) for different method invocations, by storing references to them. Subsequently, we use the references to drive the objects to the necessary state as required by the context, spawn multiple threads and invoke appropriate methods from these threads concurrently. The resulting execution can result in a race as the held locks on the shared memory accesses do not share a common lock.

We have incorporated our ideas as part of a tool, named Narada.[1] We have performed a detailed evaluation of Narada on a number of open-source multithreaded Java libraries and components. Our experimental results show that we are able to synthesize a number of multi-threaded tests that expose many races, including previously undetected ones.[2] Analyzing nine classes resulted in the synthesis of 101 multithreaded tests leading to the detection of 307 (187 harmful) races; we are able to synthesize the tests in less than four minutes with negligible memory overhead. These results substantially improve upon recent prior work on detecting thread-safety violations [20].

The paper makes the following technical contributions:

- We develop a framework to synthesize multithreaded tests detecting races in library code by using the implementation of the library under consideration and a sequential seed test-suite as input.
- Our approach analyzes sequential execution traces, identifies unprotected accesses, derives the sequence of method invocations that drives objects to states conducive for triggering a race and reuses existing sequential tests to generate the necessary objects for multithreaded execution.
- We provide details about the implementation of our proposed design and give detailed experimental results to demonstrate the efficacy of our approach.
- We also demonstrate the usefulness of our tool, named Narada, by *seamlessly* integrating it with RaceFuzzer [25] which reports a number of harmful races on multiple benchmarks after analyzing the execution of the tests synthesized by Narada.

```
WriteBehindQueues.java:
----------------------
27 public final class WriteBehindQueues {
44   public static <T> WriteBehindQueue<T>
       createSafeWriteBehindQueue(WriteBehindQueue<T> q) {
45     return new SynchronizedWriteBehindQueue<T>(q);
46   }
47
48   public static WriteBehindQueue
          createCoalescedWriteBehindQueue() {
49     return new CoalescedWriteBehindQueue();
50   }
137 }

SynchronizedWriteBehindQueue.java:
---------------------------------
23  /* Thread safe write behind queue. */
27 class SynchronizedWriteBehindQueue<E> implements
                                 WriteBehindQueue<E> {
29   private final WriteBehindQueue<E> queue;
31   private final Object mutex;
32
33   SynchronizedWriteBehindQueue(WriteBehindQueue<E> q) {
37     this.queue = q;
38     this.mutex = this;
39   }
40
63   public void removeFirst() {
64     synchronized (mutex) {
65       queue.removeFirst();
66     }
67   }
146 }

CoalescedWriteBehindQueue.java:
------------------------------
17 class CoalescedWriteBehindQueue implements
                         WriteBehindQueue<DelayedEntry> {
19   protected final Map<Data, DelayedEntry> queue;

57   public void removeFirst() {
58     final Set<Data> keySet = queue.keySet();
59     for (Data key : keySet) {
60       queue.remove(key);
61       break;
62     }
63   }
158 }
```

Figure 2: Motivating example.

[1] The name of an Indian sage renowned for setting up conflicts to address a greater good.

[2] `https://github.com/hazelcast/hazelcast/issues/4039`

2. Motivation

We motivate the problem addressed in the paper by using a real example from `hazelcast`[3], a popular open source in-memory data grid, that is under active development with 12K commits, 68 releases and 82 contributors. According to the documentation, the APIs are used to improve performance of applications, to distribute data across servers, clusters and geographies and to manage very large data sets or very high data ingest rates.

Figure 2 presents partial implementations of three classes from `hazelcast-3.3.2`. Two method implementations from `WriteBehindQueues`, a class that provides static factory methods which create write behind queues, are shown. `SynchronizedWriteBehindQueue` is a *supposedly* thread-safe class based on the comment on line `23`. The implementation of the constructor and a method `removeFirst` is shown in the figure. Finally, the partial implementation of class `CoalescedWriteBehindQueue`, one of the `WriteBehindQueue` classes, is shown in the figure where there is no synchronization performed in the implementation of method `removeFirst`.

```
public void exposeRace() {
   WriteBehindQueue cwbq =
     WriteBehindQueues.createCoalescedWriteBehindQueue();

   WriteBehindQueue<DelayedEntry> swbq1 =
     WriteBehindQueues.createSafeWriteBehindQueue(cwbq);
   WriteBehindQueue<DelayedEntry> swbq2 =
     WriteBehindQueues.createSafeWriteBehindQueue(cwbq);
                   ...
   Thread t1 = new Thread() {
       void run() { swbq1.removeFirst(); }
   }
   Thread t2 = new Thread() {
       void run() { swbq2.removeFirst(); }
   }
}
```

Figure 3: Racing test.

Based on our analysis, we claim that clients using the library can potentially have races depending upon the invoked methods and the objects on which the methods are invoked. More specifically, executing the multithreaded program shown in Figure 3 can expose a race in the class `SynchronizedWriteBehindQueue`. This is because the two objects `swbq1` and `swbq2` wrap one object `cwbq` as shown in Figure 4. Subsequently, from two threads, `removeFirst` is invoked where the state of `cwbq` is updated. However, the updates are performed while holding locks on `swbq1` and `swbq2` respectively leading to a race. *Ideally*, the update on `cwbq` should have been performed while holding a lock on `cwbq`. The developers of the library incorrectly assign the `this` object as the `mutex` instead of the `queue` object in line `38` in `SynchronizedWriteBehindQueue.java` leading to this problem.

Designing this racy test manually is a non-trivial task. It requires a nuanced understanding of the implementation of the three classes, identification of the shared memory access and the associated lock object correlations, a specific invocation order of methods (in this case, constructors) with appropriate parameters to setup the necessary context, and then an invocation of relevant methods from distinct threads to cause a race. The nesting of the invocations to manifest the race makes the task more challenging. For example, in the above scenario, the race happens when `removeFirst` invokes `remove` which updates its internal state.

[3] `http://www.hazelcast.org`

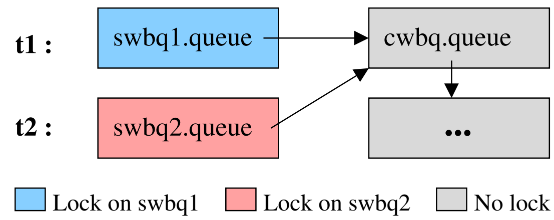

Figure 4: Concurrent updates to `cwbq.queue`.

```
public void test() {
   WriteBehindQueue cwbq =
     WriteBehindQueues.createCoalescedWriteBehindQueue();

   WriteBehindQueue<DelayedEntry> swbq =
     WriteBehindQueues.createSafeWriteBehindQueue(cwbq);
             ...
   swbq.removeFirst();
}
```

Figure 5: Sequential seed test.

Our analysis is able to automatically synthesize the racy test shown in Figure 3. The input to our analysis is the implementation of the library along with a sequential seed test shown in Figure 5. Apart from the racy test shown in Figure 3, we are able to synthesize many racy tests that expose multiple races in `SynchronizedWriteBehindQueue`. Even though it may appear that objects will not be intentionally shared as described above, when the library is used as a component in a larger application, the flow of parameters can *unintentionally* result in such sharing. Our analysis can help avoid such scenarios by synthesizing tests to detect potential races when libraries are used.

3. Design

Our analysis is broadly divided into three stages. The first analyzes execution traces derived from executing sequential tests to (a) identify *unprotected* accesses (Section 3.1), and (b) identify various ways to modify library-controlled objects from a client (Section 3.2). The second stage uses this information to build constraints pertaining to library methods, along with appropriate parameters, that need to be invoked to trigger a race (Section 3.3). The third stage synthesizes racy test which executes sequential tests to build object state that can be used to enforce these constraints (Section 3.4). Analyzing the execution of the racy test with suitable dynamic analysis engines can expose underlying races.

