Randomized Active Atomicity Violation Detection in Concurrent Programs

Chang-Seo Park
EECS Department, UC Berkeley, CA, USA.
parkcs@cs.berkeley.edu

Koushik Sen
EECS Department, UC Berkeley, CA, USA.
ksen@cs.berkeley.edu

ABSTRACT

Atomicity is an important specification that enables programmers to understand atomic blocks of code in a multi-threaded program as if they are sequential. This significantly simplifies the programmer's job to reason about correctness. Several modern multi-threaded programming languages provide no built-in support to ensure atomicity; instead they rely on the fact that programmers would use locks properly in order to guarantee that atomic code blocks are indeed atomic. However, improper use of locks can sometimes fail to ensure atomicity. Therefore, we need tools that can check atomicity properties of lock-based code automatically.

We propose a randomized dynamic analysis technique to detect a special, but important, class of atomicity violations that are often found in real-world programs. Specifically, our technique modifies the existing Java thread scheduler behavior to create atomicity violations with high probability. Our approach has several advantages over existing dynamic analysis tools. First, we can create a real atomicity violation and see if an exception can be thrown. Second, we can replay an atomicity violating execution by simply using the same seed for random number generation—we do not need to record the execution. Third, we give no false warnings unlike existing dynamic atomicity checking techniques. We have implemented the technique in a prototype tool for Java and have experimented on a number of large multi-threaded Java programs and libraries. We report a number of previously known and unknown bugs and atomicity violations in these Java programs.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Languages, Algorithms, Verification

Keywords

atomicity violation detection, dynamic analysis, random testing, concurrency

SIGSOFT 2008/FSE-16, November 9–15, Atlanta, Georgia, USA

1. INTRODUCTION

Multi-threaded concurrent programs often exhibit wrong behaviors due to unintended interference among multiple threads. Such concurrency bugs are often difficult to find because they typically happen under very specific thread interleavings. Much of the previous work on finding bugs in multi-threaded programs focused on data-race detection. A data-race occurs when two threads concurrently access a memory location and at least one of the accesses is a write. Both dynamic [11, 1, 10, 38, 47, 9, 2, 35, 40] and static [43, 19, 4, 5, 20, 25, 16, 36, 34] techniques have been developed to detect and predict data races in multi-threaded programs.

Unfortunately, the absence of data races is not sufficient to ensure that a program is free of errors. For example, in the following Java implementation of `Consumer`,

```
public class Consumer {
  private LinkedList buffer;

  public synchronized void consume(){
    if(!buffer.isEmpty()){
    // another consumer thread can make the buffer
    // empty causing an atomicity violation
      Object data = buffer.remove();
      System.out.println(((Data)data).value);
    }
  }
}

public class LinkedList ... {
  public synchronized boolean isEmpty(){ ... }
  // Retrieves and removes the head of this queue
  public synchronized Object remove(){ ... }
  ...
}
```

all accesses to the object `buffer` and its fields are protected by the implicit lock associated with the object. The implementation is, therefore, free of data races. However, the `consume` method, which checks if the `buffer` is empty, retrieves and removes the first element from the `buffer`, and prints the element's value, is not *atomic*. This is because, in between checking if the `buffer` is empty and retrieving the first element from the buffer, another consumer thread can make the buffer empty, which will lead to the retrieval of a `null` object by the consumer. This will raise an exception when the consumer tries to print the data.

A stronger *non-interference* property of multi-threaded programs that helps to avoid the above problem is called *atomicity*. A block of code in a multi-threaded program is atomic if for every possible interleaved execution of the program there exists an equivalent execution with the same overall behavior where the atomic block is executed serially, that is, the execution of the atomic block is not interleaved with actions of other threads. Therefore, if a code

block is atomic, the programmer can assume that the execution of the code block by a thread cannot be interfered by any other thread; this helps programmers to reason about atomic code blocks sequentially. For example, in the above code if we can ensure that the method `consume` is atomic, then the execution of `consume` cannot be interfered by any other thread and no exceptions will be raised.

In most modern multi-threaded programming languages, atomicity is indirectly achieved through the use of locks. However, if locks are used incorrectly, a code block may not be ensured to be atomic. The above program is an example of this problem. Our experiments show many examples of atomicity violations in real-world programs that use locks.

In recent years, a number of techniques have been proposed and applied to check atomicity properties. Such techniques include dynamic analysis [48, 21, 49], type systems [32, 23, 24], and model-checking [28, 18]. A problem with type based atomicity checking is that it requires the programmer to write complex annotations that involve the specification of which lock protects which variable and therefore, makes programming harder. Model-checking based techniques, being exhaustive in nature, do not scale for large real-world programs. The dynamic approaches could report many false warnings of atomicity violations. For example, the Atomizer tool [21] reports 97 atomicity violations in experiments out of which only 6 are real atomicity violations. Moreover, being imprecise in nature, most of the dynamic tools require manual inspection to see if an atomicity violation is real or not. Another bigger problem with these tools is that they do not create a concrete concurrent execution exhibiting an atomicity violation. Nevertheless, these tools are very effective in finding atomicity violations because they can predict atomicity violations that could potentially happen during a real execution—for such a prediction, they need to see only one concurrent execution.

We propose a new dynamic technique, called ATOMFUZZER, for precisely finding a particular class of atomicity violations in multi-threaded programs. Specifically, we are interested in finding an atomicity violating locking pattern where a thread p acquires and releases a lock L while inside an atomic block. Another thread p' subsequently acquires and releases the same lock L. Thread p then again acquires the lock L while inside the same atomic block. The preceding example can exhibit this pattern if the method `consume` is declared atomic and if a thread executes the `remove` method between the execution of the `isEmpty` and `remove` methods of `consume` by another thread. We focus on this particular pattern of atomicity violation because it is a very common bug pattern in multi-threaded Java programs and our experiments support this fact. Moreover, we have found that most of the other atomicity violation patterns are due to data races and can be caught using data race detection techniques. A technical explanation of this claim is provided later in the paper.

ATOMFUZZER works as follows. We assume that the user has indicated which code blocks are atomic using annotations. ATOMFUZZER then dynamically checks if the annotated code blocks are indeed atomic. Specifically, ATOMFUZZER performs a random execution of a multi-threaded program by choosing a random thread to execute at every program state. However, unlike simple random testing, ATOMFUZZER looks for the atomicity violation pattern. Whenever ATOMFUZZER discovers that a thread p is inside an atomic block and is about to acquire a lock L that has been previously acquired and released by the same thread inside the same atomic block, ATOMFUZZER warns that there could be a potential atomicity violation. Such a warning would be given by any of the existing atomicity violation tools, such as Atomizer. ATOMFUZZER then tries to check if the warning could be a real atomicity violation by pausing the execution of the thread p and continuing the execution of other threads. If any other thread acquires and releases the lock L while the thread p is waiting, ATOMFUZZER flags a real atomicity violation error because it has created a real atomicity violation scenario. In summary, ATOMFUZZER actively controls a randomized thread scheduler of a multi-threaded program to create real atomicity violations.

