2017 IEEE/ACM 39th International Conference on Software Engineering

Efficient Detection of Thread Safety Violations via Coverage-Guided Generation of Concurrent Tests

Ankit Choudhary
Department of Computer Science
TU Darmstadt, Germany

Shan Lu
Department of Computer Science
University of Chicago, USA

Michael Pradel
Department of Computer Science
TU Darmstadt, Germany

***Abstract*—As writing concurrent programs is challenging, developers often rely on thread-safe classes, which encapsulate most synchronization issues. Testing such classes is crucial to ensure the correctness of concurrent programs. An effective approach to uncover otherwise missed concurrency bugs is to automatically generate concurrent tests. Existing approaches either create tests randomly, which is inefficient, build on a computationally expensive analysis of potential concurrency bugs exposed by sequential tests, or focus on exposing a particular kind of concurrency bugs, such as atomicity violations. This paper presents CovCon, a coverage-guided approach to generate concurrent tests. The key idea is to measure how often pairs of methods have already been executed concurrently and to focus the test generation on infrequently or not at all covered pairs of methods. The approach is independent of any particular bug pattern, allowing it to find arbitrary concurrency bugs, and is computationally inexpensive, allowing it to generate many tests in short time. We apply CovCon to 18 thread-safe Java classes, and it detects concurrency bugs in 17 of them. Compared to five state of the art approaches, CovCon detects more bugs than any other approach while requiring less time. Specifically, our approach finds bugs faster in 38 of 47 cases, with speedups of at least 4x for 22 of 47 cases.**

***Keywords*-test generation; coverage; concurrency**

I. Introduction

Writing correct and efficient concurrent software is difficult. To alleviate the pain of developing concurrent software in object-oriented, shared-memory languages, such as Java or C++, developers often use *thread-safe classes*. These classes encapsulate concurrency-related challenges, such as how to synchronize concurrent memory accesses in an efficient and deadlock-free way, and provide an easy-to-use interface to developers. In essence, each thread can use an instance of a thread-safe class as if no other thread was using the instance concurrently, without any synchronization between the threads that access the instance. The thread-safe class guarantees that the behavior of the execution is equivalent to a method-level linearization of the concurrent calls to the shared instance.

Because thread-safe classes are widely used, ensuring their correctness is crucial for ensuring the overall correctness of concurrent programs. An effective approach for validating the correctness of a thread-safe class is automated test generation. The basic idea is that a test generator creates concurrent unit tests, i.e., tests where multiple concurrently executing threads exercise a shared instance of the class under test. To determine whether a generated test exposes a bug, the concurrent execution is then compared against linearizations of the test [5], [39] or analyzed by existing detectors of data races [16], [34], [45], [47], atomicity violations [1], [15], [38], or deadlocks [26].

Existing generators of concurrent tests fall into three categories. *Random-based* approaches select at random which methods to call in the concurrently executed threads and randomly combine threads with each other [33], [39]. These approaches have minimal requirements but are often inefficient, because they tend to repeatedly test already exercised behavior, yet miss behavior that can expose bugs. *Sequential test-based* approaches execute existing sequential unit tests of the thread-safe class, identify concurrency bugs that may occur when combining multiple sequential tests into concurrent tests, and then synthesize such tests [42]–[44]. These approaches rely on appropriate sequential tests and impose a relatively high computational cost. Finally, *coverage-based* approaches compute a set of concurrency-related coverage requirements and generate tests that cover as many of these requirements as possible [51], [55]. This direction is promising. However, existing approaches use coverage criteria that demand expensive coverage measurement and test generation. Furthermore, a common limitation of existing non-random approaches is to consider only a particular kind of concurrency bugs, e.g., data races [44], atomicity violations [43], [55], or deadlocks [42].