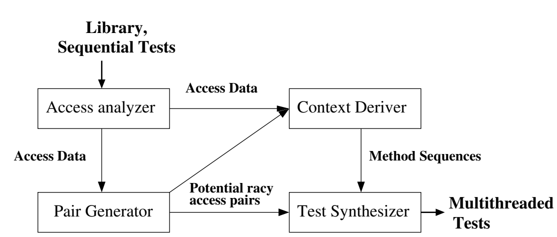

Figure 6: Overview of test synthesis.

Figure 6 presents a design overview of our analysis. The first stage of the analysis is accomplished by the Access Analyzer, the second stage is performed by a combination of Pair Generator and Context Deriver, and Test Synthesizer addresses the third stage of the analysis. We now describe each stage in detail.

$$\mathcal{H} \in \mathcal{P}((Var + Var \times \overline{Field}) \mapsto (Loc \times \{C, NC\} \times \{L, U\})) \qquad \mathcal{A} \in \mathcal{P}(Label \mapsto (Writeable \times Unprotected))$$

$$\frac{\mathcal{H} = \mathcal{R}(\mathcal{S}, \ell, C, U, x_0, x_1, \ldots, x_k) \quad \ell \in Client}{\langle x_0.\mathtt{m}(x_1, x_2, \ldots, x_k)^{\ell}.\overline{T}, \phi, \phi\rangle \rightarrow \langle \overline{T}, \mathcal{H}, \phi\rangle} \text{ (INVOKE)} \qquad \frac{\mathcal{H}' = \mathsf{bind}(\mathcal{H}, x, y)}{\langle (x := y)^{\ell}.\overline{T}, \mathcal{H}, \mathcal{A}\rangle \rightarrow \langle \overline{T}, \mathcal{H}', \mathcal{A}\rangle} \text{ (ASSIGN)}$$

$$\frac{\mathcal{H}' = \mathcal{H} \circ \mathcal{R}(\mathcal{S}, \ell, NC, U, x)}{\langle (x := \mathsf{alloc})^{\ell}.\overline{T}, \mathcal{H}, \mathcal{A}\rangle \rightarrow \langle \overline{T}, \mathcal{H}', \mathcal{A}\rangle} \text{ (ALLOC)} \qquad \frac{z \in \{x.\mathtt{f}_1, x.\mathtt{f}_1.\mathtt{f}_2, \ldots\} = \mathcal{N}(x) \quad \mathcal{H}(z, l', C, \_) = true \quad \mathcal{A}' = \mathcal{A}[\ell \mapsto (true, false)]}{\langle (\mathsf{return}(x))^{\ell}.\overline{T}, \mathcal{H}, \mathcal{A}\rangle \rightarrow \langle \overline{T}, \mathcal{H}, \mathcal{A}'\rangle} \text{ (RETURN)}$$

$$\frac{\begin{array}{c}\mathcal{H}' = \mathsf{bind}(\mathcal{H}, x, y.\mathtt{f}) \\ \mathcal{A}' = \mathcal{A}[\ell \mapsto (false, \mathcal{H}(y, l, C, U))]\end{array}}{\langle (x := y.\mathtt{f})^{\ell}.\overline{T}, \mathcal{H}, \mathcal{A}\rangle \rightarrow \langle \overline{T}, \mathcal{H}', \mathcal{A}'\rangle} \text{ (READ)} \qquad \frac{\begin{array}{c}\mathcal{H}' = \mathsf{bind}(\mathcal{H}, \mathsf{alias}(\mathcal{H}, x) \oplus \{\mathtt{f}\}, y) \\ \mathcal{A}' = \mathcal{A}[\ell \mapsto (\mathcal{H}(x, l, C, \_) \wedge \mathcal{H}(y, l, C, \_), \mathcal{H}(x, l, C, U))]\end{array}}{\langle (x.\mathtt{f} := y)^{\ell}.\overline{T}, \mathcal{H}, \mathcal{A}\rangle \rightarrow \langle \overline{T}, \mathcal{H}', \mathcal{A}'\rangle} \text{ (WRITE)}$$

$$\frac{\mathcal{H}' = \mathsf{lbind}(\mathcal{H}, \mathsf{alias}(\mathcal{H}, x), L)}{\langle \mathsf{lock}(x)^{\ell}.\overline{T}, \mathcal{H}, \mathcal{A}\rangle \rightarrow \langle \overline{T}, \mathcal{H}', \mathcal{A}\rangle} \text{ (LOCK)} \qquad \frac{\mathcal{H}' = \mathsf{lbind}(\mathcal{H}, \mathsf{alias}(\mathcal{H}, x), U)}{\langle \mathsf{unlock}(x)^{\ell}.\overline{T}, \mathcal{H}, \mathcal{A}\rangle \rightarrow \langle \overline{T}, \mathcal{H}', \mathcal{A}\rangle} \text{ (UNLOCK)}$$

Figure 7: Inference rules.

3.1 Analysis of Sequential Execution Traces

Our analysis operates over *traces*, a sequence of expressions that comprise the execution of a sequential test; these expressions may consist of code from the client as well as libraries. For the purposes of our discussion, traces are built from the following terms:

$$\begin{array}{rcl} e \in Exp & := & x \mid y \mid x.\mathtt{f} \mid \mathsf{alloc} \mid x.\mathtt{m}(x_1, \ldots, x_n) \\ stmt & ::= & x := e \mid x.\mathtt{f} := y \mid \mathsf{lock}(x) \mid \mathsf{unlock}(x) \mid \mathsf{return}(x) \end{array}$$

Expressions include variables (x, y, etc.), field accesses ($x.\mathtt{f}$), object allocations (alloc) and library method calls. Statements are assignments to variables, which may represent objects, and fields of objects, as well as lock, unlock and return statements.

Each element in a trace has a unique label that serves as its dynamic execution index [11, 29]. Given a trace, the first step towards synthesizing a racy test is to analyze it, constructing an abstract structure of the heap and the accesses made by each trace element. Beyond recording basic points-to and aliasing information, these abstractions also maintain information about whether an access to a shared variable is *unprotected* or *writeable*. These concepts, in turn, critically rely on the notion of *controllability* - intuitively, a controllable variable references an object maintained by the library that can be manipulated by the client through library methods. For example, if a field of a library-manipulated object is set to the value passed as a parameter by a client, then the access of the field is considered controllable until the field is subsequently updated to reference an object that is not influenced by any of the method's parameters (e.g., an object allocated locally).

Having information about controllable fields enables the analysis to setup the necessary context for race synthesis. Now, if the access to an object is performed without holding a relevant lock, and the variable being used to perform the access is controllable, we consider that access unprotected, and this access can potentially be influenced in a synthesized test case to construct a race. Second, if both sides of an assignment are controllable, and the access involves a write to a field, then that access is considered writeable; such accesses can also be used to drive execution of a library method to a state that can lead to a race.

Our analysis maintains two structures - $\mathcal{H}$ and $\mathcal{A}$ to maintain this information. $\mathcal{H}$ is a per-trace element abstraction of the program heap, recording the location to which a variable or field (more precisely, sequence of fields rooted at a variable) within an object is bound, whether that variable (or field) is controllable, and whether it is locked or unlocked at any specific point in the execution. In our rules, we overload $\mathcal{H}$'s definition so that $\mathcal{H}(x, l, Cont, Lock)$ is true if variable x points to a location l where $Cont$ is one of C (controllable) or NC (*not* controllable), and $Lock$ is one of L (locked) or U (unlocked). $\mathcal{A}$ is a projection of the heap that records information about accesses, collecting for each point in the trace, writeable and unprotected information that can be used to determine whether the recorded accesses can be involved in a race in a multi-threaded execution. It effectively summarizes heap state relevant for each point in the trace.

