Synthesizing Tests for Detecting Atomicity Violations

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

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

ABSTRACT

Using thread-safe libraries can help programmers avoid the complexities of multithreading. However, designing libraries that guarantee thread-safety can be challenging. Detecting and eliminating atomicity violations when methods in the libraries are invoked concurrently is vital in building reliable client applications that use the libraries. While there are dynamic analyses to detect atomicity violations, these techniques are critically dependent on *effective* multithreaded tests. Unfortunately, designing such tests is non-trivial.

In this paper, we design a novel and scalable approach for synthesizing multithreaded tests that help detect atomicity violations. The input to the approach is the implementation of the library and a sequential seed testsuite that invokes every method in the library with random parameters. We analyze the execution of the sequential tests, generate *variable lock dependencies* and construct a set of three accesses which when interleaved suitably in a multithreaded execution can cause an atomicity violation. Subsequently, we identify pairs of method invocations that correspond to these accesses and invoke them concurrently from distinct threads with *appropriate* objects to help expose atomicity violations.

We have incorporated these ideas in our tool, named Intruder, and applied it on multiple open-source `Java` multithreaded libraries. Intruder is able to synthesize 40 multithreaded tests across nine classes in less than two minutes to detect 79 *harmful* atomicity violations, including previously unknown violations in thread-safe classes. We also demonstrate the effectiveness of Intruder by comparing the results with other approaches designed for synthesizing multithreaded tests.

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*

Keywords

atomicity violation; dynamic analysis; concurrency

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2786874

1. INTRODUCTION

Designing scalable and reliable multithreaded applications is challenging due to the complexities associated with multithreading. Oftentimes, developers of such applications avoid the complexities by using *thread-safe* [24] libraries. These libraries are structured such that concurrent invocation of methods from multiple clients always corresponds to some linearization of the associated invocations [24, 16, 12]. Moreover, such libraries limit the use of synchronizations to provide better performance [3] for the client applications. Maintaining thread-safety without sacrificing performance can be a demanding task. Therefore, even if a library (or component) is only *partially* thread-safe, specifying the context under which thread-safety violations occur will enable the users of the library to take corrective action.

Atomicity violations [9] are an important class of concurrency defects in multithreaded programs. Many static analyses [18, 1] and dynamic analyses [6, 23, 8, 32, 2] are designed to detect such violations. One of the many advantages of dynamic analyses (over static analyses) is that they can be used to reproduce an erroneous execution corresponding to a reported defect. While various aspects pertaining to dynamic analyses including precision [8, 2], scalability due to the number of interleavings [32, 23] and reproducibility [22] have been investigated rigorously, their effectiveness is critically dependent on the presence of defect revealing multithreaded tests. Designing such tests requires nuanced understanding of the implementation and is therefore not easy.

ConTeGe [24] attempts to address this problem by adopting a *brute-force* approach to automatically generate multithreaded tests and detect a violation by analyzing failing executions. While an important first step, the number of possible multithreaded tests is significantly large and the probability of generating a defect revealing multithreaded test can be quite low. *Firstly*, the pairs of methods that need to be invoked concurrently is a function of the number of methods in the class. *Secondly*, appropriate parameters need to be passed to such invocations so as to trigger the violation which can be combinatorial in the number of parameters. This necessitates the design of a *directed* approach to effectively synthesize multithreaded tests for detecting atomicity violations. In previous work, we have designed and implemented *directed* approaches to synthesize multithreaded tests to enable detection of deadlocks [26] and races [28]. Just as a race detector [30, 7] or a deadlock detector [14] cannot be used to detect atomicity violations [22, 8], the aforementioned synthesizers cannot be deployed for generating tests to enable detection of atomicity violations.

In this paper, we present a novel and scalable approach for *synthesizing* tests to enable detection of atomicity violations in multithreaded libraries. The implementation of the library and a sequential seed test-suite form the input to our approach. The output is a set of multithreaded tests such that appropriate methods are invoked concurrently in each test from different threads. The key insight of our technique is the derivation of specialized constraints by analyzing the sequential execution and generation of a multithreaded test that will satisfy the derived constraints when executed. Analyzing the execution of the synthesized test using an atomicity violation detector can help reveal (and confirm) the underlying bugs.

Our approach operates by analyzing the execution of the sequential tests in the seed testsuite. We maintain a record of the synchronized blocks, the objects on which the locks are held, the field accesses and their corresponding types (read or write) within each block. We construct a set of properties that need to be satisfied in a sequential execution to help identify whether a pair of consecutive field accesses can become potential candidates for atomicity violation. We analyze the recorded information and identify the potential candidate pairs of consecutive field accesses. Furthermore, we identify a *remote* [23] access for each such candidate pair that could be potentially interleaved between the consecutive accesses in a multithreaded execution.

After identifying the three field accesses that can constitute an atomicity violation, we identify the methods that need to be invoked concurrently. The parameters to the invocations should satisfy a few constraints such that the locks and accesses happen on the *appropriate* objects for the defect to manifest. These constraints are derived from the analysis of the sequential execution. To obtain the concrete objects required for the invocations in the synthesized multithreaded test, we execute the relevant sequential tests multiple times, suspend their execution and collect the necessary objects after driving them to the required state. The synthesized tests can be used by any of the atomicity violation dynamic detectors [22, 23, 32] to detect the violations.

We have implemented a tool, named Intruder, that incorporates these ideas and evaluated it on a number of open-source multithreaded Java libraries and components. Our experimental results show that Intruder is able to generate effective multithreaded tests that expose many atomicity violations. We use an atomicity violation detector based on CTrigger [23] to detect atomicity violations by analyzing the synthesized tests. We analyzed nine classes with our approach that resulted in the synthesis of 40 multithreaded tests leading to the detection of 79 harmful atomicity violations. The time taken for the entire process is less than two minutes with negligible memory overhead. We also compare Intruder with ConTeGe [24], Omen [26] and Narada [28] and show the ability of our approach in enabling detection of atomicity violations.

The paper makes the following technical contributions:

- We develop an approach to synthesize multithreaded tests to enable detection of atomicity violations 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 the methods that drive objects to states conducive for triggering an atomicity violation and reuses existing sequential tests to generate the objects for the multithreaded execution.

- We provide a detailed discussion of the design and implementation of our approach and validate our approach by analyzing many open-source `Java` libraries.

- We demonstrate the effectiveness of our approach in synthesizing effective tests for detecting atomicity violations compared to the tests synthesized by ConTeGe [24],Omen [26] and Narada [28].

2. MOTIVATION

In this section, we provide a real example from `colt`, a popular high performance scientific and technical computing library implemented in `Java` to motivate the problem addressed in the paper. The class `DynamicBin1D` is documented to be thread-safe and hence multiple clients can invoke APIs concurrently without holding any additional locks.

```
DynamicBin1D.java:
------------------
581 synchronized DynamicBin1D sampleBootstrap(
      DynamicBin1D other, ..., BinBinFunction1D function) {
584 // since "resamples" can be quite large, we care
       about performance and memory
585  int maxCapacity = 1000;
586  int s1 = size();
587  int s2 = other.size();

593  DynamicBin1D sample2 = new DynamicBin1D();
594  cern.colt.buffer.DoubleBuffer buffer2 =
          sample2.buffered(Math.min(maxCapacity,s2));

599 // resampling steps
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);

606   bootBuffer.add(function.apply(sample1,sample2));
607  }
611 }


438 synchronized void sample(int n,
             boolean withReplacement, ...) {
442  if (!withReplacement) { // without
        ...
449  else { // with
450   Uniform uniform = new Uniform(randomGenerator);
451   int s = size();
452   for (int i=n; --i >= 0; ) {
453     buffer.add(this.elements.getQuick(
                    uniform.nextIntFromTo(0,s-1)));
454   }
456  }
457 }


134 synchronized void clear() {
137  if (this.elements != null) this.elements.clear();
141 }


DoubleArrayList.java:
---------------------
217 /* You should only use this method when you are
       absolutely sure that the index is within bounds.*/
222 public double getQuick(int index) {
223   return elements[index];
224 }
```

Figure 1: Motivating example.