This paper presents a novel coverage-guided approach for generating concurrent tests that can detect arbitrary kinds of concurrency bugs. The approach is a simple yet effective alternative to the existing approaches. We follow the philosophy of recent coverage-based approaches [51], [55] to guide random-based test generation toward not yet covered interleavings, making it less random and more likely to trigger not yet exercised (mis)behavior. *Unlike* existing approaches, we drastically simplify the analysis and computation associated with interleaving coverage. We face two major challenges in achieving this goal. First, we require a lightweight technique to assess which interleavings are not yet covered by existing test cases. Naively enumerating all possible interleavings (i.e., total orders of memory accesses) is practically infeasible. Existing coverage-based approaches focus on partial orders of memory accesses, of which there are finitely many. However, they still require an expensive coverage measurement that analyzes every access to global or heap objects. Second, we require a lightweight technique for steering automatic test generation

1558-1225/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSE.2017.32

toward tests likely to cover not-yet-covered interleavings. This is non-trivial for existing approaches, as they need to figure out which call sequences may cover specific memory accesses that are part of a targeted interleaving. Furthermore, the obvious solution, prioritizing interleavings that have been covered the least or not at all, fails for realistic classes under test because some interleavings may be infeasible.

To address these two challenges, we use a coverage metric that abstracts the set of interleavings into *concurrent method pairs*, i.e., the set of pairs of methods that execute concurrently. An important insight that enables the idea to scale to realistic classes is that concurrent method pairs yield a finite coverage domain that is computationally inexpensive to measure, yet an effective approximation of possible and covered interleavings. The approach instruments the class under test to measure which pairs of methods execute concurrently when executing a generated concurrent test. The approach then uses the coverage information to easily construct tests aimed at covering infrequently covered pairs of methods, while minimizing the priority of method pairs that simply cannot execute concurrently, e.g., because both methods are protected by the same lock. We call the presented approach CovCon, standing for Coverage-guided generation of Concurrent tests. The main benefit of CovCon is to be both conceptually and computationally simple, yet effective and efficient in triggering concurrency bugs.

Note that our work differs from recent work on generating schedules based on concurrent coverage metrics [21], [60], [65] because CovCon generates test cases instead of schedules. The problem of generating test cases is orthogonal to the problem of generating schedules for executing these test cases. Once test cases are available, e.g., generated by CovCon, they can be explored with any approach for generating schedules, including coverage-based approaches [21], [60], [65], active testing [25], bounded exploration [31], and exhaustive exploration [28], [57]. In contrast, without bug-triggering test cases, even the best schedule exploration technique cannot find any bugs.

Steering the testing process toward higher coverage is inspired by a long tradition of using coverage to direct the testing of sequential programs. Traditional sequential coverage metrics, such as statement coverage, branch coverage, or definition-use coverage, are widely used to measure testing sufficiency. They indicate which parts of the code still lack testing, and guide developers or automated test generation tools [2], [24], [36], [58] toward those parts. Coverage-directed testing generation is a natural step for concurrency testing, given the success of coverage-directed testing in sequential software. When taking this step, finding a simple yet effective coverage measurement and test generation scheme is crucial to the success of this promising direction.

We evaluate the CovCon approach with 18 thread-safe classes from popular Java projects and compare our coverage-guided approach with five existing approaches. Our approach detects concurrency bugs in 17 of the 18 classes, which is more than any other approach. At the same time, CovCon significantly reduces the time required to find bugs: For 38 of 47 cases, our approach is faster than the state of art, with speedups of at least 4x for 22 of the 47 cases.

In summary, this paper contributes the following:

- The first approach to use a finite, simple, and generic metric of interleaving coverage as a driver for generating concurrent tests.
- The insight that concurrent method pairs provide an inexpensive metric of interleaving coverage that effectively steers test generation. This insight yields a conceptually simple yet highly effective test generation approach to automatically test thread-safe classes.
- A comprehensive experimental comparison of generators of concurrent tests that exceeds previous experiments both in the number of classes and in the number of approaches. The results show that CovCon outperforms all state of the art approaches, both in terms of bug finding capabilities and efficiency.

