Fully Automatic and Precise Detection of Thread Safety Violations

Michael Pradel
Department of Computer Science
ETH Zurich

Thomas R. Gross
Department of Computer Science
ETH Zurich

Abstract

Concurrent, object-oriented programs often use thread-safe library classes. Existing techniques for testing a thread-safe class either rely on tests using the class, on formal specifications, or on both. Unfortunately, these techniques often are not fully automatic as they involve the user in analyzing the output. This paper presents an automatic testing technique that reveals concurrency bugs in supposedly thread-safe classes. The analysis requires as input only the class under test and reports only true positives. The key idea is to generate tests in which multiple threads call methods on a shared instance of the tested class. If a concurrent test exhibits an exception or a deadlock that cannot be triggered in any linearized execution of the test, the analysis reports a thread safety violation. The approach is easily applicable, because it is independent of hand-written tests and explicit specifications. The analysis finds 15 concurrency bugs in popular Java libraries, including two previously unknown bugs in the Java standard library.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.1.3 [*Programming Techniques*]: Object-oriented Programming; D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Languages, Reliability, Algorithms

Keywords Thread safety, Testing, Concurrent test generation

1. Introduction

Writing correct concurrent programs is hard for many reasons. Developers tend to think think sequentially, making concurrent programs hard to write and understand because of their non-deterministic and parallel nature. Furthermore, testing techniques for concurrent programs have not yet reached the sophistication of techniques for sequential programs.

In this paper, we address the problem of testing concurrent programs. Ideally, a testing technique requires a single input—a piece of software to analyze—and produces a single output—true positive reports of concurrency bugs in the tested software. Existing techniques do not meet this requirement, because they either require additional input, produce undesirable additional output, or both. As an additional input besides the software to analyze, dynamic approaches need tests that execute the software. Writing tests is often neglected due to time constraints. Furthermore, writing effective concurrent tests is difficult, because they should lead the program to sharing state between concurrently executing threads and should trigger many different execution paths. Another additional input that many static and dynamic analyses rely on are explicit, formal specifications, which must be developed in addition to the program itself. Unfortunately, few programs provide such specifications. As an additional output besides true positive bug reports, many existing approaches produce false positives. They reduce the usability of a bug finding technique and can even make it unusable in practice [3].

This paper presents an automatic testing technique to detect concurrency bugs in thread-safe classes. We consider a class to be *thread-safe* if it "behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or coordination on the part of the calling code" [20]. We say that a class is *thread-unsafe* otherwise. Our approach requires a single input, a program or library containing the class under test (CUT), possibly accompanied by third-party libraries that the program depends on, and produces a single output, true positive reports about concurrency bugs in the CUT. The analysis reports problems that result in an exception or a deadlock. As both are obvious signs of misbehavior, the analysis reports only true positives. Each bug report contains an executable, concurrent test that exposes the problem.

As a motivating example, consider a previously unknown bug that our approach detected in the `StringBuffer` class of Sun's Java standard library in versions 1.6 and 1.7.[1] `StringBuffer` is documented as thread-safe and tries to synchronize all accesses to its internal data by locking on the `StringBuffer` instance (Figure 1a). Our analysis generates tests that use `StringBuffer`s concurrently. For example, the test in Figure 1b creates a `StringBuffer` and uses it in two concurrent threads. Executing the test leads to an exception because `insert()` retrieves the length of `s`, a parameter of type `CharSequence`, before acquiring the lock. The documentation states: "This class synchronizes only on the string buffer performing the operation, not on the source." This behavior is fatal if the passed source is the `StringBuffer` itself, a case the developers of the class apparently forgot to consider. In this case, retrieving the size of the `StringBuffer` is non-atomic with the consecutive update, leading to a potential boundary error caused by an interleaved update. Our analysis detects this bug with no input but the `StringBuffer` class and producing no output but this bug report.

PLDI'12, June 11–16, 2012, Beijing, China.

[1] We reported the problem and the developers acknowledged it as a bug. See entry 7100996 in Sun's bug database.

```
class StringBuffer {
  StringBuffer(String s) {
    // initialize with the given String
  }
  synchronized void deleteCharAt(int index) {
    // modify while holding the lock
  }
  void insert(int dstOffset, CharSequence s) {
    int l = s.length();
    // BUG: l may change
    this.insert(dstOffset, s, 0, l);
  }
  synchronized void insert(int dstOffset,
      CharSequence s, int start, int end) {
    // modify while holding the lock
  }
}
```

(a) Supposedly thread-safe class.

```
StringBuffer sb = new StringBuffer("abc");
```

Thread 1	Thread 2
`sb.insert(1, sb);`	`sb.deleteCharAt(0);`

Result: IndexOutOfBoundsException in Thread 1

(b) Execution of a generated concurrent test exposing a thread-safety bug.

Figure 1: `StringBuffer` and a test execution exposing a thread-safety bug.

Our approach is enabled by the combination of two contributions. The first contribution is a test generation technique that creates input to exercise the methods of a CUT from multiple threads. Each generated test consists of a sequential part that instantiates the CUT and a concurrent part in which multiple threads call supposedly thread-safe methods on the CUT instance. The technique enhances existing techniques to generate sequential tests [11, 37] by adapting them to a concurrent setting. Simply running generated sequential tests in parallel is very unlikely to result in shared state and to expose concurrency bugs. Instead, our technique generates tests that share a single CUT instance between multiple threads and that expose concurrency bugs by using the instance concurrently.

Some existing bug finding techniques use a manually written test harness that, given a set of calls with concrete parameters, randomly selects and executes calls [5, 21]. Our test generation technique advances upon this approach in two ways. First, it relieves developers from providing calls and from finding appropriate parameters for these calls. The test generator creates parameters by instantiating other classes and by calling methods of other classes. Second, and even more important, an automated approach produces more diverse tests because it tries to combine many different methods, parameters, and receiver objects. As a result, generated tests include usage scenarios that a human may not come up with. For example, the test in Figure 1b triggers a bug by passing a `StringBuffer` to itself, a situation that apparently remained untested for several years.

The second contribution is a test oracle, called the *thread safety oracle*, that determines whether a concurrent test execution exposes a thread safety problem. The oracle classifies a concurrent execution as erroneous if the execution leads to an exception or to a deadlock *and* if this exception or deadlock cannot be triggered by any linearization of the calls in the concurrent execution. A linearization maps all calls of a concurrent test into a single thread while preserving the order of calls made by individual threads. The oracle assumes that sequential executions are deterministic. The thread safety oracle is generic, precise, and practical. It is generic, because it can detect different kinds of concurrency bugs, including data races, atomicity violations, and deadlocks, given that the bug eventually causes an exception or deadlock. A study of 105 real-world concurrency bugs found that 62% of all bugs lead to a crash or a deadlock [30], suggesting that the thread safety oracle addresses a significant problem. The oracle is precise, because it guarantees that each reported problem is a real bug, since exceptions and deadlocks are certainly undesired behavior. Finally, the oracle is practical, because it does not require any explicit specification to come with the CUT and instead leverages generic and implicit indicators of incorrectness.

