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  

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 libraries. These libraries are structured such that concurrent invocation of methods from multiple clients always corresponds to some linearization of the associated invocations [24][6][12]. Moreover, such libraries limit the use of synchronizations to provide better performance [9] 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][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 [23][2] and reproducibility [2] 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.

CosTeGr [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 [25]. 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 test suite. We maintain a record of the syn-
chronized blocks, the objects on which the locks are held, the field
accesses and their corresponding types (read or write) within each
element. We construct a set of properties that need to be satisfied in a
sequential execution to help identify whether a pair of consec-
tutive field accesses can become potential candidates for atomicity
violation. We analyze the recorded information and identify the po-
tential candidate pairs of consecutive field accesses. Furthermore,
we identify a remote 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 in-
voked concurrently. The parameters to the invocations should sat-
ify 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 af-
ter driving them to the required state. The synthesized tests can be
used by any of the atomicity violation dynamic detectors [22][23]
[24] to detect the violations.

We have implemented a tool, named Intruder, that incorporates these
effects and evaluated it on a number of open-source multi-
threaded 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 viola-
tion detector based on CSTagger [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 neg-
ligible memory overhead. We also compare Intruder with Con-
TeGe [24], Omex [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 en-
  able detection of atomicity violations in library code by using the
  implementation of the library under consideration and a sequen-
tial 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 gener-
  ate the objects for the multithreaded execution.

- We provide a detailed discussion of the design and implemen-
tation 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], Omex [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 imple-
mented 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 hold-
ing any additional locks.

DynamicBin1D.java:

```java
581 synchronized DynamicBin1D sampleBootstrap(
582 DynamicBin1D other, ..., BinBinFunction1D function) {
583 // since "resamples" can be quite large, we care
584 about performance and memory
585 int maxCapacity = 1000;
586 int s1 = size();
587 int s2 = other.size();
589 DynamicBin1D sample2 = new DynamicBin1D();
590 cern.colt.buffer.DoubleBuffer buffer2 =
591 sample2.buffered(Math.min(maxCapacity,s2));
599 // resampling steps
600 for (int i=0; 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));
611 }
438 synchronized void sample(int n, ...
449 true?Replacement, ...) {
452 for (int i=n; --i >= 0; ) {
459 else { // with
465 Uniform uniform = new Uniform(randomGenerator);
452 for (int i=n; --i >= 0; ) {
453 buffer.add(this.elements.getQuick((
454 uniform.nextDoubleFromTo(0,s-1)));
456 } 224 } 223 return elements[index];
465 } 224 }
```

DoubleArraylist.java:

```java
217 */ You should only use this method when you are
222 absolutely sure that the index is within bounds./
223 public double getQuick(int 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 implemen-
tation of getQuick from DoubleArraylist is also shown in
the figure where the comment above the implementation emphat-
ically places the burden of using the appropriate index on the
caller of the method.

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

Figure 1: Motivating example.
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 \textsuperscript{[24, 26, 28]}, none of the tools are able to synthesize this multithreaded test automatically. This is because \textsuperscript{[24] is random (and the maximum time to generate tests was set as five hours), \textsuperscript{[26] is searching for deadlocks and \textsuperscript{[28] is searching for races.

using the analysis described in the paper, we are able to automatically synthesize the multithreaded test shown in Figure\textsuperscript{[2]}. The implementation of the library along with a sequential seed test shown in Figure\textsuperscript{[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 defects 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

The architecture of our tool, named I\textsuperscript{ntruder}, which generates multi-threaded tests to enable detection of atomicity violations is presented in Figure\textsuperscript{[3]}. 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 pairs. 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. Feasibility 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., AromFuzz\textsuperscript{[22]}, CTrigger\textsuperscript{[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

\textsuperscript{1}PC-R stands for previous, current and remote \textsuperscript{[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.

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 (l₀, l₁, l₂). Therefore, symbol(a), symbol(a.f) and symbol(b) will return l₀, l₁ and l₂ 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 aliased 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 iₙ, we define a set of relevant locks {l₀, l₁, ..., lₙ} where iₙ = symbol(o.f₁.f₂...fₙ), l₀ = 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:

```java
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
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 \( S = \text{Symbol} \times N \)
- \( H_C \in 2^{\text{Symbol}} \), \( RL : \text{Symbol} \mapsto 2^{\text{Symbol}} \)
- \( \text{AI} : \text{Symbol} \mapsto 2^F \), where \( F : \text{Field} \mapsto (\{R, W, \bot\} \times \{N \cup \bot\}) \)
- \( D \in 2^F \), where \( P = \text{Symbol} \times \text{Field} \times (\{R, W, \bot\} \times \{R, W\}) \times H \times H_C \)