We have implemented ATOMFUZZER in a prototype tool for Java. The target program can be annotated (using comments) to indicate that a block of code is atomic. ATOMFUZZER also allows a heuristic to decide which blocks of code should be checked for atomicity in the absence of user provided annotations. The heuristic assumes that all synchronized code blocks and synchronized methods are intended to be atomic. The same heuristic has been previously used by the dynamic atomicity checking tool Atomizer [21]. We have applied ATOMFUZZER to a number of large benchmarks having a total of around 600K lines of code. In all these benchmarks, due to lack of user-provided annotations, we used the above mentioned heuristic to decide which code blocks should be considered atomic. The results of our experiments demonstrate that ATOMFUZZER can detect previously unknown atomicity violations in mature real-world programs including Sun's JDK 1.4.2. For 21 synchronized blocks, ATOMFUZZER found executions showing that these blocks were not atomic. In 14 of these cases, the methods/blocks were intended to be atomic and the reported atomicity violations were real bugs. All of these bugs resulted in unchecked exceptions when the intended atomicity was violated, making it simple-to-determine that they were not false positives. The remaining 7 of these violations are benign because they do not violate the correctness property of the programs. Our results show that the above mentioned heuristic is a reasonable assumption for finding atomicity violation related bugs in 67% of the cases and lead to false error reports in the remaining cases.
We make the following contributions in this paper.

- We identify a special atomicity violation pattern that can be found efficiently and accounts for many atomicity violation related bugs in real-world programs.
- We propose a randomized dynamic analysis technique that can create these atomicity violations.
- We have implemented ATOMFUZZER in Java and applied it to real-world programs. We have discovered previously unknown atomicity violation bugs in these programs.

The features of ATOMFUZZER are listed below.

- **Classifies real atomicity violations from false alarms.** ATOMFUZZER actively controls a randomized thread scheduler so that real atomicity violation scenarios get created with high probability. (In Section 3.2, we explain our claim about high probability through an example and empirically validate the claim in Section 5.2. In general, the high probability claim may not hold for some programs.) *This enables the user of* ATOMFUZZER *to automatically separate real atomicity violations from false warnings, which is otherwise done through manual inspection.*
- **Inexpensive replay of a concurrent execution exhibiting a real atomicity violation.** ATOMFUZZER provides a concrete concurrent execution that exhibits a real atomicity violation. Moreover, it allows the user to *replay* the concrete

execution by setting the same seed for random number generation. Note that an exact replay is possible if we can capture all sources of non-determinism (e.g. data inputs) other than scheduler non-determinism and provide deterministic values for such non-determinism. Once we have removed all non-scheduler non-determinism, the ATOMFUZZER algorithm is deterministic if we fix a seed for the random number generator used by ATOMFUZZER.

- **Demonstrates if an atomicity violation can cause an exception or a crash.** ATOMFUZZER creates an actual atomicity violation scenario. This enables us to discover if the atomicity violation could cause a real exception in a program. For code blocks that are assumed to be atomic by the use of our heuristic, we can confirm that they were actually intended to be atomic if we can cause an exception or a crash when atomicity is violated.

- **No false atomicity violation reports.** ATOMFUZZER gives *no false reports* about atomicity violations because it actually creates an atomicity violation scenario provided that the user annotates the atomic blocks.

- **Embarrassingly parallel.** Since different invocations of ATOMFUZZER are independent[1] of each other, the performance of ATOMFUZZER can be increased linearly with the number of processors or cores.

Despite the various advantages of ATOMFUZZER, it has some limitations. First, being dynamic in nature, ATOMFUZZER cannot detect all atomicity violations in a concurrent program—it detects an atomicity violation if the violation can be produced with the given test harness for some thread schedule. This can be alleviated by combining ATOMFUZZER with a symbolic execution technique. Second, being random in nature, ATOMFUZZER may not be able to create all atomicity violations that could happen with a given input. This problem can be alleviated by running ATOMFUZZER many times. Third, in the absence of user-provided atomicity annotations, ATOMFUZZER has to rely on a heuristic to decide which code blocks are atomic. Therefore, some of the atomicity violation errors that are reported by ATOMFUZZER may not be bugs because the heuristic may not match with the user intentions.

2. ALGORITHM

In this section, we define atomicity formally using a general and simple model of a concurrent system. We then describe the ATOMFUZZER algorithm using this model.

2.1 Background Definitions

We consider a concurrent system composed of a finite set of threads. Each thread executes a sequence of statements and communicates with other threads through shared objects. In a concurrent system, we assume that each thread terminates after the execution of a finite number of statements. At any point of execution, a concurrent system is in a *state*. Let S be the set of states that can be exhibited by a concurrent system starting from the initial state s_0. A concurrent system evolves by transitioning from one state to another state. Let T be the set of all transitions in a system. We say $s \xrightarrow{t} s'$ to denote that the execution of the transition t changes the state s to s'. A transition is always caused by the execution of a statement by a thread.

[1]Note that the algorithm of ATOMFUZZER is not parallel, but for the purpose of testing we have to invoke ATOMFUZZER several times with different random seeds. All such invocation are independent of each other.

Algorithm 1 Algorithm SIMPLEFUZZER

```
1: Inputs: the initial state s_0
2: s := s_0
3: while Enabled(s) ≠ ∅ do
4:     t := a random transition in Enabled(s)
5:     s := Execute(s, t)
6: end while
7: if Alive(s) ≠ ∅ then
8:     print "ERROR: actual deadlock found"
9: end if
```

`Enabled`(s) denotes the set of transitions that are enabled in the state s. `Alive`(s) denotes the set of threads whose executions have not terminated in the state s. A state s is in *deadlock*, if the set of enabled transitions at s (i.e. `Enabled`(s)) is empty and the set of threads that are alive (i.e. `Alive`(s)) is non-empty.

We next describe a simple randomized execution algorithm (see Algorithm 1) to clarify the definitions introduced above. Starting from the initial state s_0, this algorithm, at every state, randomly picks a transition enabled at the state and executes it. The algorithm terminates when the system reaches a state that has no enabled transition. At termination, if there is at least one thread that is alive, the algorithm reports a deadlock.

Given the above model of concurrent programs, we define a *happens-before* relation which is crucial to formally understand atomicity and to describe our ATOMFUZZER algorithm. Central to the definition of the *happens-before* relation is the notion of *independence* of transitions.

DEFINITION 1 (INDEPENDENT TRANSITIONS). *If two transitions in a concurrent system do not interact with each other, then we call them* independent.

For example, a transition denoting the acquire of a lock l_1 by a thread p_1 is independent of a transition denoting the acquire of a lock l_2 by another thread p_2, if l_1 and l_2 are different locks.