Figure 1 presents partial implementations of two classes from the library. The implementations of `sampleBootstrap`, `sample` and `clear` from `DynamicBin1D` are shown in the figure. As shown, all the three methods are *synchronized*. The implementation of `getQuick` from `DoubleArrayList` is also shown in the figure where the comment above the implementation emphatically places the burden of using the appropriate `index` on the caller of the method.

We claim that clients using `DynamicBin1D` can observe violation of atomicity properties depending upon the invoked methods and the objects on which the methods are invoked. More specifically, executing the multithreaded program shown in Figure 2 can

```
public void testAtomicity() {
   DynamicBin1D bin1 = new DynamicBin1D(...);
   DynamicBin1D bin2 = new DynamicBin1D(...);
     ...
   Thread t1 = new Thread() {
       void run() { bin1.sampleBootstrap(bin2,...); }
   }
   Thread t2 = new Thread() {
       void run() { bin2.clear(); }
   }
}
```

Figure 2: Multithreaded test to expose atomicity violation.

expose the atomicity violation in `DynamicBin1D` when the following happens:

- The first thread initiates the execution of `bin1.sampleBootstrap`.
- After `other.size` at line 587 is executed by the first thread, `bin2.clear` is executed from the second thread clearing all the elements in `bin2`. This should invalidate the value of `s2` obtained at line 587.
- Subsequently, when `other.sample(...)` at line 605 is invoked from the first thread, the loop in `sample` at line 452 *ideally* should not execute but is executed for a few iterations (depending upon `s2`) incorrectly.
- Elements from `this.elements` at line 453 are obtained using *invalid* indices as part of the execution of the loop conflicting with the comments given at line 217 in `DoubleArrayList.java`.

Designing this multithreaded test manually is a non-trivial task because it requires a nuanced understanding of `DynamicBin1D` and the associated classes. More specifically, we need to identify that the methods `sampleBootstrap` and `clear` need to be invoked concurrently among the 35 methods present in the class. Even though there are at least four parameters to `sampleBootstrap`, the necessary constraint that needs to be satisfied is that its first parameter and the receiver of `clear` need to refer to the same object. Moreover, there may be scenarios where the execution does not crash hiding the presence of the error from the user. For example, the distribution of the elements returned from `sample` can become biased (non-uniform) invalidating the relevant statistical computations. When we apply [24, 26, 28], none of the tools are able to synthesize this multithreaded test automatically. This is because [24] is random (and the maximum time to generate tests was set as five hours), [26] is searching for deadlocks and [28] is searching for races.

```
public void test() {
   DynamicBin1D bin1 = new DynamicBin1D(...);
   DynamicBin1D bin2 = new DynamicBin1D(...);
     ...
   bin1.sampleBootstrap(bin2,...);
   bin1.clear();
}
```

Figure 3: Sequential seed test.

Using the analysis described in the paper, we are able to automatically synthesize the multithreaded test shown in Figure 2. The implementation of the library along with a sequential seed test shown in Figure 3 forms the input to our analysis. We observe that a sequential seed testsuite that invokes each method in the class under consideration once with random parameters can be used to synthesize defect revealing multithreaded tests. Moreover, writing such sequential seed tests is trivial. We emphasize that in the sequential test, we invoke `clear` with `bin1` instead of `bin2` and yet are able to synthesize the required multithreaded test. We analyze the execution traces of the seed tests to construct the multithreaded test. The rest of the paper describes our analysis that results in this construction.

3. DESIGN

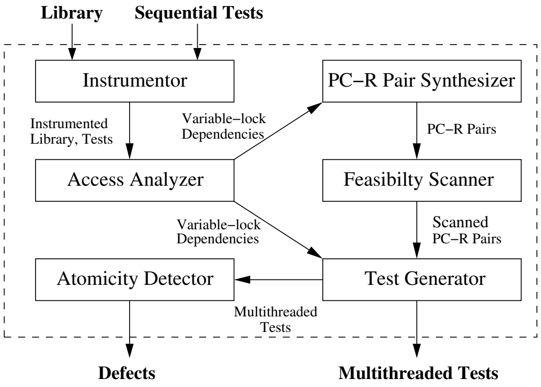

Figure 4: Overall Architecture

The architecture of our tool, named Intruder, which generates multi-threaded tests to enable detection of atomicity violations is presented in Figure 4. The entire process of synthesizing the tests is accomplished by integrating multiple components. The implementation of the library under test and the sequential seed testsuite form the input to the `Instrumentor` which performs the necessary instrumentation to track the lock acquisitions and releases, variable accesses, method invocations, etc. `Access Analyzer` takes as input the instrumented library and the sequential tests, executes the tests and monitors the execution to output *variable-lock dependencies*, where a dependency specifies the relation between the access and the various locks in the execution. The accesses are further analyzed by the `PC-R Pair Synthesizer` to derive a set of `PC-R` pair accesses[1]. The underlying intuition is that if these accesses are performed on the *same* object such that two accesses (previous and current) happen consecutively when a library method is invoked from one thread and another (remote) access happens due to a method invocation from a different thread such that the remote access interleaves with the other two accesses, then an atomicity violation can manifest. `Feasibilty Scanner` analyzes the generated pairs and eliminates the pairs where the remote access cannot interleave due to a variety of factors including being guarded by the same locks. `Test Generator` uses the feasible `PC-R` method pairs and the variable lock dependencies to synthesize the multithreaded tests to expose atomicity violations. These tests can then be used by a third-party detector (e.g., AtomFuzzer [22], CTrigger [23], etc) to detect atomicity violations. For our experiments, we use an atomicity violation detector based on CTrigger to expose atomicity violations.

3.1 Preliminaries

In this section, we provide the necessary background for our approach by describing a few primitives. To be able to synthesize a test, it is necessary to identify the defective accesses, the ability of a client to influence accesses and its ability to drive objects to states conducive for a defect to manifest. For this purpose, we

[1] PC-R stands for previous, current and remote [23].

use the primitives symbol, controllable and setter defined in [28] and briefly describe them here for the sake of completeness. We provide the explanations on a running example from Figure 5.

We define a variable to be *controllable* if it references an object within the implementation of a library that can be manipulated by clients through library methods. The controllability of the variable persists until it is reassigned to a reference that cannot be manipulated by the client (e.g., a newly allocated object). For example, in Figure 5, the variable x is controllable as it holds a reference to this.f where this (and thereby this.f) can be manipulated by a client. We use a primitive controllable(*var*) to identify the controllability of *var*.

```
class A {                       class B {
 B f = new B();                  int count = 0;

 void sync foo() {               int sync getCount() {
   B x = this.f;                   return count;
   int c = x.getCount();         }
   x.setCount(c+1);
 }                               void sync setCount(int c) {
                                   this.count = c;
 void sync setF(B b) {           }
   this.f = b;                  }
 }
}
```

Figure 5: Running example; synchronized is shortened to sync.

We define an assignment in the implementation of a library method to be *writeable* provided both sides of the assignment are controllable and the assignment is to a field. Intuitively, this provides a mechanism for the client to assign a reference of choice to the field of an object which can be helpful in driving an object to a desired state. For example, in Figure 5, the assignment of the parameter to the field f in method setF is writeable because both sides of the assignment are controllable and the field f of the object is updated. In contrast, the assignment to x in method foo is not writeable because it does not update a field. We define *writeable methods* as the methods which contain writeable assignments. Clients can invoke a sequence of these methods to drive the object to a desired state. In the running example, the method setF is a writeable method.

For our analysis, we will need to be able to assign some field $o.f_1.f_2 \ldots f_n$ to a desired object so that we can have two different containers o and o', such that $o.f_1.f_2 \ldots f_i \ldots f_n = o'.f_1.f_2 \ldots f_i \ldots f_n$, while $o.f_1.f_2 \ldots f_i \neq o'.f_1.f_2 \ldots f_i$ for some i $(0 < i < n)$. To achieve this, we define a primitive setter(x, y), where $\mathtt{x} = o.f_1.f_2 \ldots f_i$ and $\mathtt{y} = o.f_1.f_2 \ldots f_i \ldots f_n$, to return a sequence of writeable methods that enable setting of $o.f_1.f_2 \ldots f_i \ldots f_k$ to an object as specified by the client, where $i < k \leq n$. While ideally k should be equal to n, even if k is some value between i to n, it satisfies the above mentioned requirement. For the running example, setter(a, a.f) returns a unit length sequence setF that helps set a.f suitably.