II. Overview

This section outlines the main ideas of our coverage-guided approach for generating concurrent tests and illustrates them with an example. Figure 1a shows a supposedly thread-safe class under test. Methods $m2$ and $m3$ both use a field `li`, which $m2$ may set to `null` and which $m3$ may dereference. Unfortunately, $m3$ checks whether `li` is non-`null` before dereferencing it without enforcing that the check and the dereference operation are executed atomically. As a result, a client of the class that calls $m2$ and $m3$ concurrently may trigger a `NullPointerException`, violating the thread safety property that the class intends to guarantee.

To test the class, CovCon generates multi-threaded unit tests (Section IV-D), such as the test shown in Figure 1b. The test instantiates the class, may call some of its methods (not shown in the example), and then spawns two threads. The concurrent threads both use the shared instance `c` of the class under test and call some of its methods. Each generated test focuses on a particular pair of methods, $(m1, m3)$ for the example, and tries to execute these methods concurrently.

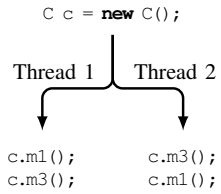
Next, CovCon executes the test and gathers an execution trace that records method entry and exit events of the concurrently executing threads (Section IV-B). The trace in Figure 1c shows the calls done by thread 1 and 2 on the left and right, respectively. Note that the trace contains events of both the direct calls, such as the calls to $m1$ and $m3$, and the indirect calls, such as the calls to $m4$ through $m3$. During the execution in Figure 1c, several pairs of methods execute concurrently: $(m3, m3)$, $(m3, m4)$, $(m1, m3)$, and $(m1, m4)$.

The main feature of CovCon is to maintain and exploit information about interleaving coverage. Our work builds on concurrent method pairs, a measure that stores how often a particular pair of methods has executed concurrently (Section IV-A). Figure 1d lists all pairs of methods of the class under test. For each pair, the approach maintains two counters: How often CovCon has already generated a test that tries to call the methods concurrently, and how often the methods

(a) Class under test:

```
class C {
  List li = ...;
  synchronized void m1() {
    ...
  }
  synchronized void m2() {
    if (...) li = null;
  }
  int m3() {
    if (li == null)
      return 0;
    m4();
    synchronized(this) {
      return li.size();
    }
  }
  void m4() { ... }
}
```

(b) Generated test:



(c) Trace of test execution:

enter m1	
exit m1	
enter m3	
	enter m3
	enter m4
	exit m4
	exit m3
	enter m1
enter m4	
exit m4	
exit m3	
	exit m1

(d) Coverage information:

Method pair	Coverage (tried/covered) Initially	Coverage (tried/covered) After 20 tests	Priority after 20 tests
m1,m1	0 / 0	2 / 0	4
m1,m2	0 / 0	2 / 0	4
m1,m3	0 / 0	2 / 4	4
m1,m4	0 / 0	2 / 7	10
m2,m2	0 / 0	2 / 0	4
m2,m3	0 / 0	2 / 2	2
m2,m4	0 / 0	2 / 8	12
m3,m3	0 / 0	2 / 2	2
m3,m4	0 / 0	2 / 3	2
m4,m4	0 / 0	2 / 5	6

Fig. 1: Example to illustrate coverage-guided generation of concurrent tests.

have indeed executed concurrently. These counters may differ because a particular execution may not schedule two methods concurrently, because a pair of methods may never or rarely be able to execute concurrently due to synchronization, and because a call in a test may call other methods.