Figure 7 defines the rules that evaluate a trace to produce these abstractions. The evaluation relation ($\rightarrow$) operates over a triple consisting of a trace $\overline{T}$ whose head element is e, and whose current heap abstraction is $\mathcal{H}$, and whose access projection is $\mathcal{A}$; each rule determines how to evaluate e to produce a new heap and a new projection.

The INVOKE rule for a library method invocation from a client[4] assumes the existence of a bootstrapping function, $\mathcal{R}$. This function takes as input the source program, $\mathcal{S}$, the trace element index ℓ, a controllable flag, a lock flag and a set of variables. It initializes the heap $\mathcal{H}$ by allocating new heap locations, and binding variables and fields accessible from the objects denoted by these variables, based on the variable's static type, with these flags. In the antecedent of this rule, since the invocation is initiated by the client and as we ignore the lock acquisitions within the client body, the heap established by $\mathcal{R}$ initializes the receiver object as well as its arguments in the invocation to be controllable and unlocked.

The operator bind in rule ASSIGN (whose definition is straightforward and elided here) performs a *deep* walk over the heap, resetting aliasing properties of its arguments; two variables and/or fields are aliased if they map to the same location. Thus, after an assignment, ($x := y$), the heap abstraction would identify x to be aliased with y (they would both map to the same heap node, say l), $x.\mathtt{f}$ to be aliased with $y.\mathtt{f}$ where f is a field in the object referenced by x and y as determined by their static type, and so forth. Note that $\mathcal{A}$ is left unmodified in the rule. This is because an assignment of this form does not modify the value of any field referenced by x or y; while these assignments can be leveraged by subsequent phases of the analysis to break aliasing relationships, they cannot be used to directly induce a race, which arise from storing into and reading from object fields.

When variable x is reassigned to point-to a newly allocated object, its prior aliasing relationship with other variables is broken, and an aliasing relationship with the new object is established ($\mathcal{H}'$). Furthermore, we initialize the controllablity and locking properties of this object using the bootstrapping function, $\mathcal{R}$, as described

[4] This is the only operation presumed to be executed by the client; all other rules assume the operation being considered occurs within a library method.

previously; the expression $\mathcal{H} \circ \mathcal{R}(\ldots)$ in the antecedent of the rule ALLOC defines heap concatenation between $\mathcal{H}$ and the heap returned by $\mathcal{R}$. Because the newly created object is allocated within a library method, its controllability flag is set to *NC*. Moreover, when a variable is assigned to a newly allocated object, there are no changes to the access projection because allocation does not involve writing to an object, only reassigning a reference.

Rule RETURN deals with values returned by a method that may be influenced by the method's parameters. This rule is applicable only on return to the client. Even though an object returned from a library method may have been allocated locally, it could have been assigned (or have one of its fields assigned) to a method parameter. In this case, we mark the return label as being associated with a writeable action, indicating that test synthesis can manipulate this method, or any other library method that takes arguments of the return type, via client actions. The auxiliary operator $\mathcal{N}$ returns the set of field access names for the returned object based on its static type. For example, in the following:

```
void foo (z) {
 x := alloc(); y := z; w := x; w.f := y; return w;
}
```

even though the return value `w` is aliased to `x`, a variable bound to locally-allocated object, the alias of parameter `z` (here, `y`), is assigned to one of `w`'s fields. Thus, from this return value, we know that its field `f` is the parameter passed to `foo`, thus providing a mechanism to influence any other method that requires an object with that state. To record this ability, the rule marks the return statement as writeable in $\mathcal{A}$.

Rule READ establishes alias relationships between x and y.`f` (and recursively from fields found in objects they reference through the use of bind); moreover, the access map is updated to reflect the fact that at this access (at label ℓ in the trace) no field is being written (there is only a read of y.`f`), and its access is unprotected only if y is controllable, and unlocked.

Rule WRITE is more complicated. Apart from performing a deep walk of the heap with x.`f` and y as parameters to re-establish aliasing relationships, it is imperative that all aliases of x are extracted (via operator alias) and a field dereference `f` from each one of them is also aliased to y. We overload the definition of bind so that the expression:

$$\mathsf{bind}(\mathcal{H}, \mathsf{alias}(\mathcal{H}, x) \oplus \{\mathtt{f}\}, y)$$

establishes aliasing relationships between *every* alias of x.`f` and y in heap $\mathcal{H}$; operator $\oplus$ concatenates its second argument to each element in the set produced by its first. For example, if z.`h` is an alias of x, then x.`f` and z.`h.f` need to be aliased to y. Furthermore, because a write to x happens, we check whether the right-hand side of the assignment (y) and target (x) are controllable from the client. If so, the write access becomes writeable. The detection of unprotectedness is performed as before.

LOCK and UNLOCK rules find all aliases of the parameter x and update their locking state to L and U respectively using the lbind operator that updates the heap state accordingly.

3.1.1 Example

We now explain the application of the inference rules with an illustrative example. Figure 8 presents the implementation of method `foo` in class `A`, and the trace obtained by executing the method. Interesting changes to $\mathcal{H}$ after each label are shown in Table 1. When the method is invoked from a client, the bootstrapping of various symbols is done (using $\mathcal{R}$) and the resulting output is shown after label 1. Subsequently, a lock on `this` updates the locking state of the appropriate heap element. When `this` is assigned to `b`, the two symbols become aliases. Furthermore, as mentioned in the rule for `x := y` (based on bind's deep walk over the heap), a.`x` and b.`x` are also aliased. The remaining transitions in Table 1 are derived similarly.

```
class A {
  void foo(Y y) {
    synchronized(this) {
      A b = this;
      X t = b.x;
      t.o = rand();
      b.y = y;
    }
  }
}
```

(a) Source.

```
1  a.foo(y)
2  lock(this)
3  b := this
4  t := b.x
5  t.o := rand();
6  b.y := y
7  unlock(this)
```

(b) Trace.

Figure 8: Illustrative example.

Table 1: Interesting changes in aliasing relationships captured by $\mathcal{H}$ at the end of each label from Figure 8. For clarity, we omit including locations and do not repeat unchanged heap elements across execution steps.

1	`a` $\mapsto (C,U)$, `a.x` $\mapsto (C,U)$, `a.y` $\mapsto (C,U)$, `a.x.o` $\mapsto (C,U)$, `this` $\mapsto (C,U)$, `this.x` $\mapsto (C,U)$, `this.y` $\mapsto (C,U)$, `this.x.o` $\mapsto (C,U)$, `y` $\mapsto (C,U)$
2	`a` $\mapsto (C,L)$, `this` $\mapsto (C,L)$
3	`b` $\mapsto (C,L)$, `b.x` $\mapsto (C,U)$, `b.y` $\mapsto (C,U)$, `b.x.o` $\mapsto (C,U)$
4	`t` $\mapsto (C,U)$, `t.o` $\mapsto (C,U)$
5	`a.x.o` $\mapsto (NC,U)$, `b.x.o` $\mapsto (NC,U)$, `t.o` $\mapsto (NC,U)$ `this.x.o` $\mapsto (NC,U)$
6	`a.y` $\mapsto (C,U)$, `b.y` $\mapsto (C,U)$, `this.y` $\mapsto (C,U)$
7	`a` $\mapsto (C,U)$, `this` $\mapsto (C,U)$, `b` $\mapsto (C,U)$

Apart from $\mathcal{H}$, we also maintain $\mathcal{A}$ to specify whether an access at a particular point in the trace is writeable and/or unprotected. Upon applying the inference rules, at the end of the method, we observe that:

$$\mathcal{A} : \{4 \mapsto (\mathtt{false}, \mathtt{false}), 5 \mapsto (\mathtt{false}, \mathtt{true}), 6 \mapsto (\mathtt{true}, \mathtt{false})\}$$

The read access at label 4 is neither writeable, since it is a read, nor unprotected because `b` is locked (L); thus, this access is not manipulable from any client-driven testcase. Similarly, the write access at label 5 is not *writeable* because the right-hand side of the assignment is not controllable (the `rand` function returns a random object whose contents cannot be controlled by a client). On the other hand, the write access at label 6 is *writeable* because the right-hand side of the assignment (`y`) and target (`b`) are controllable (C) (see $\mathcal{H}$ after label 5). The access at label 5 is *unprotected* because `t` is unlocked (U). In contrast, the access at label 6 is protected because `b` is locked (L).