DEFINITION 2 (DEPENDENT TRANSITIONS). *Two transitions are said to be* dependent*, if they are not independent.*

Transitions from the same thread are always dependent on each other. Similarly, the acquire or release of a lock by one thread is dependent on the acquire or release of the same lock by another thread. Two accesses (i.e. read or write) of a memory location are dependent if at least one of the accesses is a write.

The execution of a concurrent system can be represented by a sequence of transitions. Specifically, $\tau = t_1 t_2 \ldots t_n$ is a *transition sequence* if there exists states $s_1, s_2, \ldots, s_{n+1}$ such that s_1 is the initial state and

$$s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} \ldots \xrightarrow{t_n} s_{n+1}$$

The *happens-before* relation $\preceq$ for a transition sequence $\tau = t_1 t_2 \ldots t_n$ is defined as the smallest relation such that

1. if t_i and t_j are dependent and $1 \leq i \leq j \leq n$, then $t_i \preceq t_j$, and
2. $\preceq$ is transitively closed.

Thus $\preceq$ is a partial order relation.

In this paper, we use a robust notion of atomicity, called *causal atomicity*, introduced by Farzan et al. [17].

DEFINITION 3 (CAUSAL ATOMICITY). *A block of code B of a thread is* causally atomic *if there is no execution where a transition of another thread happens-after the beginning of B and happens-before another transition that is within the same block B.*

The definition of causal atomicity implies that an atomicity violation occurs in an execution if there are three transitions t_1, t_2, and t_3 such that

1. t_1 and t_3 are transitions of the same thread and are within the same atomic block,
2. t_2 is a transition of another thread, and
3. t_1 happens-before t_2 and t_2 happens before t_3.

The goal of ATOMFUZZER is to create such atomicity violations.

2.2 The randomized active atomicity violation detection algorithm

In this work, we are interested in detecting a special class of causal atomicity violations where the transitions t_1, t_2, and t_3 involved in the violation are acquires of the same lock l. There are three reasons for focusing on this particular pattern. First, our experiments demonstrate that this is a very common atomicity violation pattern and real-world programs often show this buggy pattern. Second, since we focus on lock acquires only, the runtime overhead of ATOMFUZZER is pretty low compared to other tools. Third, we believe that this pattern does not capture some other common situations where t_1, t_2, and t_3 are accesses to the same memory location. However, we argue that a data race over the memory location implies the remaining patterns and can be detected by a race detector. Assume that t_1, t_2, and t_3 are accesses to the same memory location m. If these accesses are not in data race, then each of these accesses is surrounded by a common lock, say l. Let t_1', t_2', and t_3' are transitions denoting the acquire of the lock l before the transitions t_1, t_2, and t_3, respectively. The transitions t_1', t_2', and t_3' then form the above mentioned atomicity violation pattern. Therefore, if there is no race among the transitions t_1, t_2, and t_3, then the resulting atomicity violation pattern is the same as the pattern we are interested in.

In ATOMFUZZER, we only consider three kinds of transitions as described below.

- `AtomicEnter`(p). The execution of an `AtomicEnter`(p) transition by a thread p indicates that the thread has entered an atomic block.
- `AtomicExit`(p). The execution of an `AtomicExit`(p) transition by a thread p indicates that the thread has exited an atomic block.
- `ACQUIRE`(p, L). On the execution of an `ACQUIRE`(p, L) transition, thread p acquires the lock L.

Normally, we will assume that the `AtomicEnter`(p) and `AtomicExit`(p) transitions will be introduced by programmers to annotate the entry and exit of atomic blocks in a concurrent system.

In Java, locks are *re-entrant*, i.e., a thread may re-acquire a lock it already holds. In our algorithm, we ignore a transition `ACQUIRE`(p, L) if p re-acquires a lock L.[2]

[2]This is implemented by associating a usage counter with a lock which is incremented whenever a thread acquires or re-acquires the lock and decremented whenever a thread releases the lock. An `ACQUIRE`(p, L) is considered whenever the thread p acquires or re-acquires the lock L and the usage counter associated with l is incremented from 0 to 1.

Algorithm 2 Algorithm ATOMFUZZER

```
1: Inputs: the initial state s_0
2: s := s_0
3: paused := alreadyAcquired := insideAtomic := ∅
4: while Enabled(s) ≠ ∅ do
5:    t := a random transition in Enabled(s) such that
      ∀p, L.(p, L, t) ∉ paused
6:    if t = AtomicEnter (p) then
7:       s := Execute(s, t)
8:       add p to insideAtomic
9:    else if t =AtomicExit(p) then
10:      s := Execute(s, t)
11:      remove p from insideAtomic
12:      remove ∀L.(p, L) from alreadyAcquired
13:   else if t =ACQUIRE(p, L) then
14:      if p ∈insideAtomic then
15:         if (p, L) ∈ alreadyAcquired then
16:            add (p, L, t) in paused with probability q // q is 0.5
17:            print "WARNING: Atomicity violation possible"
18:         else
19:            add (p, L) to alreadyAcquired
20:            s := Execute(s, t)
21:         end if
22:      else
23:         s := Execute(s, t)
24:      end if
25:      if ∃p', t'.(p', L, t') ∈ paused and p ≠ p' then
26:         remove (p', L, t') from paused
27:         print "ERROR: Atomicity violation detected"
28:      end if
29:   end if
30:   if |paused| = |Enabled(s)| then
31:      remove a random element from paused
32:   end if
33: end while
34: if Alive(s) ≠ ∅ then
35:    print "ERROR: actual deadlock found"
36: end if
```

The ATOMFUZZER algorithm performs random execution as in the simple randomized algorithm in Algorithm 1. However, unlike the simple algorithm, ATOMFUZZER looks for the atomicity violation pattern described above. In particular, whenever ATOMFUZZER finds that a thread p is inside an atomic block and is about to acquire a lock L that has been previously acquired and released inside the same atomic block, ATOMFUZZER issues an atomicity violation warning. Such a warning would be given by any other static or dynamic atomicity checking tool. However, such a warning can be spurious unless one can show that some other thread can acquire and release the same lock L immediately before the thread p acquires the lock. ATOMFUZZER tries to create this scenario by changing the default scheduler behavior. Specifically, ATOMFUZZER *pauses* the execution of the thread p just before it acquires the lock L and allows the other threads to execute. If at any point in the execution, ATOMFUZZER discovers that some other thread has acquired the lock L, then ATOMFUZZER flags an atomicity violation error because it has created a scenario showing an atomicity violation.

The algorithm is formally described in Algorithm 2. The algorithm can produce three kinds of outputs:

1. **Warnings:** These are potential atomicity violations. Existing tools, such as Atomizer, already produce these warnings. We do not show these warnings to the user and we only use them for experimental evaluation.
2. **Errors:** These are real atomicity violations, i.e. ATOMFUZZER has actually created an execution showing the vi-

olations. Sometimes atomicity violations may not result in bugs because they are benign. Moreover, if we are using the heuristic, then an atomicity violation may not result in a bug because the heuristic may not match with the user intention.