Given the execution trace in Figure 1c, CovCon updates the coverage information as follows. It increments the number of tries for $(m1, m3)$ because the test focuses on this pair of methods. Furthermore, it updates the number of times that $(m3, m3)$, $(m3, m4)$, $(m1, m3)$, and $(m1, m4)$ are covered. The third column of Figure 1d illustrates the coverage information that CovCon may gather by generating and executing 20 tests. Some method pairs, such as $(m1, m1)$, have not been covered at all because they are protected by the same lock. Other method pairs, such as $(m2, m4)$, are covered relatively often because $m4$ gets exercised both by tests that directly call $m4$ and by tests that directly call $m3$, increasing its chance to execute concurrently with other methods. Note that even though the number of tries are the same for all method pairs in the example, this need not be the case.

CovCon uses the coverage information to focus the generation of tests on pairs of methods that have not yet been covered or that have been covered less frequently than others (Section IV-C). A naive approach might always pick the pair of methods with the lowest covered count. However, this approach would spend most of its testing efforts on method pairs that cannot execute concurrently, such as $(m1, m2)$. Instead, we present a prioritization technique that accounts both for pairs that cannot be executed concurrently and pairs that are frequently executed concurrently because they are called by other methods. The last column of Figure 1d shows the score that CovCon assigns to each method pair after generating and executing 20 tests. Based on the scores, the approach is most likely to focus on the three pairs $(m2, m3)$, $(m3, m3)$, and $(m3, m4)$ in its next round of test generation, which includes the method pair $(m2, m3)$ that is prone to the atomicity violation.

The testing process illustrated in Figure 1 continues until CovCon detects a thread safety violation or until the user stops the approach when the testing reaches a time-out or a test coverage goal. During this process, the approach continuously updates the coverage information and refines the decisions on which pairs of methods to test next.

III. Background: Concurrent Tests

CovCon analyzes a supposedly thread-safe class under test by generating *concurrent tests*. Such a test consists of a *sequential prefix*, executed in a single thread, and multiple *concurrent suffixes*, executed in concurrent threads. The prefix instantiates the class under test and possibly calls additional methods to bring the instance into a state that may expose errors. The suffixes all share the single instance of the class under test created by the prefix. Similar to prior work [33], [39], [40], this paper focuses on tests with two suffixes.

To check whether executing a concurrent test exposes a concurrency bug, we build upon the *thread safety oracle* [39]. It checks whether the concurrent execution of a test leads to an exception or a deadlock, and if it does, checks whether the same exception or deadlock also occurs in any of the method-level linearizations of the test. If the exception or deadlock occurs only in a concurrent execution but not in any of the linearizations, then the oracle reports a thread safety violation. Because thread safety guarantees that each concurrent behavior of a class corresponds to a method-level linearization, the thread safety oracle reports a warning only if the class indeed suffers from a concurrency bug. The oracle may miss bugs that do not manifest as an exception or a deadlock and bugs not triggered during the concurrent execution due to non-determinism.

IV. Approach

The following describes the details of CovCon, a coverage-guided approach to generate concurrent tests. Given a supposedly thread-safe class under test, the approach repeatedly performs two steps until it finds a thread safety violation or until the user stops the approach:

1) Test generation. Create a multi-threaded unit test that exercises the class under test.

2) Test execution and validation. Execute the test and decide whether it exposes a thread safety violation.

The main contribution of this work is to improve the efficiency and effectiveness of the first step by guiding the test generator toward tests that will cover not yet or infrequently explored interleaving behaviors. We first present the interleaving coverage metric used by CovCon (Section IV-A). CovCon gathers coverage information during the execution of tests by instrumenting the class under test and by analyzing execution traces (Section IV-B). Based on the gathered coverage information, CovCon assigns priorities to interleavings, giving highest priority to interleavings that require more thorough testing (Section IV-C). Finally, CovCon uses the priorities to steer the test generation toward not yet or infrequently covered concurrent behaviors (Section IV-D).