$\mathcal{A}$ is leveraged by the subsequent stages of our analysis. For example, the derivation that the access at `5` is *unprotected* suggests the construction of a test which can exploit the access to expose a race. For example, if two threads invoke `foo` with different objects a_1 and a_2 as the receiver respectively, in a context where a_1.`x` and a_2.`x` point to the same object, then a race manifests. However, for the fields to point to the same object, it is necessary to set the context appropriately. This is obtained by analyzing the *writeable* bit in $\mathcal{A}$.

While the above analysis identifies the accesses that are *writeable* and/or *unprotected*, to fully automate racy test synthesis, we still need information connecting the associated accesses to the objects that are accessible from the client; this information includes:

- the receivers on library method invocations,
- parameters passed to the invocations, or
- return values from the invocations.

We now describe the process of deriving this data.

$$Arg : Symbol + (Symbol \times \overline{Field}) \qquad\qquad \mathcal{D} \in \mathcal{P}(Label \mapsto \mathcal{P}(Arg \leftsquigarrow Arg))$$

$$\frac{\mathcal{H}' = \mathsf{bind}(\mathcal{H}, x, y.\mathtt{f}) \qquad \mathcal{A}' = \mathcal{A}[\ell \mapsto (\mathit{false}, \mathcal{H}(y, l, C, U))] \qquad \boxed{\mathcal{D}' = \mathcal{D}[\ell \mapsto \{\mathsf{src}(\mathsf{x}, \mathcal{H}) \leftsquigarrow \mathsf{src}(\mathsf{y}, \mathcal{H}) \oplus \mathtt{f}\}]}}{\langle (x := y.\mathtt{f})^{\ell}.\overline{T}, \mathcal{H}, \mathcal{A}, \mathcal{D}\rangle \to \langle \overline{T}, \mathcal{H}', \mathcal{A}', \mathcal{D}'\rangle} \ (\textsc{read})$$

$$\frac{\mathcal{H}' = \mathsf{bind}(\mathcal{H}, \mathsf{alias}(\mathcal{H}, x) \oplus \{\mathtt{f}\}, y) \qquad \mathcal{A}' = \mathcal{A}[\ell \mapsto (\mathcal{H}(x, l, C, \_) \wedge \mathcal{H}(y, l, C, \_), \mathcal{H}(x, l, C, U))] \qquad \boxed{\mathcal{D}' = \mathcal{D}[\ell \mapsto \{\mathsf{src}(\mathsf{x}, \mathcal{H}) \oplus \mathtt{f} \leftsquigarrow \mathsf{src}(\mathsf{y}, \mathcal{H})\}]}}{\langle (x.\mathtt{f} := y)^{\ell}.\overline{T}, \mathcal{H}, \mathcal{A}, \mathcal{D}\rangle \to \langle \overline{T}, \mathcal{H}', \mathcal{A}', \mathcal{D}'\rangle} \ (\textsc{write})$$

$$\frac{z \in \{x.\mathtt{f}_1, x.\mathtt{f}_1.\mathtt{f}_2, \ldots\} = \mathcal{N}(x) \quad \mathcal{H}(z, (l', C, \_)) = \mathit{true} \quad \mathcal{A}' = \mathcal{A}[\ell \mapsto (\mathit{true}, \mathit{false})] \quad \boxed{\mathcal{D}' = \mathcal{D} \circ \mathsf{update}(\mathcal{H}, \ell, \mathcal{N}(x))}}{\langle (\mathsf{return}(x)^{\ell}.\overline{T}, \mathcal{H}, \mathcal{A}, \mathcal{D}\rangle \to \langle \overline{T}, \mathcal{H}, \mathcal{A}', \mathcal{D}'\rangle} \ (\textsc{return})$$

Figure 9: Modified inference rules.

3.2 Connecting Client Objects to Interesting Accesses

$\mathcal{A}$ determines whether an access at a label ℓ is writeable and/or unprotected. However, it does not provide the necessary information to identify the client object on which the access happens. For example, in Figure 8, we know from $\mathcal{A}$ that the access at 5 is unprotected and the access at 6 is writeable. However, we need to additionally identify the client object that corresponds to (i.e., is aliased with) the components comprising these accesses. In other words, if the client invokes a_1.foo(b_1), we need to identify that a_1.x is an unprotected write at 5 and a_1.y is the write at 6. This aliasing information between client-supplied objects and the accesses that read or write them also needs to be established.

To be able to do this effectively, we need additional variables to identify the *source* of an access precisely. For example, in the following:

```
void foo (z) {
  y := z; z := alloc; x := y.f;
}
```

when the read of y.f occurs, the object referenced by y is the first parameter to foo and not z since z is re-allocated between y's initial assignment to z and the dereference of $y.f$. To ensure that we are able to track dataflow of client objects within library methods precisely, we introduce *additional* (local) variables to record parameter values. We rewrite each method with these variables such that each parameter and the receiver is assigned to a new variable (I_i) at the beginning of the method. The inference rules given in Figure 7 are applicable to these assignments, only if the method is directly invoked from the client. Moreover, for subsequent trace elements that are part of the original program, the heap locations for $\mathtt{I}_i$s (and its deep dereferences) in $\mathcal{H}$ will never need to be modified. The above example is rewritten as follows:

```
void foo (z) {
 Ithis := this; Iz := z; y := z; z := alloc; x := y.f;
}
```

Here, when the ASSIGN rule from Figure 7 is applied on the assignment $\mathtt{I}_z$:= z, then $\mathtt{I}_z$ is aliased with z, $\mathtt{I}_z$.f is aliased with z.f and so forth. Subsequently, when z is allocated, it points to a new location in $\mathcal{H}$. However, $\mathtt{I}_z$ points to the old location even though it aliases with z initially. Therefore, when y.f is considered, we can identify that y is an alias of $\mathtt{I}_z$ and realize that it is the parameter passed to the method.

The operator $\mathsf{src}(x, \mathcal{H})$ returns the additional variable (or its dereference) that aliases with x in $\mathcal{H}$.

$$\mathsf{src}(x, \mathcal{H}) = \begin{cases} \mathtt{I}_i.* & \text{if } \exists i,\ \mathtt{I}_i.* \in \mathsf{alias}(x, \mathcal{H}) \\ \perp & \text{otherwise} \end{cases}$$

For example, at the end of the method body for the above example, we get $\mathsf{src}(\mathsf{y}, \mathcal{H}) = \mathtt{I}_z$, $\mathsf{src}(\mathsf{x}, \mathcal{H}) = \mathtt{I}_z.\mathtt{f}$ and $\mathsf{src}(\mathsf{z}, \mathcal{H}) = \perp$ (where $\perp$ represents an uninteresting value).

To achieve our goal of identifying useful client objects, we modify the inference rules as shown in Figure 9. We maintain another structure $\mathcal{D}$ that records access summaries at a label which may involve zero or more accesses. The operator ($\leftsquigarrow$) defines a relation between the value of its right-hand side and its left-hand side. The definitions of $\oplus$ and $\circ$ are as before. The evaluation relation is now redefined to operate over a quadruple with the triple as defined before, and $\mathcal{D}$. Rule READ updates $\mathcal{D}$ to denote that the first element of the pair is the additional synthesized variable (or one of its deep dereferences) that aliases with x and the second element is the field f of some synthesized variable (or its deep dereference) that aliases with y. A similar action is performed for the WRITE rule.