We build *aliases* of variables for each method invocation afresh. Irrespective of the value of the references used as parameters[2] during a method invocation, each parameter is aliased with a fresh symbol. We also perform a deep walk of various fields of the parameters and associate them with fresh symbols. Subsequently, for an assignment x := y, we associate x to be aliased with the symbol for y and its fields to alias with the symbol associated with the fields of y. In this manner, we maintain *alias sets* per program point. We define a primitive symbol that takes a variable and returns its associated symbol.

For example, if a client executes a.setF(b) twice consecutively, then before the first invocation of setF, the variables a, a.f and b will be aliased to three *different* symbols (i_0, i_1, i_2). Therefore, symbol(a), symbol(a.f) and symbol(b) will return i_0, i_1 and i_2 respectively. After executing the assignment in setF, a.f and b become aliases. For the second invocation, even though the references pointed by a.f and b are equal, the parameters and their fields are aliased with *fresh* symbols similar to the first invocation. This enables the analysis to differentiate variables that are aliases due to the design of the API (e.g., this.f and b before returning from setF) as compared to the design of the invoking client (e.g., the two invocations of setF as shown above). For the purposes of synthesizing tests, we are interested in the aliases due to the design of the API.

For a symbol i_n, we define a set of *relevant locks* $\{i_0, i_1, \ldots, i_n\}$ where $i_i = \mathtt{symbol}(o.f_1.f_2 \ldots f_i)$, $i_0 = \mathtt{symbol}(o)$ and o is an input parameter. This is because there is a strong correspondence between the locks in the relevant lockset with the symbol under consideration. We explain this using the following method rand:

```
public void rand(U u)  {
   synchronized(u.f1) { ...access(u.f1.f2.f3)... };
}
```

a field u.f1.f2.f3 is accessed while a lock on u.f1 is held. The relevant lock set for accessing field f3 in u.f1.f2 is {u, u.f1, u.f1.f2} which includes u.f1. Even though, initially, it appears that the above code is a *safe* way of accessing the field, we emphasize that such constructs can be deceptive. If a client can set u.f1.f2 to an object of its choice, then it can take two distinct objects o and o' and set o.f1.f2 and o'.f1.f2 to refer to the same object. Subsequently, when rand(o) and rand(o') methods are invoked concurrently, o.f1 and o'.f1 differ and both threads enter the synchronized region and simultaneously access o.f1.f2.f3 and o'.f1.f2.f3 which correspond to the same memory location. To expose such subtle violations, we maintain the relevant lock set where all possible prefixes of a dereference are considered interesting.

The analysis underlying the implementation of the primitives, controllable, setter and symbol is described elaborately in [28]. We build an approach that leverages these primitives appropriately. We now describe each component of INTRUDER elaborately.

3.2 Instrumentor

The implementation of the library and the sequential tests form the input to the Instrumentor. It instruments the library implementation to process the instructions corresponding to lock acquisition and release, and field accesses. It also provides a mechanism to identify the boundary between the client (in our case, sequential tests) and the library implementation to enable initialization of appropriate data structures for further analysis. This is to differentiate the method invocation in the library from another method within it as opposed to invocation from the client. The initialization will happen only in the latter scenario. The appropriately instrumented code and tests form the input to the next phase of the analysis.

3.3 Access Analyzer

The main goal of this phase is to record all field accesses and maintain information pertaining to the locks. We define two kinds of locksets – set of *held* locks for an access and set of *consistently held* locks for a pair of accesses. The held locks specify the locks held for the access. The consistently held locks specify the locks held for the pair of consecutive accesses without being released *between* the two accesses. Maintaining the set of consistently held

[2] We include the receiver as the first parameter of the invocation.

locks enables further phases to identify possible locations to violate atomicity.

The `Access Analyzer` executes the sequential tests on the instrumented code and appropriately processes the relevant instructions. It outputs a set of variable lock dependencies that corresponds to each field access in the library. More specifically, it maintains the following structures, with the variable lock dependencies (D) forming the final output.

- $H \in 2^S$, where $S = Symbol \times N$
- $H_C \in 2^{Symbol}$; $RL : Symbol \mapsto 2^{Symbol}$
- $AI : Symbol \mapsto 2^F$, where $F : Field \mapsto \{\mathtt{R}, \mathtt{W}, \perp\} \times \{N \cup \perp\}$
- $D \in 2^P$, where $P = Symbol \times Field \times \{\mathtt{R}, \mathtt{W}, \perp\} \times \{\mathtt{R}, \mathtt{W}\} \times H \times H_C$

Recall from Section 3.1 that a fresh symbol is associated with each parameter. These symbols are from $Symbol$ ($i_0, i_1, \dots$). When a method is invoked from a client, the parameters and the various fields reachable from them are assigned fresh symbols from this set. We define H to maintain the information pertaining to the currently held locks and also associate a timestamp pertaining to when the lock is obtained. The timestamp is a counter which is incremented on every lock acquisition and release and is obtained from the set of natural numbers (N). H_C is the set of consistently held locks. We also maintain a *relevant* lockset (RL) that is a mapping from a symbol to a set of symbols as discussed in Section 3.1.

We maintain a map pertaining to the access information (AI). For each symbol, it maintains information corresponding to all fields in the object associated with the symbol. The information for the field include its most recent access kind (read, write or unknown) and the timestamp of this access. Initially, all accesses are unknown ($\perp$). We also maintain D, which is a set of tuples specifying the variable lock dependencies for each access, where the six elements in each tuple are as follows: the symbol of the referenced object, the accessed field, the kind of the previous access, the kind of the current access, the set of held locks, and the set of consistently held locks for the current and previous accesses of the field.

Algorithm 1 genVarLockDependency

```
 1: $H \leftarrow \phi$; $\delta \leftarrow 0$
 2: Initialize $AI$ and $RL$;
 3: $s \leftarrow$ next executable instruction;
 4: while s is not return from the client invoked method do
 5:     if $s$ is read($var$, $field$) and controllable(var) then
 6:         recordAccess($var, field, \mathtt{R}$)
 7:     else if $s$ is write($var$, $field$) and controllable(var) then
 8:         recordAccess($var, field, \mathtt{W}$)
 9:     else if $s$ is lock($var$) then
10:         Increment $\delta$ by one
11:         Add (symbol($var$), $\delta$) to $H$
12:     else if $s$ is unlock($var$) then
13:         Increment $\delta$ by one
14:         Remove the element (symbol($var$), $*$) from $H$
15:     $s \leftarrow$ next executable instruction;
```

When a library method is invoked from the client, Algorithm 1 is applied. H is initialized to empty and the timestamp δ reset to 0. We also initialize AI and RL as mentioned previously. If the executed instruction is a lock or a release, the timestamp is incremented and H is updated accordingly (lines 9-14). If the instruction is an access to a controllable variable, then `recordAccess` (see Algorithm 2) is invoked with the appropriate access kind (lines 5-8).

In Algorithm 2, we obtain the symbol associated with the variable that is being accessed using `symbol`. The consistently held locks (H_C) is initialized to empty (line 1). Then, information pertaining to the previous access of the field is obtained from AI (line 2). For all the currently held locks, we obtain the relevant held locks (H_R) based on RL and H[3]. Subsequently, between lines 4 to 6, for each relevant held lock, we check whether the lock is acquired before the previous access of the currently accessed field and H_C is updated accordingly. The tuple pertaining to the access is added to D (line 7) and the field access information is also updated (line 8).