The thread safety oracle relates to seminal work on linearizability as correctness criterion [22] and its recent adaption to object-oriented programs in Line-Up [5]. In contrast to Line-Up, our oracle is more effective and more efficient. It is more effective, because each test execution that fails according to the oracle is a true positive. In contrast, the Line-Up oracle classifies 426 of 1,800 test executions as failing, which—after manual inspection—contain seven bugs [5]. This high violations-to-bugs ratio is due to benign linearizability violations and to multiple violations caused by the same bug. Our oracle is more efficient, because it only runs linearizations of a concurrent test if the concurrent test leads to an exception or a deadlock. Instead, Line-Up explores all linearizations before running a concurrent test.[2]

We implement our approach as a prototype tool for Java and apply it to six popular code bases, including the Java standard library and Apache Commons Database Connection Pools (DBCP). The analysis detects six previously unknown bugs, for example, two problems in supposedly thread-safe classes, `ConcurrentHashMap` and `StringBuffer`, of the most recent version of the Java standard library. In total, our implementation successfully reveals 15 bugs in the analyzed code bases. Twelve of them are exposed through a deadlock or an exception thrown by the virtual machine or the Java standard library, that is, not relying on the use of exceptions in the CUT itself.

In summary, this paper makes the following contributions:

- A technique to generate concurrent tests that exercise a CUT from multiple threads.
- A generic test oracle that finds concurrent executions that expose thread safety bugs in the CUT.
- Empirical evidence that combining concurrent test generation with the thread safety oracle is effective in automatically finding concurrency bugs.

2. Overview

We presents a dynamic analysis to detect concurrency bugs in thread-safe classes. This section explains the key ideas of the analysis; Sections 3 and 4 fill in the details.

There are three requirements for detecting a thread safety bug with a dynamic analysis. First, a test that drives the CUT to a state exposing the bug. Second, an execution environment that exposes the bug when executing the test. Third, an oracle that recognizes the execution as erroneous. Various approaches addressing the second requirement have been proposed [6, 10, 12, 33, 43]. This work focuses on the first and the third requirement. We address the first requirement by generating tests that drive a CUT (Section 3). Using automatically generated tests instead of relying on existing

[2] A direct comparison of our results with Line-Up is not possible as the details about the classes under test [5], as well as the specific bugs found, are no longer available [4].

tests has two benefits: (1) it makes our approach easy to use and (2) it provides a large range of different inputs that can expose bugs not triggered by manually written tests. We address the third requirement with an oracle that determines whether a concurrent execution exposes a thread safety problem (Section 4).

Our analysis consists of three iteratively performed steps, which correspond to the three requirements. At first, a generator of concurrent tests creates a test that sets up an instance of the CUT and that calls methods on this instance from multiple threads. Next, we execute the generated test. Since the class is supposed to be thread-safe, it should synchronize concurrent calls on the shared instance as needed. As a result, the methods should "behave as if they occur in some serial order that is consistent with the order of the method calls made by each of the individual threads involved" [1]. The third step of our analysis checks whether the CUT behaves as expected. If the analysis finds that the concurrent execution shows behavior not possible in any linearization of calls, it reports a bug and terminates. Otherwise, the analysis goes back to the first step and continues until a stopping criterion, such as a timeout or a specified maximum number of generated tests, is reached.

3. Generating Concurrent Tests

This section presents a technique to generate concurrent tests that exercise a CUT from multiple threads. The input to the test generator is a class C and, optionally, a set of auxiliary classes $\mathcal{A}$ that C depends on. The generator creates *call sequences*. Each call c_i in a call sequence $(c_1, \ldots, c_n)$ consists of a method signature, a possibly empty list of input variables, and an optional output variable. The input variables of a call represent parameters passed to the method. For an instance call, the first input variable is the receiver of the call. The output variable of a call represents its return value. We model constructor invocations as calls where the output variable represents the new object. Similarly, we model field accesses as calls where the underlying object is the only input variable and where the output variable represents the field value. We say that a call sequence is *well-defined* if each input variable of a call c_j is the output variable of a call c_i with $i < j$, that is, if each call uses only variables defined on prior calls.

A *test* consists of a prefix and a set of suffixes. The *prefix* is a call sequence supposed to be executed in a single thread. The prefix instantiates the CUT and calls methods on the CUT instance to "grow" the object, that is, to bring it into a state that may allow the suffixes to trigger a bug. A *suffix* is a call sequence supposed to be executed concurrently with other suffixes after executing the prefix. All suffixes share the output variables of the prefix and can use them as input variables for suffix calls. In particular, all suffixes share the CUT instance created in the prefix. While our general approach is independent of the number of suffixes, we focus on two suffixes per test, so that a test is a triple (p, s_1, s_2) where p is the prefix and s_1, s_2 are suffixes. The rationale for this choice is that most real-world concurrency bugs involve no more than two threads [30].

Figure 1b is a very simple example for a test. The prefix is a single call to the constructor of `StringBuffer`, which returns the CUT instance `sb`. Each of the two suffixes calls a single method using the shared CUT instance `sb` as the receiver. In practice, effective concurrent tests are not always that simple. Calling a method often requires providing parameters of a particular type, which in turn may require calling other methods. Furthermore, triggering a bug often requires to bring the CUT instance in a particular state, for example, by calling setter methods.

3.1 Tasks

To automatically generate concurrent tests, we divide the generation of a test into simpler tasks, which build and extend call sequences. Each task takes a call sequence $s_{in} = (c_1, \ldots, c_i)$ and returns a new sequence $s_{out} = (c_1, \ldots, c_i, c_j, \ldots, c_n)$ that appends $n - j + 1$ additional calls to s_{in}. The additional calls can use output variables of previous calls as their input variables. We use three kinds of tasks:

- $instantiateCUTTask$, which appends a call to create a new instance of the CUT.
- $callCUTTask$, which appends a call to the CUT instance.
- $parameterTask$, which makes a parameter of a particular type available by choosing an output variable of a previous call or by appending a call that returns an object of the required type.

A task succeeds if it extends s_{in} with additional calls in such a way that s_{out} is well-defined and that s_{out} executes in a single thread without throwing an uncaught exception. The latter requirement adapts an idea from sequential test generation [37], namely to use the result of executing call sequences for selecting which sequences to extend further. Generated call sequences that result in an exception typically contain an illegal call, for example, a call that violates the precondition for calling the method. Although focusing on non-exceptional call sequences does not guarantee that each call is legal, it ensures that extending a sequence will eventually lead to executing more calls without reaching an obviously illegal state.

To successfully accomplish a task, the test generator extends s_{in} into *candidates* for s_{out} until a candidate is found that is well-defined and that executes without an exception. If after creating a specified number of candidates no candidate has fulfilled these conditions, the test generator gives up on this task and the task fails. For example, the $callCUTTask$ may fail if the CUT instance is in a state that does not allow calling any of the CUT methods.