A. Interleaving Coverage

A major challenge for coverage-guided test generation is to find a coverage metric that is both efficient and practical to use. Efficiency refers to the cost of computing the *coverage domain*, i.e., the set of all interleaving behaviors that clients of a thread-safe class can explore, the cost of gathering *coverage facts*, i.e., which of these interleaving behaviors have been covered by test executions, and the cost of synthesizing a test case to satisfy a particular coverage requirement. Efficiency is critical because otherwise the benefits of using coverage information may not outweigh the cost, compared to a coverage-oblivious approach. It is also critical that the coverage domain is neither too small nor too big. Otherwise, the coverage goal is either too easy or too difficult to achieve in practice.

A naive approach would be to compute all possible instruction-level interleavings via static analysis, and to measure which of these interleavings are covered by an execution. Unfortunately, precisely computing this coverage domain is computationally expensive and gathering coverage facts would impose huge runtime overhead. Moreover, covering all instruction-level interleavings that clients of a class may explore is practically infeasible because the coverage domain would be very large, or even infinite. Recent coverage-based approaches [51], [55] focus on partial-order interleavings, and hence address the practicality issue of the naive approach, but cannot address the efficiency issues.

To enable CovCon to efficiently compute interleaving coverage, we use an approximate interleaving coverage metric. The metric, called *concurrent method pairs*, measures which pairs of methods are executed concurrently.[1] Each concurrent method pair consists of two publicly accessible methods of the class under test. Concurrent method pairs are inspired by Deng et al.'s concurrent function pairs [13], which was proposed to reduce the dynamic bug-detection cost for predefined inputs. Our work is the first to apply the metric to object-oriented programs and to adapt it to the problem of generating concurrent tests.

The concurrent method pair metric is both efficient and practical to use. Computing the coverage domain is straightforward as it requires only the set of publicly accessible methods of the class under test. Likewise, gathering coverage facts is computationally inexpensive, as we show in Sections IV-B. Finally, the set of concurrent method pairs is finite and can be covered in reasonable time (Section VI-F). The following section describes how we compute and use this metric.

B. Gathering Coverage Information

CovCon keeps track of three kinds of coverage related information:

- The set $\mathcal{P}$ of concurrent method pairs that could potentially be covered by test executions.
- A map $\mathcal{C}$ that assigns each method pair $p \in \mathcal{P}$ to the number of times that the pair has been covered, i.e., how often the two methods in p have been executed concurrently. We call $\mathcal{C}(p)$ the *covered count* of p.
- A map $\mathcal{R}$ that assigns each method pair $p \in \mathcal{P}$ to the number of tests where the methods in p are directly called in different suffixes, i.e., the number of tests that are designed to try increasing the covered count of p. We call $\mathcal{R}(p)$ the *tried count* of p.

$\mathcal{P}$ is computed as the set of all pairs of publicly accessible methods of the class under test.[2] $\mathcal{R}$ is updated whenever a method pair p is selected as the prioritized method pair (Section IV-C). Updating $\mathcal{C}$ requires instrumentation and trace analysis for each test execution, which will be discussed in detail in the remainder of this subsection.

1) Instrumentation: To detect which methods of the class under test execute concurrently, CovCon instruments each publicly accessible method defined in the class and its superclasses. The instrumentation records the start and the end of every invocation of the method into an execution trace.

Definition 1 (Execution trace)
The trace created while executing a concurrent test is a sequence of trace entries. Each entry is a tuple (e, m, o, h, s)*, where*

- e *is the kind of event ("start" or "end"),*
- m *is the fully qualified signature of the current method,*
- o *is a unique identifier of the object on which the method is called (or "null" for static methods),*
- h *is a unique identifier of the current thread, and*
- s *is a global, logical timestamp.*

To obtain the global timestamp, CovCon maintains a global counter that serves as a logical clock shared among all threads. Each time when the instrumented code creates a new trace entry, it increments the counter and records the incremented counter as the current timestamp.

[1] We consider $(m1, m2)$ and $(m2, m1)$ to be the same method pair because the order is irrelevant for concurrency bug detection. This terminology differs from the mathematical definition of "pair" but is in line with previous work [13].