Algorithm 2 recordAccess

```
Input: $var$: dereferenced variable, $field$: accessed field in $var$, $\tau$:
    access kind.
 1: $i \leftarrow$ symbol($var$); $H_c \leftarrow \phi$
 2: $(\tau_p, \delta_p) \leftarrow AI[i][field]$ // previous access kind and timestamp
 3: $H_R \leftarrow$ locks($H$) $\cap RL[i]$
 4: for each lock $\ell$ in $H_R$ do
 5:     if timestamp($\ell, H$) $\leq \delta_p$ then
 6:         $H_C \leftarrow H_C \cup$ symbol($\ell$)
 7: $D \leftarrow D \cup \left\langle i, field, \tau_p, \tau, H_R, H_C \right\rangle$
 8: $AI[i][field] \leftarrow (\tau, \delta)$
```

We now illustrate the process when `a.foo()` is invoked for some object `a` from a sequential test on the code given in Figure 5. Algorithm 1 is invoked and fresh symbols i_0 and i_1 are allocated. i_0 and i_1 corresponding to objects referenced by `a` and `a.f` respectively. AI is initialized to $\{i_0 \mapsto \{\mathtt{f} \mapsto (\perp, \perp)\}, i_1 \mapsto \{\mathtt{count} \mapsto (\perp, \perp)\}\}$, and RL is initialized to $\{i_0 \mapsto \{i_0\}, i_1 \mapsto \{i_1, i_0\}\}$. H and δ are initialized to empty and zero respectively.

Table 1: State changes when `a.foo()` is invoked by client.

Instruction	AI	H	δ
`lock(a)`		$\{(i_0, 1)\}$	1
`read(this,f)`	$\{i_0 \mapsto \{\mathtt{f} \mapsto (\mathtt{R}, 1)\}$, $i_1 \mapsto \{\mathtt{count} \mapsto (\perp, \perp)\}\}$		
`lock(x)`		$\{(i_0, 1)$, $(i_1, 2)\}$	2
`read(x,count)`	$\{i_0 \mapsto \{\mathtt{f} \mapsto (\mathtt{R}, 1)\}$, $i_1 \mapsto \{\mathtt{count} \mapsto (\mathtt{R}, 2)\}\}$		
`unlock(x)`		$\{(i_0, 1)\}$	3
`lock(x)`		$\{(i_0, 1)$, $(i_1, 4)\}$	4
`write(x,count)`	$\{i_0 \mapsto \{\mathtt{f} \mapsto (\mathtt{R}, 1)\}$, $i_1 \mapsto \{\mathtt{count} \mapsto (\mathtt{W}, 4)\}\}$		
`unlock(x)`		$\{(i_0, 1)\}$	5
`unlock(a)`		ϕ	6

Table 1 presents the updated AI, H and δ while executing the list of instructions. Before `lock(a)` is executed, δ is incremented and H updated to $\{(i_0, 1)\}$. When `read(this,f)` happens, AI is updated to reflect the *read* of `f` at timestamp equals one. Moreover observe that even though the method `getCount` is within the library, the states are not reset because the invocation is also within the library. The remaining rows are obtained similarly.

Table 2 presents the set of variable lock dependencies (D) that are created based on the accesses observed during the execution of `a.foo()`. When `read(this,f)` happens, the symbol corresponding to the variable `this` is i_0, the field accessed is `f`, there is no previous access of the field `f`, the current access is a read, the set of held locks correspond to $\{i_0\}$ and finally due to the absence of any previous access, the set of consistent locks is also $\{i_0\}$.

[3] `locks`(H) returns the list of locks without the timestamp from H.

Table 2: D: Tuples after executing `a.foo();` i_0 = `a`, $i_1 = i_0$.`f`.

Instruction	Variable-lock dependency	ID
`read(this,f)`	$\langle i_0, \texttt{f}, \bot, \texttt{R}, \{i_0\}, \{i_0\}\rangle$	d_1
`read(x,count)`	$\langle i_1, \texttt{count}, \bot, \texttt{R}, \{i_0, i_1\}, \{i_0, i_1\}\rangle$	d_2
`write(x,count)`	$\langle i_1, \texttt{count}, \texttt{R}, \texttt{W}, \{i_0, i_1\}, \{i_0\}\rangle$	d_3

When `write(x,count)` is performed, `recordAccess` finds a previous read access to `count` at $\delta = 2$. Moreover, it obtains the relevant locks for i_1 to be $\{i_1, i_0\}$ from RL and performs an intersection with the symbols associated with held locks $\{i_1, i_0\}$ (line 3 of Algorithm 2). When it iterates over these locks, it finds that the lock associated with i_1 is obtained at $\delta = 4$ which is greater than the timestamp for the previous access. In contrast, it detects that the lock associated with i_0 is obtained at $\delta = 1$ which is less than the timestamp for the previous field access ($\delta = 2$). Hence, the consistent lock set becomes $\{i_0\}$. In other words, we derive that some object represented by i_0 is held without being released for two consecutive accesses of the field `count`.

The set of variable-lock dependencies that are recorded for each access during the execution of the test is used subsequently to construct a set of *three* accesses that contribute to an atomicity violation. We now discuss the synthesis of variable-lock dependency pairs which can encode such access triplets.

3.4 PC-R Pair Synthesizer

For each variable lock dependency, we record the previous and current access of the associated field. We will precisely use this data along with the current access from a different variable lock dependency to construct the set of three accesses. The process of detecting atomicity violations using the previous (p), current (c) and remote (r) accesses was used successfully in CTrigger [23]. Even though we do not have multiple threads (because the analysis is on sequential executions) unlike CTrigger, we build our approach to synthesize the PC-R pairs inspired by their formulation.

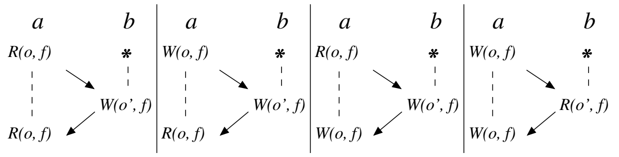

Figure 6: Four pairs of variable-lock dependencies that can help expose problematic interleavings.

Figure 6 presents four different pairs (a, b) of variable lock dependencies. Let us consider the first pair of variable lock dependencies (the left most in the figure). The a in the first pair specifies that there is a sequential test invoking a library method that *reads* $o.f$ consecutively. Here p and c are reads. The b in the pair is due to another invocation to a library method (possibly the same as before) where there is one *write* access to $o'.f$. Here r is a write. Even though the first invocation accesses $o.f$ and the second invocation accesses $o'.f$ (different memory locations), we observe that if we synthesize a test that makes the second invocation such that it also accesses $o.f$ from a different thread, we can potentially observe an atomicity violation. Moreover, the previous access in the second invocation is irrelevant because it is only necessary for one problematic access to interleave with the first invocation. Therefore, the previous access in b is denoted as *. The other three pairs of variable lock dependencies can be explained similarly.

Four other pairs (out of eight possible combinations) that are not shown in the figure cannot be used to introduce an atomicity violation. For example, if the access in b in the right most pair (in Figure 6) is a write instead of a read, then the interleaving cannot cause an atomicity violation even if the memory locations are the same [23]. Hence, we discard such pairs of variable lock dependencies while synthesizing the PC-R pairs.

Algorithm 3 PC-R Pair Synthesizer

```
1:  $T \leftarrow \phi$
2:  for all ordered pairs $a, b \in D$ do
3:      if getField($a$) = getField($b$) then
4:          $c \leftarrow$ current($a$) ; $p \leftarrow$ previous($a$)
5:          $r \leftarrow$ current($b$)
6:          if $c$ = R and $r$ = W then
7:              if $p \neq \bot$ then add $(a, b)$ to $T$
8:          else if $c$ = W then
9:              if $p$ = R and $r$ = W then add $(a, b)$ to $T$
10:             else if $p$ = W and $r$ = R then add $(a, b)$ to $T$
```

Algorithm 3 presents the algorithm to synthesize the PC-R pairs. It performs the analysis by considering all ordered pairs of dependencies (a, b) in D, where a and b are distinct. If the fields associated with the dependencies are the same (fully qualified fields obtained using the auxiliary function `getField`), it obtains p, c and r using the auxiliary functions `current` and `previous`. It applies the checks illustrated in Figure 6 and adds the ordered pair accordingly (lines 6-10). The set of triplet accesses (represented as a pair) will be available in T at the end of the phase.

When this algorithm is applied on D shown in Table 2, we get $T = \{(d_3, d_3)\}$. This is because the field under consideration is `A.f.count`, the p, c and r are `R`, `W` and `W` respectively. There are no other pairs that satisfy the constraints specified in the algorithm.