The RETURN rule is more involved. We define an update operator that creates a special variable $\mathtt{I}_r$ that is associated with the return variable x. For each element z in $\mathcal{N}(\mathtt{x})$, if $\mathtt{x}.\overline{\mathtt{f}}$ is *controllable*, then the associated state in x becomes writeable. Therefore, we identify the influencing parameter (or receiver) and add an appropriate mapping

$$\ell \mapsto (\mathtt{I}_r.\overline{\mathtt{f}} \leftsquigarrow \mathsf{src}(\mathtt{x}.\overline{\mathtt{f}}))$$

to $\mathcal{D}$. Because, this is performed for each z satisfying the above criteria, there can be multiple assignments at the label corresponding to return. Essentially, this means that even if the object being returned was allocated within the library, if one (or more) of its fields are updated within the library with parameters passed from the client, that information can be collected to enable race synthesis. Consider the following code snippet:

```
foo(x,y) {
  x.f := y; w := alloc; w.z := x; return w;
}
```

The set, $\{\mathtt{I}_r.\mathtt{z}.\mathtt{f} \leftsquigarrow \mathtt{I}_y, \mathtt{I}_r.\mathtt{z} \leftsquigarrow \mathtt{I}_x\}$, is the access summary at the return label. It denotes that a client can invoke the method foo to obtain an object whose fields are determined by the parameters passed by the client. This enables the client to drive the object to a chosen state.

For the example given in Figure 8, introduction of the additional variables results in the code and trace as shown in Figure 11. The generated $\mathcal{A}$ and $\mathcal{D}$ structures are given below:

$\mathcal{A}$: $\{4 \mapsto (\mathit{false}, \mathit{false}), 5 \mapsto (\mathit{false}, \mathit{true}), 6 \mapsto (\mathit{true}, \mathit{false})\}$
$\mathcal{D}$: $\{4 \mapsto \{\perp \leftsquigarrow \mathtt{I}_1.\mathtt{x}\}, 5 \mapsto \{\mathtt{I}_1.\mathtt{x}.\mathtt{o} \leftsquigarrow \perp\}, 6 \mapsto \{\mathtt{I}_1.\mathtt{y} \leftsquigarrow \mathtt{I}_2\}\}$

The binding at label 6 in $\mathcal{D}$ indicates that the parameter affected by the left-hand side of the assignment ($\mathtt{I}_1$) is the receiver object

$$\frac{e^{\ell} \in m \qquad (\mathtt{I}_i.\mathtt{f} \leftsquigarrow \mathtt{I}_j) \in \mathcal{D}(\ell) \quad \mathsf{type}(\mathtt{I}_x) = \mathsf{type}(\mathtt{I}_i)}{\mathcal{Q}(\mathtt{I}_x.\mathtt{f}) = m} \ (\textsc{set}) \qquad \frac{\mathcal{Q}(\mathtt{I}_x.\mathtt{f}) = m_1 \quad \mathcal{Q}(o.\mathsf{g}) = m_2 \quad \mathsf{type}(o) = \mathsf{type}(\mathtt{f})}{\mathcal{Q}(\mathtt{I}_x.\mathtt{f}.\mathsf{g}) = m_2.m_1} \ (\textsc{concat})$$

$$\frac{\ell \in client \quad \ell'' \in m \quad (\mathtt{I}_i.\mathtt{f}.\mathsf{g} \leftsquigarrow \mathtt{I}_j) \in \mathcal{D}(\ell'') \quad \mathsf{type}(\mathtt{I}_x) = \mathsf{type}(\mathtt{I}_i) \qquad \langle e^{\ell}.\overline{T}, \_, \_, \mathcal{D}\rangle \xrightarrow{*} S_i \xrightarrow{*} \langle e^{\ell''}.\overline{T}'', \_, \_, \mathcal{D}''\rangle \wedge S_i \neq \langle e^{\ell'}.\overline{T}', \_, \_, (\mathtt{I}_i.\mathtt{f} \leftsquigarrow \_)\rangle}{\mathcal{Q}(\mathtt{I}_x.\mathtt{f}.\mathsf{g}) = m} \ (\textsc{deep-set})$$

Figure 10: Deriving method sequences.

```
class A {
  void foo(Y y) {
    I1 = this; I2 = y;
    synchronized(this) {
      A b = this;
      X t = b.x;
      t.o = rand();
      b.y = y;
    }
  }
}
```

(a) Source.

```
1     a.foo(y)
1'    I1 := this
1"    I2 := y
2     lock(this)
3     b := this
4     t := b.x
5     t.o := rand();
6     b.y := y
7     unlock(this)
```

(b) Trace.

Figure 11: Example after introducing additional variables.

(a) and that parameter aliased with the expression on the right-hand side of the assignment is the supplied argument (y), denoted by $\mathtt{I}_2$. The corresponding $\mathcal{A}$ also shows that this is a writeable access. For non-writeable accesses, either the LHS or RHS of the access summary $\mathcal{D}$ at a label is $\perp$ (labels 4, 5). Nevertheless, the unprotected access at these labels can still be derived from $\mathcal{D}$. For example, the unprotected access at label 5 is $\mathtt{I}_1.\mathtt{x}.\mathtt{o}$.

3.3 Setting the Necessary Context

We use the information describing unprotected accesses to construct racy tests. An unprotected access at a label ℓ in one thread can race with

- a concurrent access at ℓ from a different thread,
- an (un)protected access on the same object at some ℓ' in a different thread.

For each *unprotected* access detected by our analysis, we generate multiple racing pairs as described above. However, the key criterion for a race to manifest is that the object instances under consideration are the same. In general, if the racing pairs are $I_x.f_1.f_2 \ldots f_k.f$ and $I_y.f'_1.f'_2 \ldots f'_{k'}.f'$, where f and f' have the same type, then the owner of f and f' need to point to the same object instance. In other words, $I_x.f_1.f_2 \ldots f_k$ and $I_y.f'_1.f'_2 \ldots f'_{k'}$ need to point to the same object instance. Therefore, initially, we need to design a mechanism to set the context such that object instances are shared appropriately.

In the example shown in Figure 11, we derive that the write access at label 5 is unprotected. If we pair the unprotected access ($\mathtt{I}_1.\mathtt{x}.\mathtt{o}$ from $\mathcal{D}$) with the access at the same label from a different thread, we need to set a context such that a race on a.x can manifest. This can happen only if $\mathtt{I}_1.\mathtt{x}$ from the two threads reference the same object. In other words, the fields x of the receivers of foo need to reference the same object. *Importantly*, there is no need to strengthen this constraint further, for example by requiring that the receivers be the same object, because imposing these additional constraints can potentially disable race detection. For instance, if the receivers (represented by I_1) are the same for foo, then the race cannot manifest because of the lock acquisition on the receivers. Therefore, we need two *different* object instances as receivers for foo, while requiring that the field x of both these receivers nonetheless refers to the same object. This translates to identifying method(s) that assign an object passed by the client to the field x of object of type A. The client can then pass the same object to the method(s) to set the field x for two different objects of type A.

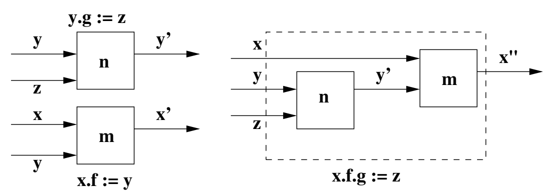

Figure 12: Illustrative example for updating object state.

We observe that the overall problem of setting a context can be reduced to the fundamental problem of making an assignment to $x.f_1.f_2 \ldots f$ with an object specified by the client. There may not be one method that takes an object as parameter and assigns it to $x.f_1.f_2 \ldots f$. A sequence of method invocations may be necessary to accomplish the required assignment. For example, as shown in Figure 12, method n may assign $y.g$, m may assign $x.f$ where y and $x.f$ have the same types. Sequentially invoking the methods n and m with the appropriate parameters can modify the state of object x by assigning an object that is available from the client to $x.f.g$ even though there is no single setter method to update the associated state. This can be extended to handle the assignment for $x.f_1.f_2 \ldots f$.