[2] A possible extension of CovCon could be to refine $\mathcal{P}$ via a static analysis that prunes methods that cannot execute concurrently due to synchronization.

Algorithm 1 Analyze execution trace to update information about covered method pairs.

Input: Execution trace $trace$ and covered counts $\mathcal{C}$
Output: Updated covered counts $\mathcal{C}$

```
1:  M_cur ← ∅ {maps threads to currently executing methods}
2:  for all (e, m, o, h, s) ∈ trace do
3:    if e = "start" then
4:      for all h' ∈ M_cur do
5:        if h' ≠ h then
6:          for all m' ∈ M_cur(h') do
7:            p ← sort(m, m')
8:            C(p) ← C(p) + 1
9:      M_cur(h).push(m)
10:   else if e = "end" then
11:     M_cur(h).pop()
```

To detect the start of a method execution, CovCon inserts a call statement at the beginning of the method. Detecting the end of a method execution is slightly more complicated because a method may have multiple exit points, including exit points caused by checked and unchecked exceptions. To consider all possible exit points of a method, CovCon surrounds the original method body with a `try-catch-finally` statement, which catches and re-throws all exceptions. The `finally` block, which is always executed at the end of the method, records the trace entry for exiting the method.

2) Analyzing the Execution Trace: Based on the execution trace of a test, CovCon determines which pairs of methods have executed concurrently and updates their covered counts accordingly. Algorithm 1 summarizes the trace analysis. The algorithm iterates through all trace entries and maintains a stack of currently executing methods for each thread. For each event that represents the start of a method m, the algorithm pushes m to the stack of the current thread h (line 9). Moreover, the algorithm checks which other methods m' execute concurrently, i.e., in another thread h', and increments the covered count for the pair (m, m') (line 8). Since the order of methods is irrelevant for the concurrent coverage pair metric, the algorithm represents each pair in a canonical format by sorting the method signatures. When the algorithm observes the end of a method m, it pops m from the stack of the current thread (line 11). The algorithm considers the general case of an arbitrary number of concurrently executing methods, even though this work focuses on tests with only two suffixes. One reason is that there may be more than two concurrently executing methods when a method called in a suffix spawns additional threads.

C. Prioritizing Method Pairs

In the following, we present how CovCon decides which method pairs to prioritize during test generation. The input to this step is the coverage information gathered during prior executions of generated tests, specifically, the maps $\mathcal{C}$ and $\mathcal{R}$, which store the covered counts and tried counts of method pairs, respectively.

The prioritization addresses three challenges:

- (C1) To maximize the chance to discover a new thread safety violation, CovCon should generate tests to exercise pairs of methods that have not yet been tested extensively. In particular, pairs that have not yet been tried at all should have highest priority.
- (C2) To avoid spending testing effort on pairs of methods that cannot be executed concurrently, CovCon should reduce the priority of such pairs. For example, a method pair may never (rarely) execute concurrently if (significant parts of) both methods are protected by the same lock.
- (C3) To avoid spending testing effort on pairs of methods that are frequently executed concurrently, even though only few tests have directly called them in concurrent suffixes, CovCon should reduce the priority of such pairs. For example, such a method pair may occur if helper methods that are called by many other methods often execute concurrently.

An obvious solution would be to assign highest priority to method pairs p with the lowest covered count $\mathcal{C}(p)$. This approach addresses C1 but fails to address C2 and C3. For example, such an approach would repeatedly attempt to test $(m1, m2)$ from Figure 1a, which cannot execute concurrently due to synchronization.

To address all three challenges, we present a prioritization score that assigns an integer value to each method pair.