3.5 Feasibility Scanner

The combination of accesses derived as mentioned above is necessary to expose an existing atomicity violation. However, the analysis does not consider the locks that are held during the accesses which can cause some of the combinations to become infeasible. The underlying intuition with this phase is that if a lock ℓ is held for p and c accesses without being released in the interim and a lock ℓ is also held for the access corresponding to r, then the PC-R pair access becomes infeasible. In this section, we describe our approach for identifying the feasibility of the derived PC-R pairs from the previous phase based on their lock dependencies.

We define auxiliary function `path` which takes two symbols as input and returns the path between their corresponding dereferences. For example, if the symbol x corresponds to $i_0.f_0 \ldots f_i$ and y corresponds to $i_0.f_0 \ldots f_i \ldots f_n$, then `path`$(x, y)$ will return $f_{i+1} \ldots f_n$. The auxiliary function `type` returns the fully qualified type of the object corresponding to the input symbol.

For any dereference $o.f_1.f_2 \ldots f_n$, we define the prefix $o.f_1.f_2 \ldots f_{n-1}$ to *dominate* the prefix $o.f_1.f_2 \ldots f_{n-2}$, and so on. We define an auxiliary function `dominant` that takes two symbols, finds the associated dereferences and returns the dominating symbol. $\bot$ is dominated by all symbols. The dominant function is used to determine the dominating lock for an access. The dominant lock, if it exists, needs to be broken for an atomicity violation to manifest. In other words, during the invocation of some method, if locks on $o.f_1$, $o.f_1.f_2$, $\ldots$, $o.f_1 \ldots . f_i$ are acquired, and the atomicity violation needs to happen on $o.f_1 \ldots . f_i \ldots f_n$, then all locks up to and including $o.f_1 \ldots . f_i$ need to be distinct to ensure concurrent access when the methods are invoked from separate threads. However, at least one of the subsequent fields from $o.f_1 \ldots . f_i . f_{i+1}$ needs to refer to the same object so that the memory location on which the violation needs to happen becomes shared.

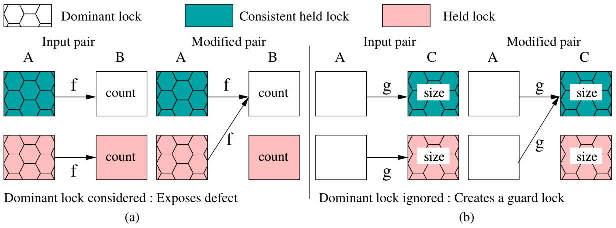

Figure 7: Illustration for the need for detecting dominant locks.

We explain this using the illustration shown in Figure 7. We have accesses of `count` such that different locks are held. The memory locations associated with `count` are also different. For the atomicity violation to manifest, the memory location for `count` needs to be the same as shown in the modified pair in Figure 7(a). Also, the locks need to be distinct so that the interleaving becomes possible. Our approach needs to consider the dominant lock dependency so that a useful test is generated. Otherwise, when the test is run, the dominant lock guards the shared access preventing any detection. Sometimes, detection of dominant locks helps identify when violation is never feasible. For example, in Figure 7(b) (based on the code given in Figure 8), the dominating lock and the object containing the field are the same. Therefore, there is no way of modifying the fields to induce an atomicity violation.

Algorithm 4 Feasibility Scanner

```
1: $T_S \leftarrow \phi$
2: for each $(a, b) \in T$ do
3:     $H_C \leftarrow$ consistentLocks($a$) ; $H_R \leftarrow$ heldLocks($b$)
4:     $i \leftarrow$ symbol($a$); $j \leftarrow$ symbol($b$)
5:     $dom \leftarrow \perp$
6:     for each pair $\alpha, \beta$ where $\alpha$ in $H_C$, $\beta$ in $H_R$ do
7:         if path($\alpha, i$) = path($\beta, j$) and type($\alpha$) = type($\beta$) then
8:             $dom \leftarrow$ dominant($\alpha, dom$)
9:     if $dom = \perp$ then add $(a, b, \perp)$ to $T_S$
10:    else if $dom \neq i$ and setter($dom, i$)) exists then
11:        add ($a, b$, setter($dom, i$)) to $T_S$
```

We now describe the approach of detecting feasibility by using the information pertaining to held locks, consistent locks and the associated dominant locks. Algorithm 4 takes as input the set of synthesized PC-R pairs (T) from the previous phase and outputs the feasible PC-R pairs in the set (T_S). For every ordered pair (a, b) in T, it gets the consistent locks for the first element a and the held locks for the second element b. The consistent lock is necessary for the first element because it locks p and c accesses whereas held locks suffices for the second element. There can be multiple consistent and held locks. Therefore, we identify the most dominant lock that guards all the accesses (lines 5 - 8). The absence of a dominant lock suggests that it is possible to interleave the PC-R pair access appropriately. Therefore, we add the corresponding PC-R pair to the set of feasible pairs (line 9). Furthermore, if the dominant lock is not the same as the containing object and setter (defined in Section 3.1) exists to change the containing object so that the dominating locks are broken, the PC-R pair also becomes feasible (lines 10 - 11).

We illustrate this on the running example from Figure 5 and Table 2. T is given by $\{(d_3, d_3)\}$, H_C is $\{i_0\}$ and H_R is $\{i_0, i_1\}$, the accessed field is `count`, the containing object is i_1, and the dominant lock (dom) is i_0. As there is a setter `setF` such that i_0.`f` (represented by i_1) can be set by invoking the method on i_0, the pair under consideration becomes feasible and the next phase attempts to synthesize a multithreaded test corresponding to this pair – invocation of method `foo` by two threads with appropriate objects.

```
class A {
  ...
  C g = new C();

  void sync bar() {
    C y = this.g;
    int size = y.getSize();
    y.setSize(size+1);
  }

  void zee () {
    C y = this.g;
    y.incSize();
  }
}
```

```
class C {
 int size = 0;

 int sync getSize() {
   return size;
 }

 void sync setSize(int s) {
   this.size = s;
 }

 void sync incSize() {
   this.size++;
 }
}
```

Figure 8: Modified running example. The ... in `A` represents the definitions of `f`, `foo` and `setF` from Figure 5.

We modify the running example to explain when the ordered pairs can become infeasible. Figure 8 presents the additional methods in class `A` and also presents the implementation of class `C`. When two objects `a′` and `a″` of type `A` are created and methods `bar` and `zee` are invoked on them, the generated variable lock dependencies are shown in Table 3. It shows the accesses of the field `g` and `size` from these two methods and the corresponding held and consistent locks after applying Algorithm 1.

Table 3: D: set of tuples after executing `a′.bar()` and `a″.zee()` where i_2 = `a′`, $i_3 = i_2$.g, i_4 = `a″`, $i_5 = i_4$.g.

Instruction	Variable-lock dependency	ID
read(this,g)	$\langle i_2, \mathtt{g}, \perp, \mathtt{R}, \{i_2\}, \{i_2\}\rangle$	d_4
read(y,size)	$\langle i_3, \mathtt{size}, \perp, \mathtt{R}, \{i_2, i_3\}, \{i_2, i_3\}\rangle$	d_5
write(y,size)	$\langle i_3, \mathtt{size}, \mathtt{R}, \mathtt{W}, \{i_2, i_3\}, \{i_2\}\rangle$	d_6
read(this,g)	$\langle i_4, \mathtt{g}, \perp, \mathtt{R}, \{\}, \{\}\rangle$	d_7
read(y,size)	$\langle i_5, \mathtt{size}, \perp, \mathtt{R}, \{i_5\}, \{i_5\}\rangle$	d_8
write(y,size)	$\langle i_5, \mathtt{size}, \mathtt{R}, \mathtt{W}, \{i_5\}, \{i_5\}\rangle$	d_9

When PC-R pairs are synthesized for the tuples shown in Table 3 and Table 2, we get the five pairs shown in the second column of Table 4. The associated method invocations are also given in the first column of the table. We have already discussed the feasibility of (d_3, d_3) above. The second pair (d_6, d_6) is not feasible because there is a guard lock **A.this** (i_2) when method `bar` is invoked. Therefore, unless there is a mechanism to set the field `this.g` (i_3), the violation on field `size` in it is not possible. Because there is no setter for `this.g` other than the constructor, the pair becomes infeasible.