3.2 Test Generation Algorithm

Algorithm 1 describes how our analysis generates a concurrent test. There are three global variables, which maintain their values over multiple invocations of the algorithm: the set $\mathcal{P}$ of prefixes, the map $\mathcal{M}$ assigning a prefix to its set of suffixes, and the set $\mathcal{T}$ of already generated but not yet returned tests.

The algorithm has three main steps. At first, it creates a new prefix or chooses a previously created prefix (lines 6 to 20). Then, it creates a new suffix for the prefix (lines 21 to 26). Finally, the algorithm creates tests by combining the new suffix with each existing suffix (lines 27 to 29). To create prefixes and suffixes, the algorithm invokes tasks. The functions $randTake$ and $randRemove$ randomly select an element of a set with a uniform probability distribution, and the $randRemove$ function also removes the selected element from the set.

Creating Prefixes During the first step (lines 6 to 20), the algorithm creates a new prefix unless $maxPrefixes$ (discussed in Section 5) have already been created. The reason for limiting the number of prefixes is that we require multiple suffixes for each prefix to test different concurrent usages of a CUT instance. To create a new prefix, the algorithm invokes the $instantiateCUTTask$. This task randomly chooses a method m from all methods in C and $\mathcal{A}$ that have C or a subtype of C as return type. If calling m requires parameters, the task invokes $parameterTask$s to make these parameters available. The $instantiateCUTTask$ returns a call sequence that creates all required parameters, stores them into output variables, and calls m. If the test generator cannot instantiate the CUT, for example, because there is no public constructor, the algorithm fails. For Figure 1b, the $instantiateCUTTask$ chose the `StringBuffer` constructor as m and obtains a string literal as parameter from $parameterTask$ (literals are passed without storing them into a variable).

Algorithm 1 Returns a concurrent test (p, s_1, s_2)

```
1: P: set of prefixes                                   ▷ global variables
2: M: maps a prefix to suffixes
3: T: set of ready-to-use tests
4: if |T| > 0 then
5:     return randRemove(T)
6: if |P| < maxPrefixes then                            ▷ create a new prefix
7:     p ← instantiateCUTTask(empty call sequence)
8:     if p = failed then
9:         if P = ∅ then
10:            fail("cannot instantiate CUT")
11:        else
12:            p ← randTake(P)
13:    else
14:        for i ← 1, maxStateChangerTries do
15:            p_ext ← callCUTTask(p)
16:            if p_ext ≠ failed then
17:                p ← p_ext
18:    P ← P ∪ {p}
19: else
20:    p ← randTake(P)
21: s_1 ← empty call sequence                           ▷ create a new suffix
22: for i ← 1, maxCUTCallTries do
23:    s_{1,ext} ← callCUTTask(s_1, p)
24:    if s_{1,ext} ≠ failed then
25:        s_1 ← s_{1,ext}
26: M(p) ← M(p) ∪ {s_1}
27: for all s_2 ∈ M(p) do                               ▷ one test for each pair of suffixes
28:    T ← T ∪ {(p, s_1, s_2)}
29: return randRemove(T)
```

After instantiating the CUT, the algorithm tries to invoke methods on the CUT instance to change the state of the CUT instance. The $callCUTTask$ randomly chooses a method among all methods of C and invokes the $parameterTask$ for each required parameter. The prefix in Figure 1b contains no state changing call. Alternatively to the shown prefix, the test generator could have created the call sequence

```
StringBuffer sb = new StringBuffer();
sb.append("abc");
```

where the second call is a state changing call.

Creating Suffixes During the second step (lines 21 to 26), the algorithm creates a new suffix for the prefix p. Therefore, it repeatedly invokes the $callCUTTask$. In addition to the suffix, the call in line 23 passes the prefix to the task to allow for using output variables from the prefix as input variables in the suffix. After appending calls to the CUT instance, the new suffix is added to the set of suffixes for the prefix p.

All tasks that append method calls depend on the $parameterTask$. This task uses three strategies to make a parameter of a particular type t available. If the given call sequence already has one or more output variables of type t or a subtype of t, the task randomly chooses between reusing a variable and creating a fresh variable. The rationale for creating fresh variables is to diversify the generated tests instead of passing the same parameters again and again. To reuse a variable, the task randomly chooses from all available variables with matching type. To create a fresh variable, the behavior depends on the type t. If t is a primitive type or `String`, the task returns a randomly created literal. Otherwise, the tasks randomly chooses a method from all methods in C and $\mathcal{A}$ returning t or a subtype of t. If calling this method requires parameters, the task recursively invokes itself. We limit the number of recursions to avoid infinite loops and fail the task if the limit is reached. If there is no method providing t, the $parameterTask$ returns `null`. Using `null` as a parameter may be illegal. In this case, executing the sequence can result in an exception, for example, a `NullPointerException`, so that the candidate is rejected.

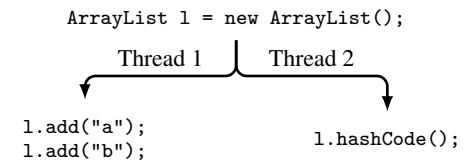


Figure 2: Test execution using a thread-unsafe class.

Creating Tests The third step of the algorithm (lines 27 to 29) combines the new suffix with each existing suffix for the prefix into a test. The algorithm stores the created tests in $\mathcal{T}$ and on further invocations returns a randomly selected test from $\mathcal{T}$ until $\mathcal{T}$ becomes empty.

3.3 Beyond This Work

The tests generated by Algorithm 1 provide input to exercise a class without any human effort, and hence, increase the usefulness of existing dynamic analyses. For example, executions of generated tests can be analyzed by existing detectors of data races [15, 32, 42, 44] or atomicity violations [16, 21, 28, 38, 45]. Without generated tests, these analyses are limited to manually created tests and the bugs exposed by these tests.

4. Thread Safety Oracle

This section presents an automatic technique to determine whether the execution of a concurrent test exposes a thread safety bug.

4.1 Thread Safety

A class is said to be thread-safe if multiple threads can use it without synchronization *and* if the behavior observed by each thread is equivalent to a linearization of all calls that maintains the order of calls in each thread [1, 20]. Saying that a class is thread-safe means that all its methods are thread-safe. Our approach can also deal with classes that guarantee thread safety for a subset of all methods by excluding the unsafe methods when generating suffixes for a test.

Figure 2 is an example for a thread-unsafe class and a test execution that exposes this property. The concurrent use of `ArrayList` from `java.util` results in an exception that does not occur in any of the three possible linearizations of the calls:

`add` → `add` → `hashCode`
`add` → `hashCode` → `add`
`hashCode` → `add` → `add`

Therefore, the execution in Figure 2 shows that `ArrayList` is thread-unsafe, as expected.