Definition 2 (Prioritization score)
Given a method pair $p \in \mathcal{P}$, let r be its tried count $\mathcal{R}(p)$ and let c be its covered count $\mathcal{C}(p)$. The prioritization score $S(p)$ *is the following:*

- *If $r = 0$, then $S(p) = 0$.*
- *If $r \neq 0$, then $S(p) = max(|r - c|, 1) \cdot max(r, 1)$.*

CovCon prioritizes method pairs with a low score. The first case of Definition 2 ($r = 0$) partly addresses C1 by prioritizing pairs that have not yet been tried to cover by any test. The first part of the second case ($max(|r-c|, 1)$ addresses C2 and C3 by assigning a higher score (i.e., lower priority) to pairs that are either infrequently covered despite frequent attempts to cover them or frequently covered despite few attempts to cover them. Finally, the second part of the second case ($max(r, 1)$) addresses C1 by assigning a lower score (i.e., higher priority) to pairs that have not yet been attempted to cover frequently.

For illustration, Table I shows the scores for tried counts and covered counts ranging from zero to ten. For a pair that has not yet been called concurrently in any test, the score is zero, i.e., this pair gets highest priority (C1). In contrast, a pair with a high tried count and a relatively low covered count, illustrated by the left-bottom corner cells in Table I, gets a high score, i.e., this pair gets low priority (C2). A pair that has a high covered count despite having a relatively low tried count, illustrated by the gray cells on the right side of Table I, also gets a relatively high score, i.e., a low priority (C3).

TABLE I: Prioritization score depending on covered count (x axis) and tried count (y axis). Lower score implies higher probability to be focused on in future tests.

$\mathcal{R}(p)$ \ $\mathcal{C}(p)$	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	2	3	4	5	6	7	8	9
2	4	2	2	2	4	6	8	10	12	14	16
3	9	6	3	3	3	6	9	12	15	18	21
4	16	12	8	4	4	4	8	12	16	20	24
5	25	20	15	10	5	5	5	10	15	20	25
6	36	30	24	18	12	6	6	6	12	18	24
7	49	42	35	28	21	14	7	7	7	14	21
8	64	56	48	40	32	24	16	8	8	8	16
9	81	72	63	54	45	36	27	18	9	9	9
10	100	90	80	70	60	50	40	30	20	10	10

Instead of probabilistically prioritizing method pairs, CovCon could also completely avoid re-testing already covered pairs. The rationale for using prioritization is that the ability of a test to expose a bug depends not only on which methods execute concurrently, but also, e.g., on the arguments provided to the methods. As another alternative to Definition 2, CovCon could use a static analysis to determine which pairs of methods can execute concurrently and which pairs of methods are likely to be called frequently by other concurrently executing methods. We have initially considered this option but found that an inexpensive static analysis does not provide an advantage over the prioritization score. In contrast, the benefit of the presented prioritization score is threefold. First, it avoids a static analysis of the class under test and its dependences. This design makes the approach both conceptually and computationally simpler, which is critical to the overall testing effectiveness. Second, the approach is language-agnostic and can be easily applied to other languages than Java, whereas adapting a static analysis to another language is non-trivial. Finally, static analysis may incorrectly judge a pair of methods that cannot execute concurrently due to ad hoc synchronization [63], but our prioritization score would not be affected.

D. Creating Tests

After CovCon decides to prioritize a method pair $p_{prio} = (m1, m2)$, it creates tests to focus on it. The execution of these tests aims at triggering the concurrent execution of $m1$ and $m2$ and at increasing the interleaving coverage. This goal is non-trivial to achieve, as we need to build up some meaningful state for $m1$ and $m2$ to execute upon and, more importantly, we want to improve interleaving coverage as much as we can with as little overhead as we can.

CovCon creates tests that have a sequential prefix and two concurrent suffixes. It builds upon the test generation approach from [39] to create the prefix that instantiates the class under test and calls up to k additional methods with appropriate arguments to bring the tested object into a state that may reveal concurrency errors. For each prioritized method pair, CovCon creates two prefixes: One without any additional calls, i.e., $k = 0$, and one with up to $k = 5$ additional calls. The rationale is that some concurrency bugs can only be triggered after bringing the object into a bug-exposing state by invoking a sequence of calls, whereas other bugs may show only on a freshly instantiated instance. For example, a class that represents a file handle may become unusable after calling a method that closes the file.