3. **Bugs/Exceptions:** These are uncaught exceptions or program crashes that result due to real atomicity violations. If we are using the heuristic, then they indicate that the found atomicity violation is a bug.

In the algorithm we maintain three sets: *paused* to maintain information about the threads that we have paused in an effort to create an atomicity violation, *insideAtomic* to keep track of threads that are already inside an atomic block, and *alreadyAcquired* to keep track of locks that a thread has already acquired and released while inside its current atomic block. These sets are initially empty.

At every state ATOMFUZZER picks a random enabled transition t, such that the transition has not been paused for the purpose of creating an atomicity violation (see line 5). If a thread p executes `AtomicEnter`(p) (see lines 6–8), then we add p to the set *insideAtomic* to record the fact that the thread is now inside an atomic block. Similarly, if a thread p executes `AtomicExit`(p) (see lines 9–12), then we remove p from the set *insideAtomic* to indicate the fact that the thread is no longer inside an atomic block. We also clear all entries corresponding to the thread p in the set *alreadyAcquired*.

The key component of the ATOMFUZZER algorithm kicks in if the randomly picked transition is `ACQUIRE`(p, L). We can then have four cases.

- **Case 1:** (lines 14–17) If the thread p is already inside an atomic block and if (p, L) is in the set *alreadyAcquired*, then we know that thread p has previously acquired and released the lock L while inside the atomic block. Therefore, we raise an atomicity violation warning following Lipton's reduction algorithm [32]. ATOMFUZZER also puts the tuple (p, L, t) to the set *paused* with probability q to indicate that ATOMFUZZER wants to see if some thread can acquire and release L while p is paused and violate p's atomicity condition.

- **Case 2:** (lines 18–20) If the thread p is already inside an atomic block and if (p, L) is not in the set *alreadyAcquired*, then we know that thread p has not previously acquired and released the lock L while inside the atomic block. Therefore, there is no possibility of an atomicity violation at the current point. We add the pair (p, L) in the set *alreadyAcquired* to indicate that thread p should now look for the atomicity violation pattern over the lock L.

- **Case 3:** (lines 22–23) If thread p is not inside an atomic block, then simply execute t.

- **Case 4:** (lines 25–28) In all the above three cases, ATOMFUZZER checks if the current lock acquire can conflict with some other lock acquire that has been paused at line 16. ATOMFUZZER then flags an atomicity violation error.

Note that if ATOMFUZZER keeps on pausing threads without discovering any atomicity violation errors, then it may sometimes end up pausing all threads. If this ever happens, ATOMFUZZER breaks the deadlock by removing a random element from the set *paused* (see lines 30–32.)

```
1: class Account {
2:    int balance = 100;
3:    public synchronized getBalance() {
4:       return balance;
5:    }
6:    public synchronized withdraw(int amount) {
7:       balance = balance - amount;
8:       assert (balance >= 0);
9:    }
10:}

11:  Initially: Account acnt = new Account();

Thread T1:
12: atomic {
13:   if (acnt.getBalance()>=70)
14:   {
15:     acnt.withdraw(70);
16:   }
17: }

Thread T2:
18: atomic {
19:   if (acnt.getBalance()>=70)
20:   {
21:     acnt.withdraw(70);
22:   }
23: }
```

Figure 1: A program with an atomicity violation

3. ADVANTAGES OF ATOMFUZZER

3.1 Example 1 illustrating ATOMFUZZER

Consider the two-threaded program in Figure 1. The program defines a Java class `Account` that has two synchronized methods `getBalance` and `withdraw`. For simplicity of description, instead of defining two threads explicitly, we simply show the code that the two threads execute. A variable `acnt` is initialized with a new account. Then each thread withdraws an amount of 70 if the balance is greater than or equal to 70. There is no data race in the program because all accesses to the only shared field `balance` are protected by the same lock `acnt`. The code executed by each thread is required to be atomic. However, due to the lock of proper synchronization, the atomicity requirement is violated. Specifically, if we execute each thread serially, then the final balance becomes 30 and the assertion at line 8 is never violated. However, in the following multi-threaded execution

`T1:12, T1:13, T1:14, T2:18, T2:19, T2:20, T1:15, T1:16, T1:17, T2:21, T2:22, T2:23`

an atomicity violation takes place. This is because `T1` enters an atomic block at line 12; it then acquires and releases the lock `acnt` at line 13. Before `T1` acquires the same lock `acnt` at line 15, `T2` acquires and releases the lock `acnt` at line 19. This creates an atomicity violating locking pattern. Due to this atomicity violation, the invariant that "an amount can only be deducted if there is sufficient balance in the account" is violated and this results in the failure of the assertion at line 8 by `T2`.

We now illustrate the execution of ATOMFUZZER using the example in Figure 1. ATOMFUZZER starts executing the program by randomly picking one thread at every state and executing its next statement. After a few steps, either thread `T1` reaches the statement at line 15 with probability 0.5 or thread `T2` reaches the statement at line 21 with probability 0.5. There could be four cases depending on what `T1` and `T2` has executed so far. Consider the case in which

```
11:   Initially: Account acnt = new Account();

Thread T1:

12: atomic {
13:   if (acnt.getBalance()>=70)
14:   {
15:     acnt.withdraw(70);
16:   }
17: }

Thread T2:

18: f1();
19: f2();
20: f3();
21: f4();
22: f5();
23: f6();
24: atomic {
25:   if (acnt.getBalance()>=70)
26:   {
27:     acnt.withdraw(70);
28:   }
29: }
```

Figure 2: Another program with a rare atomicity violation

```
11:   Initially: Account acnt = new Account();
12:   Object lock = new Object();

Thread T1:

13: atomic {
14:  synchronized (lock) {
15:   if (acnt.getBalance()>=70)
16:   {
17:     acnt.withdraw(70);
18:   }
19:  }
20: }

Thread T2:

21: atomic {
22:  synchronized (lock) {
23:   if (acnt.getBalance()>=70)
24:   {
25:     acnt.withdraw(70);
26:   }
27:  }
28: }
```

Figure 3: Another program with no atomicity violation

`T1` reaches line 15 before `T2` reaches line 21 and `T2` is at line 18. At this state, ATOMFUZZER will pause the execution of `T1` because it has identified that there is a potential for atomicity violation. The execution of `T2` will continue as the other thread is paused. As soon as `T2` acquires the lock `acnt` at line 19, ATOMFUZZER will detect that a real atomicity violation has taken place and it will resume the execution of thread `T1`. The rest of the execution will violate the assertion at line 8.

The remaining cases are the following. (1) `T1` has reached line 15 and `T2` has reached line 20. (2) `T2` has reached line 21 and `T1` has reached line 12. (3) `T2` has reached line 21 and `T1` has reached line 14. In all the three cases ATOMFUZZER will create a real atomicity violation scenario. Therefore, in this example, ATOMFUZZER will create a real atomicity violation with probability 1. Note that if we perform pure random scheduling of the threads (see Algorithm 1), a real atomicity violation will be created with probability 0.5. This is because, in pure random scheduling, there is a 0.5 probability that each of the threads gets executed serially.