A property related to thread safety is atomicity, that is, the guarantee that a sequence of operations performed by a thread appears to execute without interleaved operations by other threads. One way to make a class thread-safe is to guarantee that each call to one of its methods appears to be atomic for the calling thread. However, a thread-safe class does *not* guarantee that multiple calls to a shared instance of the class are executed atomically [45]. For example, consider the use of the thread-safe class `java.util.concurrent.CopyOnWriteArrayList` in Figure 3. Executing the concurrent test has three possible outputs. For all three,

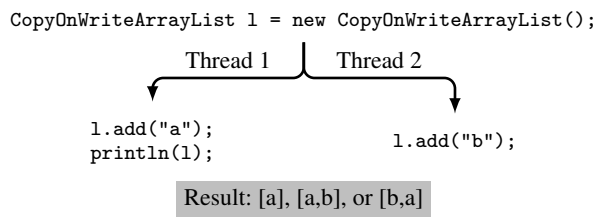


Figure 3: Test execution using a thread-safe class.

there is a linearization of calls that provides the same result, so the test execution does not expose a thread safety problem:

`add("a")` → `println()` → `add("b")` gives [a]
`add("a")` → `add("b")` → `println()` gives [a,b]
`add("b")` → `add("a")` → `println()` gives [b,a]

4.2 Definitions

The thread safety oracle answers the question whether a concurrent test execution exposes a thread safety problem by comparing the concurrent execution to executions of linearizations of the test. We use $\oplus$ to concatenate sequences, and $c \rightarrow_s c'$ indicates that call c comes before call c' in a call sequence s.

Definition 1 (Linearization)**.** *For a test (p, s_1, s_2), let $\mathcal{P}_{12}$ be the set of all permutations of the call sequence $s_1 \oplus s_2$. The set of linearizations of the test is:*

$$\mathcal{L}_{(p,s_1,s_2)} = \{p \oplus s_{12} \mid s_{12} \in \mathcal{P}_{12} \wedge (\forall c, c' \ (c \rightarrow_{s_1} c' \Rightarrow c \rightarrow_{s_{12}} c') \wedge (c \rightarrow_{s_2} c' \Rightarrow c \rightarrow_{s_{12}} c'))\}$$

That is, a linearization of a test (p, s_1, s_2) appends to p all calls from s_1 and from s_2 in a way that preserves the order of calls in s_1 and s_2.

Definition 2 (Execution)**.** *For a test (p, s_1, s_2), we denote the set of all distinguishable executions of this test as $\mathcal{E}_{(p,s_1,s_2)}$. Each $e_{(p,s_1,s_2)} \in \mathcal{E}_{(p,s_1,s_2)}$ represents the sequential execution of p followed by a concurrent execution of s_1 and s_2. Likewise, we denote the sequential execution of a call sequence s as e_s.*

A single test can have multiple distinguishable executions because of the non-determinism of concurrent executions.

The following definition of thread safety refers to the equivalence $e_1 \cong e_2$ of two executions e_1 and e_2. We discuss in Section 4.3 how our oracle decides whether two executions are equivalent.

Definition 3 (Thread safety)**.** *Let $\mathcal{T}_C$ be the set of all possible tests for a class C. C is thread-safe if and only if:*

$$\forall (p, s_1, s_2) \in \mathcal{T}_C \ \forall e_{(p,s_1,s_2)} \in \mathcal{E}_{(p,s_1,s_2)} \ \exists l \in \mathcal{L}_{(p,s_1,s_2)} \text{ so that } e_{(p,s_1,s_2)} \cong e_l$$

That is, a class is thread-safe if each concurrent test execution has an equivalent linearization.

4.3 The Test Oracle

Showing that a class is thread-safe according to Definition 3 is difficult in practice, because all possible tests and all possible executions of these tests would have to be considered. However, the thread safety oracle can show that a class C is thread-unsafe by showing:

$$\exists (p, s_1, s_2) \in \mathcal{T}_C \ \exists e_{(p,s_1,s_2)} \in \mathcal{E}_{(p,s_1,s_2)} \text{ so that } \forall l \in \mathcal{L}_{(p,s_1,s_2)} \ e_{(p,s_1,s_2)} \ncong e_l$$

Algorithm 2 Checks whether a test (p, s_1, s_2) exposes a thread safety bug

```
1:  repeat
2:    e_(p,s1,s2) ← execute(p, s1, s2)
3:    if failed(e_(p,s1,s2)) then
4:      seqFailed ← false
5:      for all l ∈ L(p, s1, s2) do
6:        if seqFailed = false then
7:          e_l ← execute(l)
8:          if failed(e_l) ∧ sameFailure(e_(p,s1,s2), e_l) then
9:            seqFailed ← true
10:     if seqFailed = false then
11:       report bug e_(p,s1,s2) and exit
12: until maxConcExecs reached
```

That is, the thread safety oracle tries to find a test that exposes behavior not possible with any linearization of the test. To decide whether two executions are equivalent, the oracle compares the exceptions and deadlocks caused by the executions.

Definition 4 (Equivalence of executions)**.** *Two executions e_1 and e_2 are equivalent if*

- *neither e_1 nor e_2 results in an exception or a deadlock, or*
- *both e_1 and e_2 fail for the same reason (that is, the same type of exception is thrown or both executions end with a deadlock).*

Although this abstraction ignores many potential differences between executions, such as different return values of method calls, it is crucial to ensure that the analysis only reports a class as thread-unsafe if this is definitely true.

Algorithm 2 shows how the analysis checks whether a test (p, s_1, s_2) exposes a thread safety problem. The algorithm repeatedly executes the test until a maximum number of concurrent executions is reached, or until a bug is found. If the test fails, that is, it throws an exception or results in a deadlock, the algorithm executes all linearizations of the test to check whether the same failure occurs during a sequential execution of the same calls. If no linearization exposes the same failure, the algorithm reports a bug because the CUT is thread-unsafe.

The thread safety oracle is sound but incomplete.[3] Every execution for which the oracle reports a bug is guaranteed to expose a real thread safety problem according to Definition 3, but the oracle may classify an execution as correct even though it exposes a thread safety problem. The soundness of the oracle ensures that our approach detects concurrency bugs without reporting false positives.

We build upon two assumptions, which we find true for most real-world classes during our evaluation. First, we assume that uncaught exceptions and deadlocks that occur in a concurrent usage of a class but not in a sequential usage are considered a problem. This assumption is in line with the commonly accepted definition of thread safety [1, 20]. Second, we assume that sequential executions of a call sequence behave deterministically. Sequentially non-deterministic methods, for example, methods that depend on the current system time, should be excluded from our analysis. Alternatively, our analysis can be combined with a runtime environment that ensures deterministic sequential execution [40].

[3] We mean soundness and completeness regarding incorrectness [47]. In other communities, such as type systems, the terms are typically used with respect to correctness, that is, inverse to the usage here.

5. Implementation

We implement the test generator and the thread safety oracle into an automatic bug detection tool for thread-safe Java classes. This section presents several challenges faced by the implementation and how we address them.