Recall from Section 3.1 that a fresh symbol is associated with each parameter. These symbols are from \( S = \text{Symbol} \{i_0, i_1, \ldots\} \). 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 accessed field, the kind of the previous access, the kind of the field (include its most recent access kind (read, write or unknown) and the timestamp of instructions. Before \( \text{recordAccess} \) (see Algorithm 2), \( H \) and \( \delta \) are initialized to empty and zero respectively. When \( \text{recordAccess} \) occurs, \( H \) and \( \delta \) are updated accordingly (lines 9-14). If the instruction is an access to a controllable variable, then \( \text{recordAccess} \) is invoked with the appropriate access kind (lines 5-8).

Algorithm 1: \text{genVarLockDependency}

1: \( H \leftarrow \emptyset; \delta \leftarrow 0 \)
2: Initialize \( \text{AI} \) 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: \( \text{recordAccess}(\text{var}, \text{field}, R, H) \)
7: else if \( s \) is write(var, field) and controllable(var) then
8: \( \text{recordAccess}(\text{var}, \text{field}, W) \)
9: else if \( s \) is lock(var) then
10: Increment \( \delta \) by one
11: Add \((\text{symbol}(\text{var}), \delta)\) to \( H \)
12: else if \( s \) is unlock(var) then
13: Increment \( \delta \) by one
14: Remove the element \((\text{symbol}(\text{var}), \delta)\) 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 \( \delta \) reset to 0. We also initialize \( \text{AI} \) 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 \( \text{recordAccess} \) 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 \( \text{AI} \) (line 2). For all the currently held locks, we obtain the relevant held locks \( (H_R) \) based on \( RL \) and \( H \). 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: \text{recordAccess}

Input: \( \text{var} \) dereferenced variable, \( \text{field} \) accessed in \( \text{var} \), \( \tau \) access kind.
1: \( i \leftarrow \text{symbol}(\text{var}); H_e \leftarrow \emptyset \)
2: \((\tau_p, \delta_p) \leftarrow \text{AI}[i][\text{field}] / \text{previous access kind and timestamp}
3: \( H_g \leftarrow \text{locks}(H) \cap RL[i] \)
4: for each lock \( \ell \) in \( H_g \) do
5: if timestamp(\( i, \tau \)) \( \leq \delta_p \) then
6: \( H_C \leftarrow H_{C} \cup \text{symbol}(\ell) \)
7: \( D \leftarrow D \cup (i, \text{field}, \tau_p, \tau_R, H_C) \)
8: \( \text{AI}[\text{field}] \leftarrow (\tau, \delta) \)

We now illustrate the process when \( \text{a.foo()} \) 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. \( \text{AI} \) is initialized to \( \{i_0 \leftrightarrow [f \leftrightarrow (\bot, \bot)], i_1 \leftrightarrow [\text{count} \leftrightarrow (\bot, \bot)]\} \), and \( RL \) is initialized to \( \{\} \). \( H \) and \( \delta \) are initialized to empty and zero respectively.

Table 1: State changes when \( \text{a.foo()} \) is invoked by client.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>( \text{AI} )</th>
<th>( H )</th>
<th>( \delta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{lock}(a) )</td>
<td>( {i_0, 1} )</td>
<td>( 1 )</td>
<td></td>
</tr>
<tr>
<td>( \text{read}(\text{this}, f) )</td>
<td>( {i_0 \leftrightarrow [f \leftrightarrow (R, 1)], i_1 \leftrightarrow [\text{count} \leftrightarrow (\bot, \bot)]} )</td>
<td>( {i_0, 1} )</td>
<td></td>
</tr>
<tr>
<td>( \text{lock}(x) )</td>
<td>( {i_0, 1} )</td>
<td>( {i_1, 2} )</td>
<td></td>
</tr>
<tr>
<td>( \text{read}(x, \text{count}) )</td>
<td>( {i_0 \leftrightarrow [f \leftrightarrow (R, 1)], i_1 \leftrightarrow [\text{count} \leftrightarrow (R, 2)]} )</td>
<td>( {i_0, 1} )</td>
<td></td>
</tr>
<tr>
<td>( \text{unlock}(x) )</td>
<td>( {i_0, 1} )</td>
<td>( {i_1, 4} )</td>
<td></td>
</tr>
<tr>
<td>( \text{lock}(x) )</td>
<td>( {i_0, 1} )</td>
<td>( {i_1, 4} )</td>
<td></td>
</tr>
<tr>
<td>( \text{write}(x, \text{count}) )</td>
<td>( {i_0 \leftrightarrow [f \leftrightarrow (R, 1)], i_1 \leftrightarrow [\text{count} \leftrightarrow (W, 4)]} )</td>
<td>( {i_0, 1} )</td>
<td></td>
</tr>
<tr>
<td>( \text{unlock}(x) )</td>
<td>( {i_0, 1} )</td>
<td>( {i_1, 4} )</td>
<td></td>
</tr>
<tr>
<td>( \text{unlock}(a) )</td>
<td>( \emptyset )</td>
<td>( 0 )</td>
<td></td>
</tr>
</tbody>
</table>