Table 4: Feasibility Scanner execution for modified running example from Figure 8. F represents feasibility of PC-R pair.

Method Pairs	PC-R Pairs	$\langle i, j\rangle$	dom	F	Reason	
					Guard	Setter
foo,foo	(d_3, d_3)	$\langle i_1, i_1\rangle$	i_0	✓	A.this	setF
bar,bar	(d_6, d_6)	$\langle i_3, i_3\rangle$	i_2	✗	A.this	-
bar,zee	(d_6, d_9)	$\langle i_3, i_5\rangle$	$\perp$	✓	-	-
zee,bar	(d_9, d_6)	$\langle i_5, i_3\rangle$	i_5	✗	C.this	-
zee,zee	(d_9, d_9)	$\langle i_5, i_5\rangle$	i_5	✗	C.this	-

The third pair (d_6, d_9) is feasible because there is no lock that is consistently held during the invocation of `bar` that is also held during the write in incSize. The only held lock is i_5 whose type is C and the consistent lock i_2's type is A. Therefore, line 7 in Algorithm 4 fails causing the pair to be feasible. In other words, the write to size in incSize from one thread can potentially interleave between the two accesses of size in bar from a different thread. Interestingly, the reversal of this pair (d_9, d_6) is infeasible because the consistent lock is given by i_5 whose type is C and the held lock (when size is written in setSize from bar) is i_3 whose type is also C. Because the consistent lock is the same as the object whose field is being updated, the test of $dom \neq i$ fails at line 10 in the algorithm making this pair infeasible. The infeasibility of the fifth pair can be explained on similar lines.

3.6 Test Generator

Every PC-R pair that is output by the previous phase is processed to synthesize a multithreaded test. The pair references two methods (one method corresponding to the PC accesses, and the other method corresponding to the R access) that need to be invoked by two threads to expose a potential atomicity violation. However, recall that the PC accesses can happen on $o.f$ and the R access can happen on $o'.f$ (see Section 3.4). Therefore, the parameters to these invocations should be such that the accesses are to the shared location. Synthesizing such parameters and writing a test is addressed in this phase.

We use the sequential tests to generate the parameters for the method invocations. This is because our analysis has already witnessed invocations to the methods and objects of the necessary types can be generated. This is achieved by re-executing the sequential test upto the required library invocation, suspending the execution and collecting the actual parameters that will be passed to the invocation. This results in the collection of objects that are instantiated and driven to the required state appropriately. For each PC-R pair, we execute the sequential tests twice to reach the corresponding method invocations and collect the necessary objects.

Algorithm 5 Test Generator

```
 1: for each $(a, b, \sigma) \in T_S$ do        // $\sigma$ is the setter
 2:     $i \leftarrow$ symbol($a$); $j \leftarrow$ symbol($b$)
 3:     $m_a \leftarrow$ getMethod($i$) ; $m_b \leftarrow$ getMethod($j$)
 4:     $O_a \leftarrow$ collectObjects($m_a$);
 5:     $O_b \leftarrow$ collectObjects($m_b$)
 6:     $O_{sa} \leftarrow$ collectSetterObjects($i$);
 7:     $O_{sb} \leftarrow$ collectSetterObjects($j$);
 8:     shareObjects($O_{sa}, O_{sb}, O_a, O_b$)
 9:     Invoke setter methods sequentially with objects in $O_{sa}$
10:     Invoke setter methods sequentially with objects in $O_{sb}$
11:     Invoke $m_a$ with $O_a$ and $m_b$ with $O_b$ from distinct threads
```

Algorithm 5 depicts the outline where for every PC-R pair, we obtain the two methods that need to be invoked concurrently (lines 2-3). Subsequently, we execute the relevant sequential tests upto the method invocation and collect the objects for the invocation (lines 4-5). The collected objects cannot be reused directly because these invocations need not necessarily access the *same* memory locations. For example, if the field of the receiver is accessed in the two method invocations, the sequential tests may be invoking the methods on two different objects. On the other hand, blindly sharing the required objects can prevent exposing the defect. This is because there may be dominating locks before the access and when the methods are invoked concurrently, interleaving may become infeasible.

We use the methods from the sequence returned by the setter primitive (described in Section 3.1) to drive the corresponding objects to the required state. These setters are already recorded for each pair and available as σ. The objects for invoking the methods in the sequence returned by setter is achieved by re-executing the sequential test appropriately (lines 6-7). Subsequently, we share the objects between the two methods corresponding to the PC-R pair (line 8). If the setter methods require other parameters, the corresponding parameter values in the sequential test are used accordingly. Essentially, if there is an atomicity violation on $o.f_1.f_2 \ldots f_n$, then we need to have the same reference to the field from different object instances. After executing the setter methods which will drive the objects to the desired states (lines 9-10), we invoke the methods corresponding to the PC-R pair from two distinct threads (line 11). The entire process of executing sequential tests, collecting objects, sharing them appropriately and then concurrently executing the methods forms the multithreaded test.

We will explain the process for the running example and use the PC-R pair (d_3, d_3) (see Table 4). The sequential test under consideration is

```
A a = new A(); B b = new B();
a.setF(b); a.foo();
```

The corresponding symbols are $\langle i_1, i_1 \rangle$ and the associated methods are foo and foo respectively. The necessary objects for invoking the methods are obtained, $O_a = \{a_1\}$ and $O_b = \{a_2\}$, where the sequential test is invoked twice and executed until the invocation of foo with a_1 and a_2 being the object instances in the executions respectively. setter(i_0, i_1) returns {setF} and needs to be executed in the context of each method. Therefore, the setter objects are collected as $O_{sa} = \{a_3, b_3\}$, and $O_{sb} = \{a_4, b_4\}$. These instances are obtained by re-executing the sequential test (subscript denotes the run number). Invocation of shareObjects at line 8 modifies the sets to $O_{sa} = \{a_1, b_3\}, O_{sb} = \{a_2, b_3\}, O_a = \{a_1\}, O_b = \{a_2\}$. We invoke a_1.setF(b_3) and a_2.setF(b_3) at lines 9 and 10 respectively. This results in the field f of *distinct* objects a_1 and a_2 referencing b_3. Two distinct threads invoke methods foo on a_1 and a_2 respectively at line 11. This is the required test case which can be analyzed by an existing atomicity violation dynamic detector [22, 23, 32].

4. EXPERIMENTAL VALIDATION

We have implemented Intruder using the soot bytecode instrumentation framework [34] and evaluated it on many open source multithreaded Java libraries, that include thread-safe classes. We perform the experiments on Ubuntu-14.04 desktop running on a 3.5 Ghz Intel Core i7 processor with 16GB RAM. Table 5 presents the information pertaining to the benchmarks used for our experiments. colt is a high performance scientific computing library, openjdk is the Java development kit, Carbonado is an extensible, high performance persistence abstraction layer, CometD is a scalable HTTP-based event routing bus, eXo is a open-source social-collaboration software, Batik is a toolkit for applications to use images in SVG format, and OpenNLP is a machine learning based toolkit for processing natural language text. The classes analyzed in the benchmarks are shown in the table. We choose these classes based on whether the class is either declared thread-safe or contains *synchronized*/*volatile* keyword in its implementation. We studied the effectiveness of Intruder in synthesizing atomicity violation revealing tests and also compared with other multithreaded test synthesizers [24, 26, 28]. We used an atomicity violation detector based on CTrigger [23] to analyze the executions of the synthesized tests.

Table 6 presents the information on the number of synthesized tests and the number of true atomicity violations present in the ref-

Table 5: Benchmark Information.