The test generator executes many call sequences and must do so both efficiently and without interference between different executions. To execute sequences efficiently we take a reflection-based approach similar to the sequential test generator Randoop [37]. A problem not addressed by Randoop is that different call sequences may interfere because of static state. For example, a call sequence s_1 may assign a value to a static field and a call sequence s_2 may read the static field. As a result, the outcome of executing s_2 may vary depending on whether s_1 is executed before s_2. This problem is independent of concurrency. We address the problem by resetting all static state to the state of the freshly loaded classes before each execution of a call sequence. For this purpose, our implementation instruments classes so that all modifications of static state are recorded and can be reset. Csallner and Smaragdakis describe a similar approach for a sequential test generator [11].

The test generator takes a random seed as an input, which allows for precise replay of the test generation process, as long as the tested classes behave deterministically in sequential executions. Experience with sequential, random test generation shows that short runs with many different seeds trigger bugs faster than few runs with a small number of seeds [9]. Our initial experiments confirmed this observation, so we run the analysis in multiple rounds that each use a different random seed. The first rounds stop after trying a small number of suffixes (ten) for a single prefix. Later rounds gradually raise the maximum number of generated prefixes ($maxPrefixes$) to 25 and the maximum number of generated suffixes to 500. The values of other parameters used in Algorithm 1 are $maxStateChangerTries = 5$, $maxConcExecs = 100$, and $maxCUTCallTries = 2$.

To detect deadlocks, we use the management interface for the thread system of the Java virtual machine. A daemon thread periodically queries this interface and notifies the thread safety oracle in case of a deadlock.

Although being a prototype, the performance of our tool is acceptable for a testing tool (details in Section 6). The by far most important bottleneck of our implementation is the repeated concurrent execution of tests. For example, for the CUT taking the longest to analyze, 99.5% of the time is spent with concurrent executions. We see two ways to address this issue. First, the analysis can exploit multiple cores by exploring different concurrent executions of the same test in parallel. Second, our analysis can be easily combined with existing techniques to increase the probability of hitting a bug by controlling or perturbing the scheduler [6, 10, 12, 33, 43]. Our current implementation executes tests with the standard Java thread scheduler. To plug a more sophisticated scheduling technique into our approach, one can redefine the $execute$ function of Algorithm 2.

6. Evaluation

We evaluate our approach by applying the prototype implementation to Java classes from six popular code bases: the Java standard library shipped with Sun's JDK, the database connection pool library Apache Commons DBCP, the serialization library XStream, the text processing toolkit LingPipe, the chart library JFreeChart, and Joda-Time, a library to handle data and time.

6.1 Experimental Setup

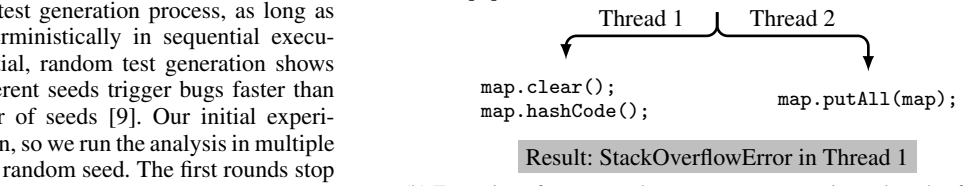
All experiments are done on an eight-core machine with 3GHz Intel Xeon processors and 8GB memory running 32-bit Ubuntu Linux and the Java Hotspot VM version 1.6.0_27, giving 2GB memory to the VM. We run experiments with different CUTs in parallel but run at most four tests at a time to reserve a core for each concurrent thread exercising the CUT. We repeat each experiment ten times with different random seeds [37].

To analyze a CUT, the test generator uses all other public classes from the code base and common classes from the Java standard library as auxiliary classes.

```
class ConcurrentHashMap {
  void clear() {
    final Segment[] segments = this.segments;
    for (int j = 0; j < segments.length; ++j) {
      Segment s = segmentAt(segments, j);
      if (s != null)
        s.clear(); // locks the segment
    }
  }
  void putAll(Map m) {
    for (Map.Entry e : m.entrySet())
      // BUG: m's entries may change
      put(e.getKey(), e.getValue());
  }
  Object put(Object key, Object value) {
    Segment s = /* get segment in a thread-safe way */;
    return s.put(key, value); // locks the segment
  }
}
```

(a) Supposedly thread-safe class.

```
ConcurrentHashMap map = new ConcurrentHashMap();
map.put("a", "b");
```



Result: StackOverflowError in Thread 1

(b) Execution of a generated concurrent test exposing a thread-safety bug.

Figure 4: Concurrency bug in `ConcurrentHashMap`.

6.2 Bugs Found

The analysis found 15 bugs in supposedly thread-safe classes, six of them previously unknown. Each bug can cause concurrency bugs in clients relying on thread-safe classes. Table 1 lists all bugs along with the reason for failing. The last column indicates whether the reason is a deadlock or an exception thrown implicitly by the virtual machine or by the Java standard library, or if triggering the bug requires an explicitly thrown exception in the analyzed code base. For twelve of 15 bugs, an implicit exception is sufficient to reveal the bug. That is, our approach reveals most bugs without any requirement on the analyzed classes, such as throwing an exception if an unsafe state is reached.

The analysis reveals two previously unknown bugs in the Java standard library, one of them shown as the motivating example in the introduction (Figure 1). The other new bug in the Java standard library is illustrated in Figure 4. The class `ConcurrentHashMap` is part of the `java.util.concurrent` package, which provides thread-safe collection classes. For better scalability, the class divides the map into segments that are locked independently of each other, instead of relying on a single exclusion lock. Unfortunately, `putAll()` does not consider the case where the passed map is the same as the receiver object of the call. The method retrieves the entries of the passed map without any synchronization and then passes each el-