Table 1 presents the updated \( \text{AI} \), \( H \) and \( \delta \) while executing the list of instructions. Before \( \text{lock}(a) \) is executed, \( \delta \) is incremented and \( H \) updated to \( \{i_0, 1\} \). When \( \text{read}(\text{this}, f) \) happens, \( AI \) is updated to reflect the \( \text{read} \) of \( f \) at timestamp equals one. Moreover observe that even though the method \( \text{getCount} \) 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 \( a . f() \). When \( \text{read}(\text{this}, f) \) happens, the symbol corresponding to the variable \( \text{this} \) 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 \( i_0 \).

1\( \text{locks}(H) \) returns the list of locks without the timestamp from \( H \).
When \texttt{write}(x, count) is performed, \texttt{recordAccess} finds a previous read access to \texttt{count} at \(\delta = 2\). Moreover, it obtains the relevant locks for \(\texttt{i}_1\) to be \((\texttt{i}_1, \texttt{i}_0)\) from RL and performs an intersection with the symbols associated with held locks \((\texttt{i}_1, \texttt{i}_0)\) (line 3 of Algorithm 2). When it iterates over these locks, it finds that the lock associated with \(\texttt{i}_1\) is obtained at \(\delta = 4\) which is greater than the timestamp for the previous access. In contrast, it detects that the lock associated with \(\texttt{i}_0\) is obtained at \(\delta = 1\) which is less than the timestamp for the previous field access (\(\delta = 2\)). Hence, the consistent lock set becomes \((\texttt{i}_0)\). In other words, we derive that some object represented by \(\texttt{i}_0\) is held without being released for two consecutive accesses of the field \texttt{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 CTR\textsubscript{3}G\textsubscript{3} [25]. Even though we do not have multiple threads (because the analysis is on sequential executions) unlike CTR\textsubscript{3}G\textsubscript{3}, 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.](image)

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 \texttt{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 \(a.f\) 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: \quad if \texttt{getField}(a) = \texttt{getField}(b) then
4: \quad \quad \(c \leftarrow \texttt{current}(a); \ p \leftarrow \texttt{previous}(a)\)
5: \quad \quad \(r \leftarrow \texttt{current}(b)\)
6: \quad \quad if \(c = R\) and \(r = W\) then
7: \quad \quad \quad if \(p \neq \top\) then add \((a, b)\) to \(T\)
8: \quad \quad \quad else if \(c = W\) then
9: \quad \quad \quad \quad if \(p = R\) and \(r = W\) then add \((a, b)\) to \(T\)
10: \quad \quad \quad \quad 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 \texttt{getField}), it obtains \(p\), \(c\) and \(r\) using the auxiliary functions \texttt{current} and \texttt{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 \(T = \{(d_0, d_1)\}\). This is because the field under consideration is \(A.f\) and the access \(d_0.f\) is held for \(p\) and \(c\) access to \(d_1.f\) 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 \texttt{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 \(i_0.f_0 \ldots f_i\), then path\((x, y)\) will return \(f_0 \ldots f_i\). The auxiliary function \texttt{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 \texttt{dominant} that takes two symbols, finds the associated dereferences and returns the dominating symbol. \(\top\) 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 \(o.f_0, o.f_1, \ldots, o.f_i, \ldots, f_j\) are acquired, and the atomicity violation needs to happen on \(o.f_i, \ldots, f_j\) then all locks up to and including \(o.f_i, \ldots, f_j\) 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 \(o.f_1, \ldots, f_j\) needs to re-
fer to the same object so that the memory location on which the violation needs to happen becomes shared.