$\mathcal{D}$ plays a significant role in deciding the method(s) that need to be invoked. We use it to derive the required sequence of method invocations as shown by the rules in Figure 10. $\mathcal{Q}$ is a query operator that takes the field dereference under consideration and outputs either a method or a sequence of methods. The SET rule identifies the method m where $\mathtt{I}_j$ can be assigned to the field f of $\mathtt{I}_i$ by a client, such that the type of $\mathtt{I}_i$ and $\mathtt{I}_x$ are equivalent. The CONCAT rule essentially identifies a sequence of two methods m_1 and m_2 such that the first method assigns $\mathtt{I}_x.\mathtt{f}$ and the second method assigns $o.\mathsf{g}$ where the types of o and f match. The rule DEEP-SET identifies a method in which $\mathtt{I}_x.\mathtt{f}.\mathsf{g}$ is assigned. If there is no re-assignment to $\mathtt{I}_i.f$ until $\mathtt{I}_i.\mathtt{f}.\mathsf{g}$ is assigned in m, then the method m is considered to make an assignment to $\mathtt{I}_x.\mathtt{f}.\mathsf{g}$.

In the above description, the source of the assignment is always the object that is passed as a parameter. However, in practice, the source of the assignment can be a field of the object that is passed as a parameter. For example, in a trace $z := y.g; x.f := z$, where y is a parameter passed to a method, $x.f$ is associated with $\mathtt{I}_y.\mathsf{g}$. To handle such a scenario, we need to repetitively apply the aforementioned rules on $\mathtt{I}_y.\mathsf{g}$.

We now illustrate setting the context for the example given in Figure 8 by adding method bar to class A and adding class Z as shown in Figure 13. Assume that the methods bar and baz are executed as part of some sequential test(s). Based on our analysis, analyzing the execution trace of bar will detect the presence of a *writeable* assignment to A.x, (i.e) the corresponding $\mathcal{D}$ will have ($\mathtt{I}_{this}.\mathtt{x} \leftsquigarrow \mathtt{I}_z.\mathtt{w}$). Similarly, our analysis detects a writeable assignment for Z.w in baz.

```
class A {
  X x; Y y;
  void foo(Y y) {
   synchronized(this) {
     A b = this;
     X t = b.x;
     t.o = rand(); // unprotected access of this.x
     b.y = y;      // protected access of this
   }
  }
  void bar(Z z) {
    this.x = z.w;  // sets field x of A
  }
}
class Z {
  X w = null;
  void baz(X x) {
    this.w = x;    // sets field w of Z
  }
}
```

Figure 13: Illustrative example for setting context.

For a race to manifest on the field `x` in class `A` when method `foo` is invoked by two threads, it is essential that the field `x` across the two invocations point to the same object. In other words, when a.`foo` and a'.`foo` are invoked by two different threads, then a.`x` and a'.`x` need to refer to the same object. Trivially, this is possible by ensuring a and a' refer to the same object. However, this will not help in manifesting the race as a lock is acquired on the receivers (a and a'). Thus, we also need to ensure a and a' are distinct. Initially, we can consider distinct a and a' and identify the possibility of ensuring that their field `x` can be set appropriately. We can achieve this by invoking method `bar` on a and a' which will help set the field `x`. Since the right-hand side of the assignment in `bar` is a field `w` of the object `z` passed as its parameter, we invoke method `baz` to set the field `w` appropriately. To summarize, the following context can be derived to manifest the potential race under consideration:

```
z.baz(x);  a.bar(z); a'.bar(z); // context
a.foo(...); // thread 1, unprotected access
a'.foo(...); // thread 2, unprotected access
```

Setting the context as shown above accomplishes the task of appropriate object sharing. However, in general, for a successful execution of the synthesized code, we need *legal* object instances of different types (e.g., x, z, a, a'). In the next section, we address the challenge of creating such object instances to allow us to thereby correctly set the required context and invoke relevant methods concurrently from different threads to expose a race.

3.4 Synthesizing Tests

We now describe our approach for synthesizing an executable test. Primarily, we need to have legal object instances to execute the synthesized code as described in previous sections. For this purpose, we use a simple yet effective approach where we execute the sequential tests, that have been used earlier in our analysis, *multiple* times. However, instead of running the tests to completion, we suspend the execution before a method is invoked on the objects of interest, i.e., those that have been determined to be relevant for a race. For example, if we need to collect objects of type `Z` and `X` that need to be passed to z.`baz`(x), we execute the sequential test until the invocation of `baz` and collect the objects pointed by z and x for constructing the racing test. In other words, we store the references to these objects for later use. Similarly, if we need to invoke a method twice such that each invocation is with a different set of parameters, we execute the sequential test twice and collect the objects before the two invocations.

Unless object sharing is explicitly required, we do not share objects across different method invocations. If the objects are shared unnecessarily, it can potentially disable race detection. For instance, in the running example (Figure 13), when `foo` needs to be invoked separately by two threads, we ensure that two different sets of object instances are collected to be used as receivers to the invocations respectively. Otherwise, if the object referenced by the receivers are the same, the two threads cannot enter the synchronization body in `foo` concurrently. On the other hand, with the objects being distinct, concurrent access to the synchronized body becomes feasible. Conversely, in the running example, when the context is being set and `bar` is invoked twice, we ensure the same object of type `Z` is passed as parameter to both the invocations.

Algorithm 1 Outline of a Generated Test

Input: Method pairs $\mathtt{m}_r, \mathtt{m}_{r'}$ to manifest a race (r, r').
Sequence of relevant setter methods Q_r, Q'_r.
1: **for** (each `m` in Q_r) $\mathtt{O}_r[m] \leftarrow$ `collectObjects(m)`
2: **for** (each `m` in $Q_{r'}$) $\mathtt{O}_{r'}[m] \leftarrow$ `collectObjects(m)`
3: $\mathtt{P}_r \leftarrow$ `collectObjects`($\mathtt{m}_r$)
4: $\mathtt{P}_{r'} \leftarrow$ `collectObjects`($\mathtt{m}_{r'}$)
5: `shareObjects`($\mathtt{P}_r, \mathtt{P}_{r'}, \mathtt{O}_r, \mathtt{O}_{r'}$)
6: **for** (each `m` in Q_r) Invoke `m` with $\mathtt{O}_r[m]$ as parameters.
7: **for** (each `m` in $Q_{r'}$) Invoke `m` with $\mathtt{O}_{r'}[m]$ as parameters.
8: Spawn a new thread and invoke $\mathtt{m}_r$ with parameters in $\mathtt{P}_r$.
9: Spawn a new thread and invoke $\mathtt{m}_{r'}$ with parameters in $\mathtt{P}_{r'}$.

Algorithm 1 presents the outline of the multi-threaded tests synthesized by our analysis. To synthesize a test that manifests a race between accesses r and r', we use the corresponding method invocations $(\mathtt{m}_r, \mathtt{m}_{r'})$ as input. We also use a pair of method sequences given by Q_r and $Q_{r'}$ as input to set the relevant context which enables the objects to be driven to a state conducive for manifesting a race. Initially, we collect the objects passed as parameters to the various methods in the sequences using the auxiliary function, `collectObjects` (lines 1−2). The auxiliary function invokes the appropriate sequential test, suspends the execution before the method of interest is invoked and stores the references to the objects passed as parameters to the invocation. After collecting the objects for the methods involved in setting the context, we also collect the objects for the pair of methods, $\mathtt{m}_r$ and $\mathtt{m}_{r'}$, which contain the racy accesses (lines 3−4). We use the auxiliary function, `shareObjects` to re-arrange objects among $\mathtt{P}_r, \mathtt{P}_{r'}, \mathtt{O}_r$ and $\mathtt{O}_{r'}$ so that invoking the methods with the updated object references will help expose the race. Since all the pre-requisites for manifesting a race are satisfied, we invoke the methods in the setter methods with the appropriate objects (lines 6−7). Subsequently, we spawn new threads and invoke the methods that contain the racy accesses concurrently.