ID	Code base	Class	Declared thread-safe	Found unsafe	Reason for failing	Implicit
Previously unknown bugs:						
(1)	JDK 1.6.0_27 and 1.7.0	StringBuffer	yes	yes	IndexOutOfBoundsException	yes
(2)	JDK 1.6.0_27 and 1.7.0	ConcurrentHashMap	yes	yes	StackOverflowError	yes
(3)	Commons DBCP 1.4	SharedPoolDataSource	yes	yes	ConcurrentModificationException	yes
(4)	Commons DBCP 1.4	PerUserPoolDataSource	yes	yes	ConcurrentModificationException	yes
(5)	XStream 1.4.1	XStream	yes	yes	NullPointerException	yes
(6)	LingPipe 4.1.0	MedlineSentenceModel	yes	yes	IllegalStateException	no
Known bugs:						
(7)	JDK 1.1	BufferedInputStream	yes	yes	NullPointerException	yes
(8)	JDK 1.4.1	Logger	yes	yes	NullPointerException	yes
(9)	JDK 1.4.2	SynchronizedMap	yes	yes	Deadlock	yes
(10)	JFreeChart 0.9.8	TimeSeries	yes	yes	NullPointerException	yes
(11)	JFreeChart 0.9.8	XYSeries	yes	yes	ConcurrentModificationException	yes
(12)	JFreeChart 0.9.12	NumberAxis	yes	yes	IllegalArgumentException	no
(13)	JFreeChart 1.0.1	PeriodAxis	yes	yes	IllegalArgumentException	no
(14)	JFreeChart 1.0.9	XYPlot	yes	yes	ConcurrentModificationException	yes
(15)	JFreeChart 1.0.13	Day	yes	yes	NumberFormatException	yes
Automatic classification of classes as thread-unsafe:						
(16)	Joda-Time 2.0	DateTimeFormatterBuilder	no	yes	IndexOutOfBoundsException (10x)	yes
(17)	Joda-Time 2.0	DateTimeParserBucket	no	yes	IllegalArgumentException (9x)	no
					NullPointerException (1x)	yes
(18)	Joda-Time 2.0	DateTimeZoneBuilder	no	yes	NullPointerException (6x)	yes
					ArrayIndexOutOfBoundsException (2x)	yes
					IllegalFieldValueException (2x)	yes
(19)	Joda-Time 2.0	MutableDateTime	no	yes	IllegalFieldValueException (9x)	no
					ArithmeticException (1x)	yes
(20)	Joda-Time 2.0	MutableInterval	no	yes	IllegalArgumentException (10x)	no
(21)	Joda-Time 2.0	MutablePeriod	no	yes	ArithmeticException (10x)	yes
(22)	Joda-Time 2.0	PeriodFormatterBuilder	no	yes	ConcurrentModificationException (5x)	yes
					IndexOutOfBoundsException (4x)	yes
					IllegalStateException (1x)	no
	Joda-Time 2.0	ZoneInfoCompiler	no	no	(stopped after 24h)	–

Table 1: Summary of results. The last column indicates whether the reason for failing is implicit in the Java runtime environment or explicitly specified in the code base under test.

ement to the correctly synchronized `put()`. As a result, a call to `map.putAll(map)` can undo changes done by a concurrently executing thread that also modifies the map—a clear thread safety bug. Our analysis generates the test in Figure 4b, which exposes the problem. Calling `hashCode()` results in a stack overflow for self-referential collections, because the method recursively calls itself (this is documented behavior). If `ConcurrentHashMap` were thread-safe, this behavior should not be triggered by the test, because all three possible linearizations of the calls in Figure 4b call `clear()` before calling `hashCode()`. However, the test fails with a `StackOverflowError`, because `putAll()` undoes the effects of the concurrently executed `clear()`.

Figure 5 is a previously unknown bug that our analysis detects in Apache Commons DBCP. The supposedly thread-safe class `SharedPoolDataSource` provides two methods `setDataSourceName()` and `close()` that register and unregister the data source via static methods of `InstanceKeyObjectFactory`, respectively. The factory class maintains a thread-unsafe `HashMap` assigning names to data sources. Although registering new instances is synchronized to avoid concurrent accesses to the `HashMap`, unregistering instances is not synchronized. The generated test in Figure 5b shows that this lack of synchronization leads to an exception when calling `setDataSourceName()` and `close()` concurrently.

To allow for reproducing our results, all analyzed classes, descriptions of the bugs, and a generated test to trigger each bug are available at `http://mp.binaervarianz.de/pldi2012`.

6.3 Annotating Classes as Thread-unsafe

Beyond finding bugs, our analysis can be used to analyze classes having no documentation on their thread safety and to automatically annotate these classes as thread-unsafe where appropriate. In a preliminary study for this work, we found that one of the most common concurrency-related questions of Java developers is whether a particular library class is thread-safe. Few libraries come with precise documentation to answer this question. To address this lack of documentation, our analysis can automatically annotate classes as thread-unsafe. Since the oracle is sound, these annotations are guaranteed to be correct.

To evaluate this usage scenario, we run our analysis on a library that specifies for each class whether it is thread-safe or not. The library (Joda-Time) contains 41 classes, of which 33 are documented as thread-safe and eight are documented as thread-unsafe. The lower part of Table 1 summarizes the results. For seven of the eight thread-unsafe classes, our analysis detects a thread safety problem. The missing class, `ZoneInfoCompiler`, reads files from the file system, transforms them, and writes other files as output.

```
class SharedPoolDataSource {
  void setDataSourceName(String v) {
    key = InstanceKeyObjectFactory
                    .registerNewInstance(this);
  }
  void close() {
    InstanceKeyObjectFactory.removeInstance(key);
  }
}

class InstanceKeyObjectFactory {
  static final Map instanceMap = new HashMap();
  synchronized static String
      registerNewInstance(SharedPoolDataSource ds) {
    // iterate over instanceMap
  }
  static void removeInstance(String key) {
    // BUG: unsynchronized access to instanceMap
    instanceMap.remove(key);
  }
}
```

(a) Supposedly thread-safe class.

```
SharedPoolDataSource ds1 = new SharedPoolDataSource();
ds1.setConnectionPoolDataSource(null);
```

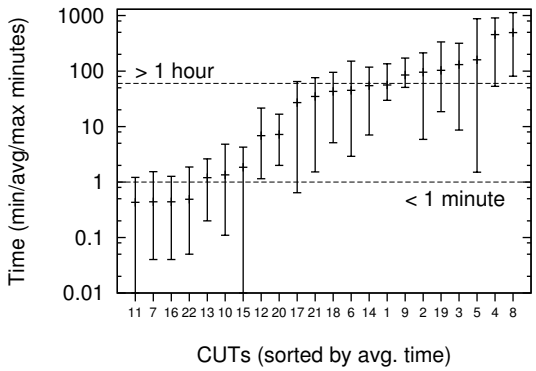
Thread 1 Thread 2

```
dataSource.setDataSourceName("a");   dataSource.close();
```

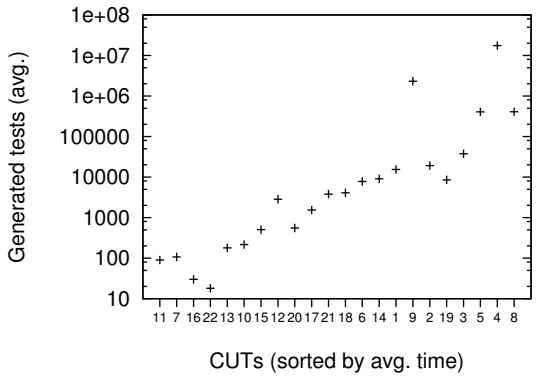
Result: ConcurrentModificationException in Thread 1

(b) Execution of a generated concurrent test exposing a thread-safety bug.

Figure 5: Concurrency bug in Apache Commons DBCP.



(a) Time to trigger a thread-safety problem.



(b) Tests generated before triggering a thread-safety problem.

Figure 6: Effort required to trigger a thread-safety problem.