![Diagram](image)

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. \( T_f \leftarrow \phi \)
2. for each \((a, b) \in T\) do
3. \( H_c \leftarrow \text{consistentLocks}(a) \); \( H_b \leftarrow \text{heldLocks}(b) \)
4. \( i \leftarrow \text{symbol}(a) \); \( j \leftarrow \text{symbol}(b) \)
5. \( \text{dom} \leftarrow \perp \)
6. for each pair \(a, \beta \) where \(a \in H_c, \beta \in H_b\) do
7. if \( \text{path}(a, i) = \text{path}(\beta, j) \) and \(\text{type}(a) = \text{type}(\beta)\) then
8. \(\text{dom} \leftarrow \text{dominant}(a, \text{dom})\)
9. if \(\text{dom} \neq \perp\) then add \((a, b, \text{setter}(\text{dom}, i))\) exists then
10. else if \(\text{dom} \neq \perp\) and \(\text{setter}(\text{dom}, i)\) exists then
11. add \((a, b, \text{setter}(\text{dom}, i))\) to \(T_f\)

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 \(T_f\). 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 5), we add the corresponding PC-R pair to the set of possible pairs (line 9). The third pair \((d_0, d_0)\) is feasible because there is no lock that is consistently held during the invocation of \(\text{bar}\) that is also held

![Diagram](image)

Figure 8: Modified running example. The \(\ldots\) in \(A\) represents the definitions of \(\ell, \text{foo}\) and \(\text{setF}\) 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 \(a^\prime\) of type \(A\) are created and methods \(\text{bar}\) and \(\text{zee}\) are invoked on them, the generated variable lock dependencies are shown in Table 5. It shows the accesses of the field \(g\) and \(size\) from these two methods and the corresponding held and consistent locks after applying Algorithm 4.

![Table](image)

Table 3: \(D\): set of tuples after executing \(a^* \cdot \text{bar}()\) and \(a^\prime \cdot \text{zee}()\) where \(i_2 = a^*, i_3 = b^*\).

![Diagram](image)

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

![Table](image)

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

The third pair \((d_0, d_0)\) is feasible because there is no lock that is consistently held during the invocation of \(\text{bar}\) that is also held
during the write in incSize. The only held lock is \( i_5 \) whose type is \( C \) and the consistent lock \( i_5 \)'s type is \( A \). Therefore, line 7 in Algorithm 3 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 \( (d_h, d_o) \) is infeasible because the consistent lock is given by \( i_5 \)'s type is \( C \) and the held lock (when size is written in setSize from bar) is \( i_5 \) 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 3.3). 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 up to 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: \( i \leftarrow \text{symbol}(a); j \leftarrow \text{symbol}(b) \)
3: \( m_a \leftarrow \text{getMethod}(i); m_b \leftarrow \text{getMethod}(j) \)
4: \( O_a \leftarrow \text{collectObjects}(m_a) \)
5: \( O_b \leftarrow \text{collectObjects}(m_b) \)
6: \( O_{sa} \leftarrow \text{collectSetterObjects}(i) \)
7: \( O_{sb} \leftarrow \text{collectSetterObjects}(j) \)
8: \( \text{shareObjects}(O_a, O_b, O_{sa}, O_{sb}) \)
9: Invoke setter methods sequentially with objects in \( O_a \)
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 up to 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 \( \sigma \). 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 \((d_1, d_2)\) (see Table 4). The sequential test under consideration is

\[
A \ a = \text{new} \ A(); \ B \ b = \text{new} \ B(); \ a.\text{setF}(b); \ a.\text{foo}();
\]

The corresponding symbols are \( (i_1, i_1) \) and the associated methods are \( \text{foo} \) and \( \text{foo} \) 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 \( \text{foo} \) with \( a_1 \) and \( a_2 \) being the object instances in the executions respectively. \( \text{setter}(i_1, i_1) \) returns \( \{\text{setF}\} \) and needs to be executed in the context of each method. Therefore, the setter objects are collected as \( O_{sa} = [a_1, b_1] \) and \( O_{sb} = [a_1, b_2] \). These instances are obtained by re-executing the sequential test (script denotes the run number). Invocation of \( \text{shareObjects} \) at line 8 modifies the sets to \( O_{su} = [a_1, b_1] \), \( O_{sb} = [a_2, b_1] \), \( O_a = [a_1] \), \( O_b = [a_2] \). We invoke \( a_1.\text{setF}(b_1) \) and \( a_2.\text{setF}(b_2) \) at lines 9 and 10 respectively. This results in the field \( e \) of distinct objects \( a_1 \) and \( a_2 \) referencing \( b_1 \) and \( b_2 \) 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 Crawler [23] to analyze the executions of the synthesized tests.