3.2 Example 2 illustrating that ATOMFUZZER can detect atomicity violations with high probability

The example in the previous section may not be convincing enough to establish the fact that ATOMFUZZER is better than the pure random scheduler to create atomicity violation scenarios with high probability. Therefore, to illustrate our claim that ATOMFUZZER can detect atomicity violations with high probability in many cases, we modify the example in Figure 1 slightly. The modified code is shown in Figure 2.

The modified example introduces a number of statements in `T2` to ensure that the atomic block in `T2` gets executed after the execution of a large number of statements and the atomic block in `T1` gets executed at the beginning. This snippet represents a pattern in real-world programs.

A pure random thread scheduler needs to generate the following executions

`T1:12, T1:13, T1:14, T2:18, T2:19, T2:20, T2:21, T2:22, T2:23, T2:24, T2:25, T2:26, T1:15, T1:16, T1:17, T2:27, T2:28, T2:29`

or

`T2:18, T2:19, T2:20, T2:21, T2:22, T2:23, T2:24, T2:25, T2:26, T1:12, T1:13, T1:14, T2:27, T2:28, T2:29,T1:15, T1:16, T1:17`

to create a real atomicity violation scenario. However, the probability of generating these executions is very low (i.e. less than 0.05).

However, it can be shown by similar reasoning as in the previous example that ATOMFUZZER will create a real atomicity violation scenario with probability 1 for this case.

3.3 Example 3 illustrating that ATOMFUZZER gives no false positives

In order to demonstrate that ATOMFUZZER gives no atomicity violation errors where an existing dynamic analysis tool [21] can give false warnings, we modify the example in Figure 1 again. The modified code is shown in Figure 3. In the modified code, we add an extra `lock` to protect the code of each thread. Due to the `lock`, each thread executes its code serially. Therefore, under no circumstances ATOMFUZZER would be able to create a real atomicity violation scenario. However, a tool like Atomizer will give an atomicity violation warning at line 17 and line 25. This is because Atomizer classifies any lock acquire as a right-mover (R) and any lock release as a left-mover (L) [32]. Therefore, from the execution of `T1`, Atomizer will infer the execution exhibits the following sequence $RRLRLL$, which violates atomicity according to Lipton's reduction algorithm [32].

3.4 Example 4 illustrating incompleteness of ATOMFUZZER

In order to demonstrate that ATOMFUZZER may not give an atomicity violation error when there is one, we modify the example in Figure 1 again. The modified code is shown in Figure 4. In the modified code, we introduce a user input `c`. If the value of input is `'y'`, the atomic block in `Thread T2` is executed. Therefore, if ATOMFUZZER is run with `c='n'`, then ATOMFUZZER will not

```
11:  Initially: Account acnt = new Account();

Thread T1:

12: atomic {
13:   if (acnt.getBalance()>=70)
14:   {
15:     acnt.withdraw(70);
16:   }
17: }

Thread T2:
18: char c = read_user_input();
19: if (c == 'y') {
20:   atomic {
21:     if (acnt.getBalance()>=70)
22:     {
23:       acnt.withdraw(70);
24:     }
25:   }
26: }
```

Figure 4: Another program with an atomicity violation. The program is tested on input `c='n'`

be able to create the atomicity violation error scenario. This shows that AtomFuzzer cannot catch all atomicity violations because the provided test suite may not be sufficient to create them.

4. IMPLEMENTATION

We have implemented the AtomFuzzer algorithm for Java in a prototype tool. The implementation is part of the CalFuzzer tool set [39, 40]. The implementation allows one to specify a code block as atomic.

AtomFuzzer instruments Java bytecode to observe relevant transitions and to control the thread scheduler. Bytecode instrumentation allows us to analyze any Java program for which the source code is not available. We use the SOOT compiler framework [45] to perform the instrumentation. The instrumentation inserts various methods provided by AtomFuzzer inside Java programs. These methods implement the AtomFuzzer algorithm. The instrumentor of AtomFuzzer modifies all bytecode associated with a Java program including the libraries it uses, except for the classes that are used to implement AtomFuzzer. This is because AtomFuzzer runs in the same memory space as the program under analysis. AtomFuzzer cannot track lock acquires and releases by native code or uninstrumented Java libraries. For such reasons, there is a possibility that AtomFuzzer can go into a deadlock if there are synchronization operations inside uninstrumented classes or native code. To avoid such scenarios, AtomFuzzer runs a monitor thread that periodically polls to check if there is any deadlock. If the monitor discovers a deadlock, it then removes one thread from the set *paused*.

AtomFuzzer can also escape from livelocks. Livelocks happen when all threads of the program end up in the *paused* set, except for one thread that does something in a loop without synchronizing with other threads. We observed such livelocks in a couple of our benchmarks. Even in the presence of livelocks, these benchmarks work correctly because the correctness of these benchmarks assumes that the underlying Java thread scheduler is fair. In order to avoid livelocks, AtomFuzzer creates a monitor thread that periodically removes those threads from the *paused* set that are waiting for a long time.

In order to control the default thread schedule, i.e. to pause and wake up a thread on demand, AtomFuzzer associates a semaphore with every thread. If a thread needs to pause, it tries to acquire the semaphore associated with it. A paused thread can be woken up by some other thread by releasing the semaphore associated with the thread. The use of semaphores helps AtomFuzzer to control the Java thread schedule without modifying the JVM scheduler.

In [33], it has been shown that it is sufficient to perform thread switches before synchronization operations, provided that the algorithm tracks all data races. AtomFuzzer, therefore, only performs random thread switches before synchronization operations. This particular restriction on thread switches keeps our implementation fast. Since AtomFuzzer only tracks synchronization operations, the runtime overhead of AtomFuzzer is significantly lower than that of other existing dynamic atomicity checking tools.

5. EMPIRICAL EVALUATION

5.1 Benchmark Programs

We have evaluated AtomFuzzer on a variety of Java multi-threaded programs. The benchmark includes both closed programs and open libraries that require test drivers to close them. We ran our experiments on a laptop with a 2.0 GHz Intel Core 2 Duo processor and 2GB RAM. We considered the following closed benchmark programs in our experiments: `cache4j`, a fast thread-safe implementation of a cache for Java objects, `sor`, a successive over-relaxation benchmark from ETH [47], `hedc`, a web-crawler application kernel also developed at ETH [47], `weblech`, a multi-threaded web site download and mirror tool, `jspider`, a highly configurable and customizable Web Spider engine, and `jigsaw 2.2.6`, W3C's leading-edge Web server platform. The total lines of code in these benchmark programs is approximately 600,000.