Benchmark	Version	Class Name	ID
Colt	1.2.0	DynamicBin1D	C1
OpenJdk	1.7	StringBuffer	C2
		Vector	C3
Carbonado	1.2.3	SkipCursor	C4
Cometd	2.7.0	TimesyncClientExtension	C5
eXo	3.8.2	ApplicationStatistic	C6
		PortalStatistic	C7
Batik	1.7	CompositeGraphicsNode	C8
OpenNLP	1.5.3	PerformanceMonitor	C9

erenced classes. The number of methods in the classes varies from 4 to 50. We build a sequential seed testsuite that invokes every method in the class once and supply random objects based on the type of the parameters to the invocations. The number of lines of code in each class is given which correspond only to the starting point of the method invocation. These invocations invoke methods from other classes in the package.

Table 6: Experimental Results. T: Unique PC-R pairs, T_S: Number of synthesized tests, A: number of atomicity violations, TP: True Positives.

Class	|M|	LoC	Time(s)	Pairs	$\|T\|$	$\|T_S\|$	|A|	|TP|
C1	35	313	33.9	269	44	11	33	24
C2	50	239	11.1	16	8	8	16	8
C3	43	431	12.7	10	6	6	12	6
C4	4	47	3.1	14	3	3	9	0
C5	7	60	3.6	2	1	1	2	2
C6	7	48	1.5	1	1	1	1	1
C7	6	45	1.4	1	1	1	1	1
C8	40	596	15.5	44	8	7	58	36
C9	6	92	2.4	2	2	2	1	1
Total			85.2	359	74	**40**	133	**79**

Executing the sequential seed testsuite and deploying Intruder results in the synthesis of a number of atomicity violation revealing multithreaded tests in less than two minutes. This demonstrates the scalability of our approach in that it intelligently invokes the relevant combination of methods with appropriate objects from a large state space. The number of PC-R pairs that are synthesized for all the classes is 359 which includes a few redundant pairs. The redundancy corresponds to multiple accesses of fields due to loops, similar pairs of accesses (e.g., field accesses with H and H_C being the same for different fields), etc. We eliminate these redundancies and the number of unique PC-R pairs ($|T|$, output of `PC-R Pair Synthesizer`) is 74. Application of the `Feasibility Scanner` reduces the overall number of pairs and triggers the generation of 40 tests. A significant number of the pairs from `C1` are reduced due to the presence of a guard lock (a common lock between the consistently held lock set of p and c and the held lock set of r) and the absence of setter methods to manipulate the internal fields.

The number of feasible PC-R pairs ranges from one to eight and also corresponds to the number of multithreaded tests synthesized by Intruder. These tests help expose 133 possible violations. On careful observation, we find that the number of possible violations is more than the number of tests (e.g., 33 possible violations with 11 tests for `C1`). This is because we had earlier eliminated redundant PC-R pairs. Since, the violations exist on different fields in the classes in the context of the same method invocation, the defects are reported accordingly. Out of the 133 possible defects, manual analysis reveals the presence of 79 true positives (atomicity violations). This is because even if the interleaving is problematic (as shown in Figure 6), all violations do not lead to a user-observable faulty behavior.

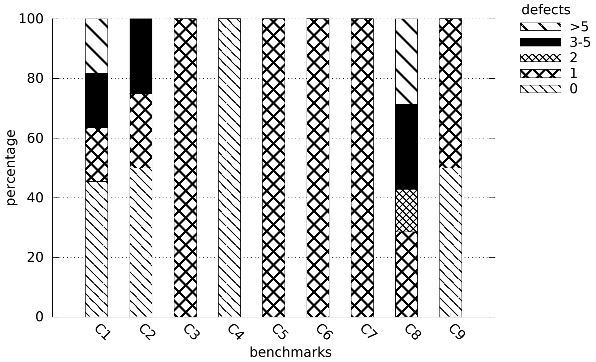

Figure 9: Distribution of tests.

The distribution of tests as a function of the number of detected (harmful) atomicity violations is given in Figure 9. We observe that a minor percentage of the synthesized tests expose only benign violations (reported as zero defects) as mentioned above. In the case of `C1` and `C8`, a few tests expose more than five violations. Moreover, for test cases that expose multiple violations, potentially a single fix can help eliminate multiple violations. The implementation and the raw experimental data (synthesized tests and bugs) are publicly available[4] and we refer the reader to it for more details.

Table 7: Comparison with other test synthesizers. T_O: Generated test cases, TP: True positive atomicity violations.

Class	ConTeGe		Omen		Narada	
	$\|T_O\|$	|TP|	$\|T_O\|$	|TP|	$\|T_O\|$	|TP|
C1	9k	0	10	0	6	0
C2	1.3k	0	0	0	0	0
C3	31k	0	15	0	0	0
C4	274	0	0	0	0	0
C5	174	0	0	0	0	0
C6	172	0	0	0	0	0
C7	165	0	0	0	0	0
C8	10k	0	0	0	4	20
C9	1.2k	0	0	0	0	0
Total	53k	0	25	0	10	20

We also studied the effectiveness of ConTeGe [24], Omen [26] and Narada [28] in enabling the detection of atomicity violations and present our findings in Table 7. Because ConTeGe employs a randomized approach, we set a limit on the number of suffix generations, an internal parameter to bound the tests. The tool is self-contained and reports any violations without requiring an external detector to detect violations. For the sake of comparison, we record the number of tests generated internally by ConTeGe. Unfortunately, it was unable to detect any atomicity violations even after generating 53k tests and executing for five hours approximately. While running it longer will likely expose more violations, this demonstrates the drawbacks of employing randomization to synthesize tests. Omen synthesized 25 multithreaded tests corresponding to `C1` and `C3` which reveal a few deadlocks but does not expose any atomicity violations. When we apply Narada, it synthesizes 10 multithreaded tests. While the six tests synthesized for `C1` do not reveal any atomicity violations (detects a few races), the four tests synthesized for `C8` expose 20 harmful violations. This is coincidental because the concurrent method invocations expose

[4] `http://www.csa.iisc.ernet.in/~sss/tools/intruder`

races apart from atomicity violations. Interestingly, this makes the remaining 59 defects due to INTRUDER more compelling. This is because even though precaution is taken to avoid races by guarding the fields appropriately, the code is not robust enough to avoid atomicity violations. OMEN and NARADA fail to synthesize tests for many classes due to the absence of potential deadlocks and races in these classes respectively.

4.1 Discussion

In this paper, our primary focus is on detecting violations on individual variables. The detection of atomicity violations due to multiple variables requires an understanding of the set of variables that need to be handled atomically [17]. Designing an approach to synthesize tests to detect such atomicity violations is left for the future.

We do not consider data flow constraints as part of our synthesis. Consequently, this can potentially suppress the synthesis of a few multithreaded tests. More specifically, if a seed test has multiple instances of an object required by the multithreaded test, we choose one object instance randomly[5]. Also, the setter could potentially change the overall control flow of the methods under test due to unintended state modifications. Despite these design choices, our experimental results on the benchmarks show that INTRUDER is able to synthesize useful tests.

In our experiments, we used a sequential seed testsuite that invokes every method in the class once with random objects as parameters to the corresponding invocations. We adopted this approach to demonstrate that even with such a simple sequential testsuite, INTRUDER is able to synthesize many atomicity violation revealing multithreaded tests. Obviously, the quality of the multithreaded tests is dependent on the input sequential seed testsuite. For example, if the code pertaining to an atomicity violation is not covered by the sequential test, our approach will be unable to synthesize a multithreaded test. Apart from developing a manual sequential seed testsuite as described above, we can also generate these testsuites using automatic test generators (RANDOOP [21], EVOSUITE [10]). Another possibility is to use the testsuites that accompany the implementation of the libraries. Previous experience [26] shows that using tests generated using automated approaches or manually designed by third-party developers is less effective. Nevertheless, we plan to elaborately study the impact of employing various sequential seed testsuites on synthesizing effective multithreaded tests as part of future work.