Table 5 presents the information on the number of synthesized tests and the number of true atomicity violations present in the ref-
erenced classes. The number of methods in the classes varies from 4 to 50. We build a seed sequential testsuite that invokes every method in the class once and supply random objects based on the type of the parameters to the invocations. The numbers 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 | | | | | |
|-------|-------|-------|-------|-------|
| C1    | 35    | 315   | 339   | 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     | 2     | 2     |       |
| C6    | 7     | 48    | 1.5   | 1     | 1     | 1     | 1     |       |
| C7    | 6     | 45    | 1.4   | 1     | 1     | 1     |       |       |
| C8    | 40    | 596   | 15.5  | 44    | 8     | 7     | 58    | 36    |
| C9    | 6     | 92    | 2.4   | 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 $H_e$ being the same for different fields), etc. We eliminate these redundancies and the number of unique PC-R pairs ($|P|$ 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.

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[26] and we refer the reader to it for more details.

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

<table>
<thead>
<tr>
<th>Class</th>
<th>ConTeGe</th>
<th>Omen</th>
<th>Narada</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$</td>
<td>P</td>
<td>$</td>
</tr>
<tr>
<td>C1</td>
<td>9k</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>C2</td>
<td>1.3k</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C3</td>
<td>31k</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>C4</td>
<td>274</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C5</td>
<td>174</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C6</td>
<td>172</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C7</td>
<td>165</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C8</td>
<td>10k</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C9</td>
<td>1.2k</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>53k</td>
<td>0</td>
<td>25</td>
</tr>
</tbody>
</table>

We also studied the effectiveness of ConTeGe, Omen, and Narada 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 synthesizes 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 redundant PC-R pairs. Since, the violations exist on different fields 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.

Table 5: Benchmark Information.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Version</th>
<th>Class Name</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Colt</td>
<td>1.2.0</td>
<td>DynamicBin1D</td>
<td>C1</td>
</tr>
<tr>
<td>OpenJdk</td>
<td>1.7</td>
<td>StringBuffer</td>
<td>C2</td>
</tr>
<tr>
<td>Carbonado</td>
<td>1.2.3</td>
<td>SkipCursor</td>
<td>C3</td>
</tr>
<tr>
<td>CometD</td>
<td>2.7.0</td>
<td>TimesyncClientExtension</td>
<td>C4</td>
</tr>
<tr>
<td>xFire</td>
<td>3.8.2</td>
<td>ApplicationStatistic</td>
<td>C5</td>
</tr>
<tr>
<td>Portal</td>
<td>1.7</td>
<td>CompositeGraphicsNode</td>
<td>C6</td>
</tr>
<tr>
<td>Batik</td>
<td>1.7</td>
<td>PerformanceMonitor</td>
<td>C7</td>
</tr>
<tr>
<td>OpenNLP</td>
<td>1.5.3</td>
<td></td>
<td>C8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>C9</td>
</tr>
</tbody>
</table>

Figure 9: Distribution of tests.

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. 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 test suite that invokes every method in the class once with random objects as parameters to the corresponding invocations. We adopted this approach to synthesize tests for multithreaded classes. More specifically, if a seed test has multiple instances of an object required by the multithreaded test, we choose one object instance randomly. 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 test suite that invokes every method in the class once with random objects as parameters to the corresponding invocations. We adopted this approach to synthesize tests for multithreaded classes. More specifically, if a seed test has multiple instances of an object required by the multithreaded test, we choose one object instance randomly. 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 test suite that invokes every method in the class once with random objects as parameters to the corresponding invocations. We adopted this approach to synthesize tests for multithreaded classes. More specifically, if a seed test has multiple instances of an object required by the multithreaded test, we choose one object instance randomly. 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 test suite that invokes every method in the class once with random objects as parameters to the corresponding invocations. We adopted this approach to synthesize tests for multithreaded classes. More specifically, if a seed test has multiple instances of an object required by the multithreaded test, we choose one object instance randomly. 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.
7. REPLICATION PACKAGE

The implementation and experimental results of Intruder 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