The open programs consist of several synchronized Collection classes provided with Sun's JDK, such as `Vector` in JDK 1.1, `ArrayList`, `LinkedList`, `HashSet`, and `TreeSet` in JDK 1.4.2. Most of these classes (except the `Vector` class) are not synchronized by default. The `java.util` package provides special functions `Collections.synchronizedList` and `Collections.synchronizedSet` to make the above classes synchronized. We considered two other widely known libraries: the Apache Commons-Collections and the Google Collections Library. In order to close the Collection classes, we wrote a multi-threaded test driver for each such class. A test driver starts by creating two empty objects of the class. The test driver also creates and starts a set of threads, where each thread executes different methods on either of the two objects concurrently. We created two objects because some of the methods, such as `containsAll`, take as an argument an object of the same type. For such methods, we call the method on one object and pass the other object as an argument.

Since we do not have atomicity annotations in our benchmark programs, we use the heuristic that any code block that is `synchronized` is atomic, in our experiments. The same assumption has been made in the Atomizer tool [21]. A rationale behind this assumption is that often programmers surround a code block with `synchronized` to achieve mutual exclusion, i.e. to ensure that the data inside the code block is accessed without interference from other threads. In other words, programmers often assume that a synchronized code block will behave atomically. Note that this assumption might give some false warnings if a synchronized block need not be atomic in a program. However, this does not affect our general claim that AtomFuzzer gives no false warnings—if the programmer properly annotates atomic blocks, then we get no false

Program name	Lines of code	Avg. Runtime in sec. Normal	AF	Slowdown	# Warnings	# Errors reported	# Bugs confirmed	Probability of detecting error	# Previously known
StringBuffer	1320	0.17	0.21	1.23	1	1	1	0.78	1
Vector	709	0.13	0.13	1.00	0	0	0	-	0
ArrayList	5,866	0.14	0.24	1.71	2	2	2	0.97	0
LinkedList	5,979	0.16	0.24	1.50	2	2	2	0.99	0
HashSet	7,086	0.14	0.24	1.71	2	2	2	0.98	0
TreeSet	7,532	0.15	0.24	1.60	2	2	2	0.99	0
LinkedHashSet	12,926	0.15	0.24	1.50	2	2	2	0.77	0
Apache BlockingBuffer	977	0.146	0.344	2.36	1	1	1	0.64	0
Apache BoundedFifoBuffer	1,437	0.140	0.296	2.11	1	1	1	0.73	0
Apache CircularFifoBuffer	3,370	0.139	0.309	2.22	1	1	1	0.70	0
Google ConcurrentMultiset	17,946	0.13	0.82	6.31	0	0	0	-	0
cache4j	3,897	3.3	35	10.60	1	1	0	1.00	0
sor	17,689	0.13	1.0	7.69	0	0	0	-	0
hedc	29,947	0.99	1.8	1.82	3	0	0	-	1
weblech	35,175	0.81	13.78	17.01	25	0	0	-	0
jspider	64,933	4.8	51	10.6	28	4	0	1.00	0
jigsaw	381,348	-[1]	-[1]	-	60	2	0	-	1

(1: running times for jigsaw are not reported due to the interactive nature of the webserver)

Table 1: Experimental results. AF stands for ATOMFUZZER.

warnings.

5.2 Results

Table 1 summarizes the results of our experiments. The open programs (libraries) are listed before the closed ones. Column 2 reports the total number of lines of code instrumented. Some Java language-level libraries are not instrumented, and therefore not included in the count. Columns 3 and 4 report the average runtime of each execution with and without ATOMFUZZER, respectively. We do not report the runtime on Jigsaw due to the interactive nature of the webserver. Column 5 reports the average slowdown due to ATOMFUZZER. We observed an average slowdown in the range of 1x-17x. This slowdown is significantly lower than the 20x-40x slowdown observed by Atomizer [21]. This is because we do not instrument the memory read and write operations. Since ATOMFUZZER is a tool for testing and debugging, we are not worried about the runtime as long as the runtime is a few seconds.

Columns 6 reports the number of warnings observed by ATOMFUZZER. Note that any such warning would also be reported by an existing atomicity checking tool such as Atomizer. Users of ATOMFUZZER should ignore these warnings; however, we recorded them to illustrate how well ATOMFUZZER performs in comparison with Atomizer. Column 7 reports the number of real atomicity violations detected by ATOMFUZZER. The total number of warnings found is 131. Out of them, ATOMFUZZER found 21 to be real atomicity violations. Therefore, ATOMFUZZER reduced the number of false warnings by a factor of 6x. Atomicity violations observed by ATOMFUZZER may not result in a real bug in a program. This is due to two reasons: 1) We use a heuristic to identify the atomic blocks and in some situations the heuristic identification of atomic blocks may not match the user intention. 2) An atomicity violation may be benign—it does not affect the correctness of the program. Therefore, in order to understand the effectiveness of the tool, we tried to identify if an atomicity violation error reported by ATOMFUZZER could lead to a bug. If in an execution, ATOMFUZZER observes an uncaught exception being thrown due to an atomicity violation, we conclude the atomicity violation error is a bug. Otherwise, we manually check the error and try to classify if the error is benign or it was reported due to a misclassification done by our heuristic.

Column 8 counts the number of bugs that resulted due to atomicity violations. ATOMFUZZER observed that 14 out of 21 atomicity violations could result in an uncaught exception. Note that existing atomicity checking tools cannot report the number of real atomicity violations or the number of exceptions that are thrown due to such atomicity violations. By reporting the real violations and exceptions and by making them reproducible, we significantly simplify the job of the programmer. Column 10 reports the number of previously known atomicity violations, i.e., the atomicity violations that were detected by Atomizer [21] and similar tools. We are able to discover a previously known atomicity violation in the `StringBuffer` class. However, we missed the previously known atomicity violations in `hedc` and `Jigsaw`. After a closer investigation, we found that these atomicity violations are due to race conditions and do not fall under the special pattern in which we are interested. Our previous dynamic race detection tool [40] reported them as races. Therefore, we believe that even if we miss some atomicity violations due to the consideration of a special kind of locking pattern, the missed ones can be easily detected by a race detection tool.

Column 9 shows that in most cases ATOMFUZZER can create a real atomicity violations in our benchmark programs with high probability. We ran ATOMFUZZER 10–1000 times for each benchmark and report for which fraction of the runs, at least one atomicity violation was detected. This roughly gives us the empirical probability of ATOMFUZZER creating an atomicity violation.

5.3 Bugs Found

ATOMFUZZER discovered several previously unknown atomicity violations in the JDK 1.4.2 synchronized classes `LinkedList`, `ArrayList`, `HashSet`, `TreeSet`, and `LinkedHashSet`. All these violations lead to uncaught exceptions; therefore, these violations indicate real bugs. Java provides wrappers to Collection classes to make them "thread-safe" in a concurrent program. For example, `java.util.Collections.synchronizedSet(Set s)` wraps a `java.util.Set` object, so that operations on a set are protected by a mutex. ATOMFUZZER discovered real atomicity violations in `containsAll` and `removeAll` methods. For example, if we call `l1.containsAll(l2)` then the following methods get called.

```
Collections.java:
  public boolean containsAll(Collection coll) {
    synchronized(mutex) {
      return c.containsAll(coll);
    }
```

```
  }


AbstractCollections.java:
    public boolean containsAll(Collection c) {
        Iterator e = c.iterator();
        while (e.hasNext()) {
           synchronized(c) {
             if(!contains(e.next()))
               return false;
           }
        }
        return true;
    }
```

The problem happens in `containsAll` of `AbstractCollections.java`. The while loop should be atomic. However, between two calls to `e.next()` another thread can change the collection `c` and this will result in a `ConcurrentModificationException`.