Table 2: Application of Algorithm 1 on example.

	Before sharing	After sharing
O_r	`baz`: (z_1, x_1), `bar`: (a_2, z_2)	`baz`: (z_1, x_1), `bar`: $(\boxed{a_5}, z_1)$
$O_{r'}$	`baz`: (z_3, x_3), `bar`: (a_4, z_4)	`baz`: $\boxed{(z_1, x_1)}$, `bar`: $\boxed{(a_6, z_1)}$
$\mathtt{P}_r$	`foo`: (a_5, y_5)	`foo`: (a_5, y_5)
$\mathtt{P}_{r'}$	`foo`: (a_6, y_6)	`foo`: (a_6, y_6)

For the example presented in Figure 13, the method of interest is `foo`. The method sequences Q_r and $Q_{r'}$ are {`baz`, `bar`} as described previously. When the synthesized multi-threaded test is executed as shown in Algorithm 1, the collected objects for all these methods are shown in Table 2. Due to `shareObjects`, the parameters of the invocations are re-arranged suitably (shown in boxes). Subsequently, after the methods for setting the context, z_1.`baz`(x_1), a_5.`bar`(z_1) and a_6.`bar`(z_1) are executed, a_5.`x` and a_6.`x` will point to x_1. Finally, the test case invokes a_5.`foo`(y_5) and a_6.`foo`(y_6) from two newly spawned threads respectively. The synthesized test can expose a race on a_5.x.o (equivalent to a_6.x.o) as it is modified by two threads concurrently.

4. Implementation

We have implemented the analysis described in Section 3 to synthesize racy tests for multithreaded Java libraries. We use soot [28] for instrumenting the bytecode and obtain the execution traces from the sequential test for further analysis. There can be nested calls from a library invocation. We scope the variable names by assigning unique index for each method invocation.

We perform *lazy initialization* to implement the functionality described by $\mathcal{R}$ in Section 3. This is because it is not always possible to assign a separate heap location for every variable by performing a deep walk on the type graph (e.g., linked list). Therefore, when a library method is invoked from the client (the sequential test), we initialize the variables corresponding to the various parameters passed to the invocation and set the controllable and locked flags. Subsequently, for an unseen variable, we assign the flags based on its owner state. For example, the flags for an unseen variable $x.f$ will be assigned based on the state of x.

Our implementation considers the access to some field $x.f$ to be *unprotected* if a lock on x is not held. In practice, there can be scenarios where $x.f$ is always accessed with a lock held on $x.g$; in other words, there can be a strong correlation between the accesses and some other field. We adopt a conservative approach and consider the access as unprotected and attempt to synthesize a test. The downside of this conservative analysis is the synthesis of tests that will not expose any races.

It is possible to synthesize varied method sequences to set the same context. For example, there can be multiple setters for a field. Our implementation randomly selects one of the possible methods to derive the required method sequence. We treat constructor as any other method to help set the context, but discard unprotected accesses found in them while building the racing pairs. Moreover, in some scenarios, we may not be able to derive the context for assigning the entire field dereference $x.f_1 \ldots f$. We attempt to assign the prefixes of the dereference so that the objects at some point of the hierarchy are shared, which can also lead to tests that do not expose races. Methods need not always run to completion to drive an object to a specific state. For example, there can be a strong non-controllable update to a field after the controllable assignment, which can override the earlier update and will not help in setting the required context. We handle it by letting a separate thread invoke the method and suspend its execution at the label corresponding to the writeable assignment or the closest point where all held locks are released.

We integrate the output of our implementation, named NARADA, with RACEFUZZER [25]. The integration is seamless and does not require *any* modifications to the race detector.

5. Experimental Validation

We validate our approach by synthesizing racy tests for multithreaded Java libraries, including thread-safe classes, using our implementation. The experiments are performed on an Ubuntu-14.04 desktop running on a 3.5 Ghz Intel Core i7 processor with 16GB RAM. The information regarding the various libraries used is given in Table 3. `hazelcast` is an open-source in-memory data grid, `openjdk` is the Java Development Kit, `colt` is a high performance scientific computing library, `hsqldb` is a leading SQL relation database software, `hedc` is a web-crawler application, `h2` is a SQL database engine, and `classpath` contains core class libraries for use with virtual machines and compilers. The versions of the benchmarks and the classes analyzed for synthesizing races are given in the Table 3. For brevity, we refer to the analyzed classes as `C1...C9`.

Table 3: Benchmark Information.

Benchmark	Version	Class name
hazelcast	3.3.2	SynchronizedWriteBehindQueue (C1)
openjdk	1.7	SynchronizedCollection (C2) CharArrayWriter (C3)
colt	1.2.0	DynamicBin1D (C4)
hsqldb	2.3.2	DoubleIntIndex (C5) Scanner (C6)
hedc	NA	PooledExecutorWithInvalidate (C7)
h2	1.4.182	Sequence (C8)
classpath	0.99	CharArrayReader (C9)

Table 4: Synthesized test count and synthesis time.

Class	Methods	LoC	Race Pairs	Tests	Time (in secs)
C1	14	104	65	15	12.2
C2	19	85	131	40	13.5
C3	13	92	13	9	2.2
C4	35	313	26	11	33.0
C5	32	508	136	8	7.4
C6	26	1802	85	8	121.7
C7	9	191	4	4	3.6
C8	18	233	4	4	5.8
C9	8	102	2	2	1.9
Total			466	101	201.3

Table 4 presents the number of methods and the lines of code in each class. If a brute-force approach is employed to detect races in a class, not only all possible pairs of methods need to be invoked concurrently (e.g., 35 × 35 for `C4`), the objects passed to them should be in a state that can induce a race. Consequently, the overall search space is quite large. We analyze the classes with our implementation by constructing a seed-testsuite for each class where each method in the class is invoked exactly once, without *any* constraints on the state of the objects involved in the invocations. With this trivial effort, we are able to identify 466 racing pairs across all the classes.[5] Because our analysis also provides enough information pertaining to the context, it helps prune the search space significantly leading to the overall synthesis of 101 tests. It is not necessary for all racing pairs to have unique tests because there can be multiple unprotected accesses of the same field within a method. For example, 15 tests are synthesized to enable detection of 65 racing pairs in `C1`. The overall analysis time for all the classes to synthesize the tests is less than four minutes.

Table 5 shows the results of applying RACEFUZZER [25], a dynamic data race detection tool, on the execution of the tests synthesized by NARADA. 307 races were detected in total, of which 259 races were automatically reproduced. We manually analyzed the output of the automatically reproduced races and identified that 187 of these races are indeed harmful. The 62 benign races in `C6` are due to a **reset** method which resets a number of fields to constant values. Moreover, we manually triaged races that could not be reproduced by RACEFUZZER, and discovered that 44 out of these 48 races were true positives (TP), with the remaining four being false positives (FP) that arise due to imprecision in the detector.

[5] We did not list eight other classes in `openjdk` because the races were very similar to the races in `SynchronizedCollection`.

Table 5: Analysis of synthesized tests by RaceFuzzer. Manually analyzed data for races detected by RaceFuzzer but not reproduced by it is also shown.

Class	Races	Reproduced		Manual	
	Detected	Harmful	Benign	TP	FP
C1	76	58	2	12	4
C2	84	65	1	18	-
C3	8	7	1	-	-
C4	4	2	0	2	0
C5	36	30	6	-	-
C6	89	15	62	12	-
C7	4	4	-	-	-
C8	4	4	-	-	-
C9	2	2	-	-	-
Total	307	187	72	44	4

There are a few races that are not feasible for which our system produces a racing pair. For example, we construct 26 racing pairs for `C4` where only four races are detected. This is because the races on some fields can never manifest as the necessary fields to set a suitable context can never be influenced from clients. The implementation of Narada and the raw experimental data are publicly available[6] and we refer the interested reader to it for further details.