5. RELATED WORK

Synthesizing multithreaded tests to detect bugs in thread-safe libraries has been helpful in identifying many previously unknown concurrency bugs [24, 26, 28]. In [24], the authors propose a *randomized* approach for generating multithreaded tests and run the test multiple times. If the concurrent execution results in an exception and the corresponding sequential execution succeeds, then a potential thread-safety violation is detected. Even though, this approach can detect atomicity violations along with races and deadlocks, a fundamental drawback of the approach is the randomized nature of generating tests in an extremely large state space. BALLERINA [20] generates multithreaded tests by concurrently invoking random methods on the object under test. While useful in detecting concurrency bugs, the approach is not suitable for synthesizing complex tests. For example, the test shown in Figure 2 cannot be synthesized by BALLERINA as concurrent invocations on different objects are required. CONSUITE [33] is another effective tool for generating unit tests for concurrent classes but its definition of concurrency coverage may not cover all possible atomicity violations.

Previously, we have designed and implemented multithreaded test synthesizers for detecting deadlocks [26, 27] and races [28]. Unfortunately, these approaches cannot be used to synthesize tests to detect atomicity violations. This is because the properties that are being analyzed during a sequential execution are directed towards deadlocks and races respectively. In Section 4, we also compare INTRUDER with CONTEGE [24], OMEN [26] and NARADA [28], and show that the synthesized tests by these tools are ineffective in detecting atomicity violations.

There are a number of dynamic analyses that have been designed to detect atomicity violations. ATOMIZER [6] integrates the ideas from ERASER [29] and Lipton's theory of reduction [15] to identify possible interference by other threads in an atomic block. VELODROME [8] is a sound and complete atomicity checker that reports an error *iff* the observed trace is not conflict serializable. CTRIGGER [23] uses minimum execution perturbation to expose low probability interleavings. DOUBLECHECKER [2] uses two phases of dynamic analyses to enable precise detection of atomicity violations efficiently. PENELOPE [32] explores alternative schedules effectively to detect atomicity violations. ATOMFUZZER [22] deploys fuzzing techniques to detect the bugs. There are numerous other techniques that are designed that address the problem of *detecting* atomicity violations. In this paper, we propose an approach for *synthesizing* tests that can form the input to any one of these techniques.

Static analysis techniques [18, 1] have also been applied to detect concurrency bugs including atomicity violations. Employing a dynamic technique provides an opportunity for the developer to have a reproducer [22] which can enable reasoning about the correctness of the reported defect. Our approach helps generate the necessary tests for such dynamic analyses. Other testing frameworks like CHESS [19] that systematically explore the state space for detecting bugs can greatly benefit from our synthesized tests.

Many approaches have been designed for automatically generating tests [11, 31, 4, 5, 21]. These approaches have been developed in the context of sequential programs that deploy a combination of concolic testing and feedback directed test generation. In this paper, we explore the synthesis of multithreaded tests by analyzing sequential test executions. Frameworks for ease of writing multithreaded tests [13, 25] address a different dimension of detecting bugs in multithreaded programs. These frameworks provide an easier interface for developers to specify the various interleavings to be tested and help eliminate the non-determinism from the multithreaded tests. The tests generated by our implementation can be subsequently refactored into these frameworks for regression testing.

6. CONCLUSIONS

We motivate the need for effective multithreaded tests to detect atomicity violations in thread-safe libraries and components. We design a lightweight approach that analyzes sequential executions to synthesize defect-revealing multithreaded tests. We incorporate the design as part of a tool, named INTRUDER, and demonstrate that it is able to synthesize 40 tests for multiple open-source `Java` libraries and helps in the detection of 79 (including previously unknown) harmful atomicity violations.

[5] If two method invocations require the same object to be passed as parameters, then our approach will use the same *random* object for both invocations.

7. REPLICATION PACKAGE

The implementation and experimental results of Intruder[6] has been successfully evaluated by the Replication Packages Evaluation Committee and found to meet expectations. The tool is packaged as a bootable VM image and is structured as follows:

```
├── benchs
├── classes
├── lib
├── saved_output
│   ├── C1
│   └── ...
├── saved_output_vm
│   ├── C1
│   └── ...
├── scripts
├── test
│   ├── test1
│   └── ...
└── README.txt
```

- **benchs** contains the benchmarks used for the evaluation of Intruder along with the sequential seed tests written by us.
- **lib** contains the supporting jar files used by the tool and the benchmarks.
- **tests** contains simple test cases which correspond to various aspects of Java multithreading.

7.1 Setup

There are two environmental variables, INTRUDER_CLASSPATH and CTRIGGER, that need to be set appropriately. These variables are initialized by executing `source ./scripts/init.sh`. By default, the atomicity violation detection is turned off. The default setting runs faster as it only synthesizes the multithreaded tests. The atomicity violations exposed by the generated tests can be detected by setting CTRIGGER to ON.

7.2 Usage

Simple Test:

To test the tool, a simple test can be used by executing `sh ./scripts/test2.sh`. The source files related to the test are contained in `tests/test2`. The library implementation provided for this test contains one atomicity violation which should be detected with the help of a test case that is generated. The generated output is placed in `./output/test2`.

- The total number of generated tests and the detected atomicity violations are reported in `summary.txt`.
- The generated multithreaded tests are named: `TestDriver<testNum>.java`, where `<testNum>` is the index of each test case.
- The output of atomicity violation detection for test case `TestDriver<testNum>.java` will be placed in `TestCase_<testNum>_output.txt`.

Benchmarks:

- To test a class `C<id>`, execute `sh ./scripts/C<id>.sh`. The output will be placed in `./output/C<id>`.
- All the classes can be tested with a single command by executing `sh ./scripts/benchmark.sh`.

Thread safety violations might be witnessed during the execution of the generated tests and can lead to exceptions. The number of atomicity violations detected (when CTRIGGER=ON) is dependent on the execution schedule and can vary from the numbers reported in the paper. Also, the numbers reported in this paper are on bare metal and the tool runs slower on the VM. Therefore we provide the saved output of running the benchmarks on bare metal in `saved_output`. We also provide the output of running the benchmarks on the VM in `saved_output_vm`.

8. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their feedback which helped improve the presentation of the paper. We thank Anuta Mukherjee for implementing the atomicity violation detector used in our experiments. We are grateful to Google India and Microsoft Research India for providing travel support. This work is partially supported by the Ministry of Human Resources and Development, Government of India.

9. REFERENCES

[1] N. Ayewah and W. Pugh. The Google FindBugs Fixit. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 241–252, 2010.

[2] S. Biswas, J. Huang, A. Sengupta, and M. D. Bond. Doublechecker: Efficient sound and precise atomicity checking. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 28–39, 2014.

[3] J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 35–46, 1999.

[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] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. In *Proceedings of the 12th International Conference on Model Checking Software*, SPIN'05, pages 2–23, 2005.

[6] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04.

[7] 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, 2009.

[8] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08.

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

[9] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 338–349, 2003.

[10] G. Fraser and A. Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419, 2011.

[11] 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, 2005.

[12] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[13] 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.

[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, 2009.

[15] R. J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, Dec. 1975.

[16] P. Liu, O. Tripp, and X. Zhang. Flint: Fixing linearizability violations. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, pages 543–560, 2014.

[17] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 103–116, 2007.

[18] 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.

[19] 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, 2007.

[20] A. Nistor, Q. Luo, M. Pradel, T. R. Gross, and D. Marinov. Ballerina: Automatic generation and clustering of efficient random unit tests for multithreaded code. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 727–737. IEEE, 2012.

[21] 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.

[22] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 135–145, 2008.

[23] S. Park, S. Lu, and Y. Zhou. CTrigger: Exposing atomicity violation bugs from their hiding places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, 2009.

[24] 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, 2012.

[25] W. Pugh and N. Ayewah. Unit testing concurrent software. In *In ASE*, 2007.

[26] 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.

[27] M. Samak and M. K. Ramanathan. Omen+: A precise dynamic deadlock detector for multithreaded java libraries. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 735–738, 2014.

[28] M. Samak, M. K. Ramanathan, and S. Jagannathan. Synthesizing racy tests. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 175–185, 2015.

[29] 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.

[30] 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.

[31] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, 2005.

[32] F. Sorrentino, A. Farzan, and P. Madhusudan. Penelope: Weaving threads to expose atomicity violations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 37–46, 2010.

[33] S. Steenbuck and G. Fraser. Generating unit tests for concurrent classes. In *Proceedings of the Sixth International Conference on Software Testing, Verification and Validation (ICST)*, pages 144–153. IEEE, 2013.

[34] 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.