Similarly, the `removeAll` method has a real atomicity violation. The code is shown below.

```
Collections.java:
    public boolean removeAll(Collection coll) {
        synchronized(mutex) {
           return c.removeAll(coll);
        }
    }

AbstractCollections.java:
    public boolean removeAll(Collection c) {
       boolean modified = false;
       Iterator e = iterator();
       while (e.hasNext()) {
          if(c.contains(e.next())) {
             e.remove();
             modified = true;
          }
       }
       return modified;
   }

Collections.java:
   public boolean contains(Object o) {
     synchronized(mutex) {
       return c.contains(o);
     }
   }
```

The while loop inside `removeAll` should be atomic. However, `c.contains()` acquires and releases a lock in each iteration. This causes an atomicity violation in `removeAll`.

The errors discovered in the Apache Commons-Collections library are due to non-synchronized use of iterators. For example, the `BoundedFifoBuffer` is a wrapper class for an unbounded buffer, which in turn extends from JDK `AbstractCollection`. `BoundedFifoBuffer` is a synchronized Buffer, which locks the underlying collection before each operation. However, the implementation of the underlying collection (an `AbstractCollection` in this case) does not take synchronization into account, and uses the iterator in an unsafe manner. This causes an atomicity violation error which results in an exception.

5.4 Incompleteness Analysis

Since ATOMFUZZER is incomplete, we cannot definitely say that an atomicity violation warning is not an error if ATOMFUZZER has not classified the warning as an error. Similarly, we cannot definitely say that an atomicity violation error is not a bug if ATOMFUZZER has not classified the error as a bug. For example, in the case of `jspider` ATOMFUZZER reports 28 warnings, 4 errors, and 0 bugs. We cannot definitely say that the remaining 24 warnings are not errors and that the 4 errors are not bugs. In order to better understand the effectiveness of our technique, we manually analyzed the 28 warnings and 4 errors and found that all the remaining warnings (i.e. the warnings that were not classified as errors by ATOMFUZZER) are not errors and all the reported errors are not bugs. The results of our manual analysis show that ATOMFUZZER is relatively complete for `jspider`; however, they do not imply that ATOMFUZZER is complete for all concurrent programs. We next give a detailed description of the results of our manual analysis.

We found that there are two key reasons why some of the warnings are not errors. First, we found that many times the `main` thread releases and acquires a lock while inside an atomic block, but before creating any thread. Therefore, no other thread can interleave between the release and acquire of the lock. However, Lipton's reduction algorithm will give an atomicity violation warning. A simplified code snippet that gives such atomicity violation warning is shown below.

```
public static void initialize() {
  atomic {
     synchronized(L) {
        // do something
     }
     synchronized(L) {
        // do something
     }
  }
}

public static void main(String args[]) {
  initialize();
  (new SomeThread()).start();
}
```

Second, we found that in some situations all atomic blocks that access a particular lock (say L) are synchronized by a common lock (say L'.) In such situations, no other thread can acquire and release the lock L while a thread is in an atomic block and is accessing the same lock L. Therefore, a real atomicity violation cannot happen although Lipton's reduction algorithm will give an atomicity violation warning. A simplified code snippet that gives such atomicity violation warning is shown below.

```
public void foo()  {
  atomic {
    synchronized(L') {
       synchronized(L) {
          // do something
       }
       synchronized(L) {
          // do something
       }
    }
  }
}

Thread T1:

foo();

Thread T2:

foo();
```

We observed that there are two reasons for getting atomicity errors that are not bugs. First, we observed that our heuristic for identifying atomic blocks does not match the user intention in some situations. For example, in `jspider` some of the `run` methods that are entry method for threads are `synchronized`. Because of the heuristics we used, ATOMFUZZER treats them as atomic. However, it is unrealistic to assume that the entry method of a thread is atomic because such an assumption would make the entire thread atomic. In some other situations we found that a code block has been synchronized over a lock L because the thread calls L.`notify()` or L.`wait()` inside the block. In such scenarios the block should not treated as atomic because a call to l.`wait()` would naturally violate the atomicity assumption. However, since our heuristic treats any such block as atomic, we get a false atomicity error report. We can remove these false error reports by modifying our heuristic such that it does not treat such synchronized blocks as atomic.

ATOMFUZZER reports some atomicity errors which we have found to be benign. An example of such a benign error is shown below.

```
public static synchronized int getUniqueId()  {
  count++;
  return count;
}

Thread T1:

atomic {
  getUniqueId();
  ...
  getUniqueId();
}

Thread T2:

  getUniqueId();
  ...
```

Here any interleaving of calls to `getUniqueId()` is benign because the semantics of `getUniqueId()` allows such interleavings. We found such a benign atomicity violation error in `cache4j`.

6. OTHER RELATED WORK

Recently, a couple of random testing techniques [15, 44] for concurrent programs have been proposed. These techniques randomly seed a Java program under test with the `sleep()`, the `yield()`, and the `priority()` primitives at shared memory accesses and synchronization events. Although these techniques have successfully detected bugs in many programs, they have two limitations. First, these techniques are not systematic as the primitives `sleep()`, `yield()`, `priority()` can only advise the scheduler to make a thread switch, but cannot force a thread switch. Second, reproducibility cannot be guaranteed in such systems [44] unless there is built-in support for capture-and-replay [15]. ATOMFUZZER removes these limitations by explicitly controlling the scheduler. Moreover, ATOMFUZZER tries to bias the random thread scheduler so that real atomicity violations get created. Flanagan et al. [22] have independently proposed a sound and dynamic technique for catching atomicity violations in Java programs. The technique is based on happens-before analysis. However, their analysis does not focus on our proposed atomicity violation pattern.

Static verification [3, 13, 30, 37, 7] and model checking [14, 31, 26, 29, 46, 33] or path-sensitive search of the state space are alternative approaches to finding bugs in concurrent programs. Model checkers, being exhaustive in nature, can often find all concurrency related bugs in concurrent programs. Unfortunately, model checking does not scale with program size. Several other systematic and exhaustive techniques [6, 8, 42, 41] for testing concurrent and parallel programs have been developed recently. These techniques exhaustively explore all interleavings of a concurrent program by systematically switching threads at synchronization points.