Since the thread-safety oracle does not check the integrity of such files, it cannot detect problems caused by concurrent usages of the class. For the 33 classes that are documented as thread-safe, no problems are found after running the analysis for 24 hours.

6.4 Effort of Using the Analysis

Using our approach involves minimal human effort, because the analysis requires the source code or byte code of the classes under test as only input and produces true positives as only output. Experience from applying automated bug finding techniques in industry shows that both properties are important [3].

The computational effort of our implementation is acceptable for an automatic testing tool. Figure 6a shows how long the analysis takes to trigger the problems from Table 1. The horizontal axis shows the IDs from Table 1, sorting the classes by the average time required to find the problem. The vertical axis gives the minimum, average, and maximum time taken to find the problem over ten runs. Most of the problems are found within one hour. For several problems, the analysis takes only a few seconds. Other classes require several hours of computation time, with up to 8.2 hours for bug 8 (JDK's `Logger`). Given that the bug remained unnoticed for several years in one of the most popular Java libraries, we consider this time to be still acceptable. Section 5 outlines ways to reduce the running time of our implementation with additional engineering effort. The approach has very moderate memory requirements. The test generator selects methods randomly and therefore does not maintain significant amounts of state. If executing a generated call sequence exceeds the available memory, an exception is thrown and the sequence is not extended.

A question related to running time is how many tests the analysis generates and executes before hitting a bug. Figure 6b shows the average number of generated tests for each bug, listing the bugs in the same order as in Figure 6a. For some bugs, a small number of tests (several hundreds or even less than hundred) suffices to expose the problem. Other bugs require more tests, up to 17 million for bug 4. A manual inspection of bugs requiring many tests suggests that the task of executing a bug-exposing test with the "right" thread interleaving dominates over the task of generating a bug-exposing test. Combining our work with techniques to control scheduling may reduce the number of tests for hitting a bug.

7. Limitations

First, the approach assumes that exceptions and deadlocks are undesirable. We found this "implicit specification" to be true in practice, but in principle a class could throw exceptions as part of its legal, concurrent behavior. Second, the approach has limited support for multi-object bugs. Although the generated tests combine multiple classes and objects, they only reveal bugs triggered by concurrent calls to the same object. Third, the approach is limited to bugs that manifest through an exception or deadlock, and therefore it may miss more subtle problems leading to incorrect but not obviously wrong results.

8. Related Work

8.1 Finding Concurrency Bugs

Table 2 compares this work with other bug finding techniques based on four criteria: whether static or dynamic analysis is used, the

Approach		Input			Output		Correctness criterion				
		P	PT	PTS	B	BF	DR	Atom	DL	Crash	Other
[15, 32, 36, 42, 49]	D	○	●	○	○	●	●	○	○	○	○
[8]	SD	○	●	○	○	●	●	○	○	○	○
[44]	D	○	●	○	●	○	●	○	○	●	○
[2, 14, 21, 29, 39, 48]	D	○	●	○	○	●	○	●	○	○	○
[16, 45]	D	○	○	●	●	○	○	●	○	○	○
[28, 38]	D	○	○	●	●	○	○	●	○	●	○
[34]	S	○	●	○	○	●	○	○	●	○	○
[25]	D	○	●	○	●	○	○	○	●	○	○
[26]	D	○	○	●	○	●	○	○	●	○	○
[51]	D	○	●	○	●	○	○	○	○	●	○
[13, 18]	D	○	○	●	●	○	○	○	○	○	●
[5, 24, 31]	D	○	●	○	○	●	○	○	○	○	●
[7, 17]	D	○	○	●	○	●	○	○	○	○	●
[52]	SD	○	●	○	○	●	○	○	○	○	●
This work	D	●	○	○	●	○	○	○	●	●	○

Table 2: Comparison with existing static (S) and dynamic (D) approaches. Input: program (P), program and tests (PT), or program, tests, and specifications (PTS). Output: bugs (B) or bugs and false positives (BF). Correctness criterion: data race (DR), atomicity violations (Atom), deadlock (DL), crash, or other.

input and output of the analysis, and the correctness criterion. The unique feature of our work is to require only a program as input.

Data Races Dynamic data race detectors search for unsynchronized, conflicting accesses to shared data by analyzing happens-before relations [15, 32], by checking whether the program follows a locking discipline [42, 49], or by a combination of both techniques [36]. Our approach detects data races, if they manifest through an exception or a deadlock.

Atomicity Violations Analyses for finding atomicity violations rely on specifications of atomic blocks or of sets of atomic variable accesses, provided manually [16, 38, 45] or inferred heuristically [2, 14, 21]. Inference causes false positives when violating source code is not supposed to be atomic. Our analysis detects atomicity violations, if they lead to an exception or a deadlock.

Deadlocks Naik et al. [34] search deadlocks statically but require tests. Joshi et al. [26] model check a reduced program and rely on annotations of condition variables. Both analyses report false positives. Our analysis finds deadlocks triggered by generated tests.

Active Testing To avoid false positives, active testing validates potential bugs by controlling the scheduler to provoke an exception or a deadlock. The approach has been applied to data races [44], atomicity violations [28, 38], deadlocks [25], and memory-related concurrency bugs [51]. Our approach shares the idea of reporting problems only if a certainly undesired situation occurs but does not rely on manually written tests.

Linearizability Herlihy and Wing introduce linearizability as a correctness criterion for concurrent objects [22]. Line-Up [5] checks the linearizability of calls but requires manually specified method parameters. As shown in Figure 1, unusual parameters that a human may miss can trigger long-standing bugs. Line-Up executes all linearizations before running a concurrent test, whereas our oracle analyzes linearizations only if the test fails. Elmas et al. [13] and Fonseca et al. [17] propose linearizability-based analyses that require specifications to abstract the state of a component.

Other Correctness Criteria Gao et al. [18] search for typestate errors in multi-threaded programs and rely on typestate specifications. Joshi et al. [27] filter false warnings from verifying concurrent programs by using a sequential version of a program as an oracle for the concurrent program. The approach relies on formal specifications and tests. Other approaches check for violations of inferred invariants [24, 31, 46] or unusual orderings [52]. The price paid for not relying on explicit specifications are false positives.

8.2 Support for Finding Concurrency Bugs

Several techniques control the thread scheduling when running a concurrent program repeatedly, for example, based on model checking [10, 33], random scheduling [6, 43], or artificial delays [12]. These techniques reduce the time to trigger a bug and could enhance the performance of our analysis. Pugh and Ayewah [41] and Jagannath et al. [23] address the problem of manually writing concurrent unit tests. Our work is orthogonal to theirs, because we generate tests automatically.

8.3 Test Generation

This work is inspired by techniques to generate sequential tests, such as [11, 19, 37]. In contrast to them, our test generator creates concurrent tests. Integrating more elaborate test generation techniques, such as learning from observed call sequences [50], into our approach could help to detect complex bugs faster.