Figure 14 presents the distribution of tests as a function of the number of detected races. For `C5`, `C6...C8`, each test detects at least one race. For the remaining classes, we also synthesize tests that do not detect any race. As mentioned previously, when we are unable to set a context for a specific field, we try to achieve object sharing to the extent possible by making assignments to its ancestors. For example, in `C4` where setting the context is not possible, we still synthesize tests for those racing pairs resulting in a majority of the tests not enabling race detection.

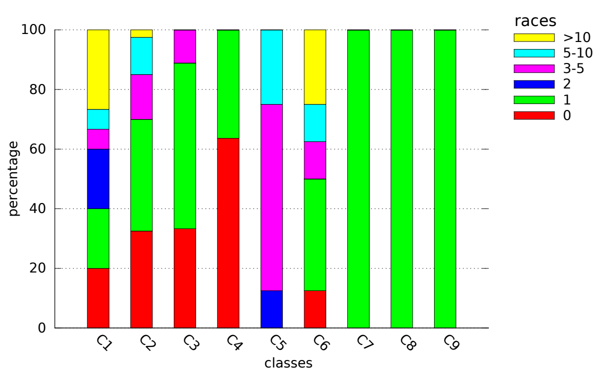

Figure 14: Distribution of tests w.r.t the number of detected races.

We also analyzed these library classes with contege [20], a tool that performs a *random* search to detect thread safety violations. It was able to detect two thread-safety violations in `C5` and one thread-safety violation in `C6` by generating 2.9K and 105 tests respectively. The detected thread-safety violations are races, which are also detected by our analysis. For other benchmarks, it generated between 1K- 70K tests, yet was unable to detect any thread-safety violations.

[6] `http://www.csa.iisc.ernet.in/~sss/tools/narada`

6. Related Work

We are the first to design and implement a *directed* approach for synthesizing tests to enable race detection. In ConTeGe [20], the authors propose an approach for precisely detecting thread safety violations by generating *random* multithreaded executions. If the multithreaded execution crashes or deadlocks and the corresponding serialized execution executes without a problem, the tool reports a thread safety violation. Because of the randomized nature of executing the tests, any two methods can be invoked concurrently and the objects passed to them need not necessarily be shared. In contrast, our approach analyzes the sequential execution to identify problematic regions and synthesizes a racy test accordingly. Because of the directed nature of our design, we are able to prune the overall search space for races effectively.

In previous work [22], we designed an approach for synthesizing tests to detect deadlocks [23]. We differ from this approach in the intended application (races *vs* deadlocks) and also the properties that need to be inferred for the purpose of synthesizing tests to enable race detection. In [6], the authors propose an approach that employs concolic execution techniques to cover code regions to enable race detection. In contrast, we propose a technique for synthesizing multithreaded tests. The execution of the synthesized tests can potentially be integrated with their approach for exposing racing schedules.

A number of dynamic analysis techniques have been proposed to enable race detection. FastTrack [7] is a dynamic race detector that leverages the *happens-before* relation to detect races precisely. It improves the performance of Djit+ [19] by employing epochs to minimize the comparison of the vector clock times. RaceFuzzer [25] presents a mechanism for *fuzzing* the schedule to expose races. Eraser [24] uses the lockset algorithm to detect potential races. All these approaches require defect inducing multithreaded tests to be executed to detect races and can leverage the tests synthesized by our implementation.

Chess [13] systematically explores the state space to detect concurrency bugs. The priority-based probabilistic concurrency testing (PCT) [3] and parallel PCT [14] propose strategies for quickly detecting concurrency bugs. Maple [30] is a tool that exposes untested thread interleavings to enable concurrency bug detection. The availability of multithreaded tests is a pre-requisite for these approaches; all these approaches can benefit from the tests synthesized using our implementation. IMUnit [9] is a framework for specifying the schedules on multithreaded tests. We can leverage the framework to suitably specify racing schedules on the synthesized tests.

Many static analysis approaches have been proposed to detect races [5, 15, 21]. These approaches require a programmer to verify the correctness of the reported defect. Moreover, because of the static nature of the analysis, there can be many false positives. In contrast, by automatically synthesizing tests and integrating it with dynamic race detectors, we can automatically identify real races and eliminate the need for manual intervention.

Numerous techniques for automatically generating tests have been proposed in the literature [4, 8, 16]. The primary goal for these approaches is to improve code coverage and detect sequential bugs. Apart from the goals being different, we address a different set of challenges including identifying the relevant methods to be invoked concurrently and the appropriate sharing of objects. Our approach uses sequential tests as part of a seed testsuite and can use the output of the sequential test generators for bootstrapping.

In [17], the authors propose a synthesis mechanism inspired by test-driven development where the input/output examples are consumed to synthesize programs. Test-driven repair [27] combines static and dynamic analysis to identify modifications to code that will prevent races. We differ from these approaches in terms of the intended application and the underlying mechanism.

7. Conclusions

Multithreaded tests are necessary to detect races using dynamic analysis engines. However, developing effective multithreaded tests is hard. In this paper, we presented the design and implementation of a novel approach to automatically synthesize racy tests to enable race detection in multithreaded libraries. We demonstrate the efficacy of our approach by applying our implementation, named NARADA, on multiple open-source Java libraries leading to the detection of many harmful races.

Acknowledgements

We thank the anonymous reviewers for their feedback which helped improve the presentation of the paper. We thank Monika Dhok for proof-reading an earlier draft of the paper. We are grateful to Google India for providing travel support for the first author. The first two authors are supported by the Ministry of Human Resources and Development, Government of India.

References

[1] D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: A dialect of Java without data races. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 382–400, 2000.

[2] E. Bodden and K. Havelund. Aspect-oriented race detection in Java. *IEEE Transactions on Software Engineering*, 36(4):509–527, 2010.

[3] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 167–178, 2010.

[4] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, 2008.

[5] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 237–252, 2003.

[6] 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.

[7] C. Flanagan and S. N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 121–133, 2009.

[8] 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, pages 213–223, 2005.

[9] 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 European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 223–233, 2011.

[10] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu. Automated concurrency-bug fixing. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 389–400, 2012.

[11] 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, pages 110–120, 2009.

[12] S. McPeak, C.-H. Gros, and M. K. Ramanathan. Scalable and incremental software bug detection. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, ESEC/FSE 2013, pages 554–564, 2013.

[13] 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, pages 446–455, 2007.

[14] 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, pages 543–554, 2012.

[15] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[16] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 75–84, 2007.

[17] D. Perelman, S. Gulwani, D. Grossman, and P. Provost. Test-driven synthesis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[18] B. Petrov, M. Vechev, M. Sridharan, and J. Dolby. Race detection for web applications. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 251–262, 2012.

[19] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 179–190, 2003.

[20] 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, pages 521–530, 2012.

[21] P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: Context-sensitive correlation analysis for race detection. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 320–331, 2006.

[22] M. Samak and M. K. Ramanathan. Multithreaded test synthesis for deadlock detection. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, pages 473–489, 2014.

[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, pages 29–42, 2014.

[24] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, Nov. 1997.

[25] K. Sen. Race directed random testing of concurrent programs. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 11–21, 2008.

[26] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. Sound predictive race detection in polynomial time. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 387–400, 2012.

[27] R. Surendran, R. Raman, S. Chaudhuri, J. Mellor-Crummey, and V. Sarkar. Test-driven repair of data races in structured parallel programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[28] 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.

[29] B. Xin, W. N. Sumner, and X. Zhang. Efficient program execution indexing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[30] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: A coverage-driven testing tool for multithreaded programs. In *Proceedings of the ACM Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 485–502, 2012.