Randomized algorithms for model checking have also been proposed. For example Monte Carlo Model Checking [27] uses a random walk on the state space to give a probabilistic guarantee of the validity of properties expressed in linear temporal logic. Randomized depth-first search [12] and its parallel extensions have been developed to dramatically improve the cost-effectiveness of state-space search techniques using parallelism. A randomized partial order sampling algorithm [39] helps to sample partial orders (i.e. non-equivalent executions) almost uniformly at random. Race directed random testing [40] uses an existing dynamic analysis tool to identify a set of pairs of statements that could potentially race in a multi-threaded execution. Each such pair is then used to bias a random scheduler so that the statements in the pair can be executed temporally next to each other. Note that race directed random testing needs an existing dynamic analysis tool to identify the pairs of program statements that could race. ATOMFUZZER does not use any existing analysis to identify the program points where atomicity could be violated. It identifies them at runtime.

7. CONCLUSION

We described ATOMFUZZER, a simple, but practical and effective technique to detect real atomicity problems in multi-threaded programs. An attractive feature of our technique is that it gives no false warnings if programmers appropriately annotate the atomic blocks. The tool also provides full support for replaying buggy executions; replay makes debugging simpler. Our tool has detected several previously unknown atomicity violations in mature Java code such as the JDK 1.4.2 synchronized collections and the Apache Commons framework.

Acknowledgments

We would like to thank Jacob Burnim and Leo Meyerovich for providing valuable comments on a draft of this paper. This work is supported in part by the NSF Grant CNS-0720906.

8. REFERENCES

[1] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting data races on weak memory systems. In *18th annual International Symposium on Computer architecture (ISCA)*, pages 234–243. ACM, 1991.

[2] R. Agarwal, A. Sasturkar, L. Wang, and S. D. Stoller. Optimized run-time race detection and atomicity checking using partial discovered types. In *20th IEEE/ACM International Conference on Automated software engineering (ASE)*, pages 233–242. ACM, 2005.

[3] A. Aiken and D. Gay. Barrier inference. In *25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 342–354. ACM, 1998.

[4] D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: a dialect of java without data races. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'00)*, pages 382–400, 2000.

[5] C. Boyapati and M. C. Rinard. A parameterized type system for race-free java programs. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'01)*, pages 56–69, 2001.

[6] D. Bruening. Systematic testing of multithreaded Java programs. Master's thesis, MIT, 1999.

[7] S. Burckhardt, R. Alur, and M. M. K. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *CM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI)*, pages 12–21, 2007.

[8] R. H. Carver and Y. Lei. A general model for reachability testing of concurrent programs. In *6th International Conference on Formal Engineering Methods (ICFEM'04)*, volume 3308 of *LNCS*, pages 76–98, 2004.

[9] J. D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proc. of the ACM SIGPLAN Conference on Programming language design and implementation*, pages 258–269, 2002.

[10] J.-D. Choi, B. P. Miller, and R. H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Trans. Program. Lang. Syst.*, 13(4):491–530, 1991.

[11] A. Dinning and E. Schonberg. Detecting access anomalies in programs with critical sections. In *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging*, 1991.

[12] M. B. Dwyer, S. Elbaum, S. Person, and R. Purandare. Parallel randomized state-space search. In *29th International Conference on Software Engineering (ICSE)*, pages 3–12. IEEE, 2007.

[13] M. B. Dwyer, J. Hatcliff, Robby, and V. P. Ranganath. Exploiting object escape and locking information in partial-order reductions for concurrent object-oriented programs. *Form. Methods Syst. Des.*, 25(2–3):199–240, 2004.

[14] J. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, 1999.

[15] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, , and S. Ur. Multithreaded Java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002.

[16] D. R. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 237–252, 2003.

[17] A. Farzan and P. Madhusudan. Causal atomicity. In *Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 315–328. Springer, 2006.

[18] C. Flanagan. Verifying commit-atomicity using model-checking. In *11th International SPIN Workshop*, pages 252–266, 2004.

[19] C. Flanagan and S. N. Freund. Type-based race detection for java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*, pages 219–232, 2000.

[20] C. Flanagan and S. N. Freund. Detecting race conditions in large programs. In *Proc. of the Program Analysis for Software Tools and Engineering Conference*, 2001.

[21] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–267, 2004.

[22] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 293–303. ACM, 2008.

[23] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proc. of the ACM SIGPLAN conference on Programming language design and implementation (PLDI'03)*, pages 338–349, 2003.

[24] S. N. Freund and S. Qadeer. Checking concise specifications for multithreaded software. *Journal of Object Technology*, 3(6):81–101, 2004.

[25] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *ACM SIGPLAN Conference on Programming language design and implementation*, pages 1–11, 2003.

[26] P. Godefroid. Model checking for programming languages using verisoft. In *24th Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[27] R. Grosu and S. A. Smolka. Monte carlo model checking. In *11th International Conference Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *LNCS*, pages 271–286, 2005.

[28] J. Hatcliff, Robby, and M. B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *Proc. of the International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, pages 175–190, 2004.

[29] K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. *Int. Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[30] T. A. Henzinger, R. Jhala, and R. Majumdar. Race checking by context inference. *SIGPLAN Not.*, 39(6):1–13, 2004.

[31] G. Holzmann. The Spin model checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

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

[33] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *ACM Symposium on Programming Language Design and Implementation (PLDI'07)*, 2007.

[34] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–319, 2006.

[35] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically classifying benign and harmful data racesallusing replay analysis. In *ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI)*, pages 22–31, 2007.

[36] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: context-sensitive correlation analysis for race detection. In *ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 320–331. ACM, 2006.

[37] S. Qadeer and D. Wu. Kiss: keep it simple and sequential. In *ACM SIGPLAN 2004 conference on Programming language design and implementation (PLDI)*, pages 14–24. ACM, 2004.

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

[39] K. Sen. Effective random testing of concurrent programs. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*, 2007.

[40] K. Sen. Race directed random testing of concurrent programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, 2008. To appear.

[41] K. Sen and G. Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *Haifa verification conference 2006 (HVC'06)*, Lecture Notes in Computer Science. Springer, 2006.

[42] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke. Using model checking with symbolic execution to verify parallel numerical programs. In *International symposium on Software testing and analysis (ISSTA)*, pages 157–168. ACM Press, 2006.

[43] N. Sterling. Warlock: A static data race analysis tool. In *USENIX Winter Technical Conference*, pages 97–106, 1993.

[44] S. D. Stoller. Testing concurrent Java programs using randomized scheduling. In *Workshop on Runtime Verification (RV'02)*, volume 70 of *ENTCS*, 2002.

[45] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *CASCON 1999*, pages 125–135, 1999.

[46] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *15th International Conference on Automated Software Engineering (ASE)*. IEEE, 2000.

[47] C. von Praun and T. R. Gross. Object race detection. In *16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA)*, pages 70–82. ACM, 2001.

[48] L. Wang and S. D. Stoller. Run-time analysis for atomicity. In *3rd Workshop on Run-time Verification (RV'03)*, volume 89 of *ENTCS*, 2003.

[49] L. Wang and S. D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *Proc. ACM SIGPLAN 2006 Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 137–146. ACM Press, Mar. 2006.