Ballerina [35] generates efficient multi-threaded tests, showing that two threads, each with a single call, can trigger many concurrency bugs. The test generator described here is directed towards generating objects of required types and aims to generate (prefix, suffix, suffix) triples to expose concurrency problems. In addition, Ballerina checks test executions for linearizability (similar to [5]) and therefore produces false positives.

9. Conclusions

We present an automatic testing technique to reveal severe bugs in thread-safe classes. Using our approach involves very little human effort because it requires neither tests nor explicit specifications for the classes under test, and because it produces only true positive bug reports. The analysis is enabled by two contributions: (1) a technique for generating tests that exercise a thread-safe class from multiple threads, and (2) an oracle that reports a bug when a concurrent test execution results in an exception or a deadlock that cannot occur in any linearized execution of the test. We validate our claims by applying the analysis to six popular Java libraries. The analysis reveals 15 bugs in supposedly thread-safe classes, including two previously unknown bugs in the Java standard library.

Our technique for generating concurrent tests provides input that can drive other dynamic analyses, which traditionally rely on manually written tests. Generated tests not only add diversity to otherwise available tests but also allow for dynamically analyzing classes that have no tests at all.

Acknowledgments

Thanks to Zoltán Majó, Albert Noll, Faheem Ullah, and the anonymous reviewers for their valuable comments. The work presented in this paper was partially supported by the Swiss National Science Foundation under grant number 200021-134453.

References

[1] API documentation of *java.lang.StringBuffer* (Java platform standard edition 6), 2011.

[2] C. Artho, K. Havelund, and A. Biere. High-level data races. *Softw Test Verif Rel*, 13(4):207–227, 2003.

[3] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. R. Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun ACM*, 53(2):66–75, 2010.

[4] S. Burckhardt and R. Tan. Private communication, August 2011.

[5] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-Up: a complete and automatic linearizability checker. In *PLDI*, pages 330–340, 2010.

[6] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS*, pages 167–178, 2010.

[7] J. Burnim, T. Elmas, G. C. Necula, and K. Sen. NDSeq: runtime checking for nondeterministic sequential specifications of parallel correctness. In *PLDI*, pages 401–414, 2011.

[8] 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 *PLDI*, pages 258–269, 2002.

[9] I. Ciupa, A. Pretschner, A. Leitner, M. Oriol, and B. Meyer. On the predictability of random tests for object-oriented software. In *ICST*, pages 72–81, 2008.

[10] K. E. Coons, S. Burckhardt, and M. Musuvathi. GAMBIT: effective unit testing for concurrency libraries. In *PPOPP*, pages 15–24, 2010.

[11] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software Pract Exper*, 34(11):1025–1050, Sept. 2004.

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

[13] T. Elmas, S. Tasiran, and S. Qadeer. VYRD: verifying concurrent programs by runtime refinement-violation detection. In *PLDI*, pages 27–37, 2005.

[14] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL*, pages 256–267, 2004.

[15] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *PLDI*, pages 121–133, 2009.

[16] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *PLDI*, pages 293–303, 2008.

[17] P. Fonseca, C. Li, and R. Rodrigues. Finding complex concurrency bugs in large multi-threaded applications. In *EuroSys*, pages 215–228, 2011.

[18] Q. Gao, W. Zhang, Z. Chen, M. Zheng, and F. Qin. 2ndstrike: toward manifesting hidden concurrency typestate bugs. In *ASPLOS*, pages 239–250, 2011.

[19] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, pages 213–223, 2005.

[20] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. 2006.

[21] C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic detection of atomic-set-serializability violations. In *ICSE*, pages 231–240, 2008.

[22] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM T Progr Lang Sys*, 12(3):463–492, 1990.

[23] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, and D. Marinov. Improved multithreaded unit testing. In *ESEC/FSE*, pages 223–233, 2011.

[24] G. Jin, A. V. Thakur, B. Liblit, and S. Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *OOPSLA*, pages 241–255, 2010.

[25] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI*, pages 110–120, 2009.

[26] P. Joshi, M. Naik, K. Sen, and D. Gay. An effective dynamic analysis for detecting generalized deadlocks. In *FSE*, pages 327–336, 2010.

[27] S. Joshi, S. K. Lahiri, and A. Lal. Underspecified harnesses and interleaved bugs. In *POPL*, pages 19–30, 2012.

[28] Z. Lai, S.-C. Cheung, and W. K. Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *ICSE*, pages 235–244, 2010.

[29] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *ASPLOS*, pages 37–48, 2006.

[30] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, pages 329–339, 2008.

[31] B. Lucia and L. Ceze. Finding concurrency bugs with context-aware communication graphs. In *MICRO*, pages 553–563, 2009.

[32] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: effective sampling for lightweight data-race detection. In *PLDI*, pages 134–143, 2009.

[33] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *OSDI*, pages 267–280, 2008.

[34] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *ICSE*, pages 386–396, 2009.

[35] A. Nistor, Q. Luo, M. Pradel, T. R. Gross, and D. Marinov. Ballerina: Automatic generation and clustering of efficient random unit tests for multithreaded code. In *ICSE*, 2012.

[36] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *PPOPP*, pages 167–178, 2003.

[37] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE*, pages 75–84, 2007.

[38] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *FSE*, pages 135–145, 2008.

[39] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *ASPLOS*, pages 25–36, 2009.

[40] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs. In *CGO*, pages 2–11, 2010.

[41] W. Pugh and N. Ayewah. Unit testing concurrent software. In *ASE*, pages 513–516, 2007.

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

[43] K. Sen. Effective random testing of concurrent programs. In *ASE*, pages 323–332, 2007.

[44] K. Sen. Race directed random testing of concurrent programs. In *PLDI*, pages 11–21, 2008.

[45] O. Shacham, N. Bronson, A. Aiken, M. Sagiv, M. Vechev, and E. Yahav. Testing atomicity of composed concurrent operations. In *OOPSLA*, pages 51–64, 2011.

[46] Y. Shi, S. Park, Z. Yin, S. Lu, Y. Zhou, W. Chen, and W. Zheng. Do I use the wrong definition?: Defuse: definition-use invariants for detecting concurrency and sequential bugs. In *OOPSLA*, pages 160–174, 2010.

[47] Y. Smaragdakis and C. Csallner. Combining static and dynamic reasoning for bug detection. In *TAP*, pages 1–16, 2007.

[48] F. Sorrentino, A. Farzan, and P. Madhusudan. PENELOPE: weaving threads to expose atomicity violations. In *FSE*, pages 37–46, 2010.

[49] C. von Praun and T. R. Gross. Object race detection. In *OOPSLA*, pages 70–82, 2001.

[50] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst. Combined static and dynamic automated test generation. In *ISSTA*, pages 353–363, 2011.

[51] W. Zhang, C. Sun, and S. Lu. ConMem: detecting severe concurrency bugs through an effect-oriented approach. In *ASPLOS*, pages 179–192, 2010.

[52] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. W. Reps. ConSeq: detecting concurrency bugs through sequential errors. In *ASPLOS*, pages 251–264, 2011.