The suffix generation is more tricky. The first question is which methods should be called from the suffix. The existing test generation approach from [39] generates the suffixes using randomly selected methods, which clearly does not fit the prioritization goal here. Instead, CovCon generates suffixes that call only methods from the pair p_{prio}. The second question is how to call these methods $m1$ and $m2$ to improve the test coverage. One possible approach is to create a suffix $m1(), m1(), \ldots$ and a suffix $m2(), m2(), \ldots$, and to combine these suffixes into a concurrent test. Such a test is likely to improve coverage for the pair $(m1, m2)$ but unlikely to also cover $(m1, m1)$ and $(m2, m2)$. As an alternative, the test generator could create a test that runs the same suffix $m1(), m2(), m1(), \ldots$ in two concurrent threads. Such a test is not only likely to cover the pair $(m1, m2)$ but may also cover $(m1, m1)$ and $(m2, m2)$. Unfortunately, the chance to trigger an execution where $m1$ and $m2$ run concurrently is relatively low because both threads start by calling $m1$ and alternate between the two methods in the same order.

Instead of these two naive approaches, CovCon creates two suffixes, a suffix $m1(), m2(), m1(), \ldots$ and a suffix $m2(), m1(), m2(), \ldots$, and combines the prefix with these two suffixes into a test. That is, each suffix calls the methods of p_{prio} alternately. The benefit of starting this alternation with different methods in the two suffixes is to maximize the chance that the pair of methods $(m1, m2)$ executes concurrently. One could achieve the same goal by inserting sleeps or synchronization operations [14]. In contrast, CovCon's approach is more lightweight as it does not introduce any overhead.

CovCon creates suffixes that have up to l_{max} calls per suffix. For the first five tests created for a particular pair p_{prio}, we set $l_{max} = 2$. For example, when $l_{max} = 2$, CovCon creates a test with a suffix $m1(), m2()$ and another suffix $m2(), m1()$. After the first five tests, we set $l_{max} = 5$. The rationale for varying l_{max} is to quickly identify bugs that can be triggered with short suffixes, and to then explore longer suffixes that allow for more interleavings but that also require the thread safety oracle to analyze more linearizations.

E. Putting Everything Together

Algorithm 2 summarizes the CovCon test generation approach. Given a supposedly thread-safe class under test, the algorithm generates and executes tests until it finds a thread safety violation, or until the user terminates the automatic testing after a timeout or a coverage goal is reached.

At first, the algorithm computes the set $\mathcal{P}$ of concurrent method pairs for the given class (line 1). Then, the algorithm initializes the maps $\mathcal{C}$ and $\mathcal{R}$ with zero for each method pair

Algorithm 2 Coverage-guided generated of concurrent tests.

Input: Class under test C
Output: Thread safety violation $viol$

```
1:  P ← computeCMPs(C)
2:  for all p ∈ P do
3:      R(p) ← 0
4:      C(p) ← 0
5:  while true do
6:      p_prio ← prioritizeCMP(C, R)
7:      updateTriedCounts(p, R)
8:      T ← generateTests(p_prio)
9:      for all T ∈ T do
10:         (trace, viol) ← executeAndValidate(T)
11:         if viol ≠ none then
12:             return viol
13:         updateCoveredCounts(trace, C)
```

in the set $\mathcal{P}$, to indicate that no pair has been tried yet and that no pair has been covered yet (lines 2 to 4).

The main part of the algorithm repeatedly generates, executes, and validates tests, while gathering coverage information to steer the test generation toward not yet covered method pairs in the set $\mathcal{P}$ (lines 5 to 13). This part consists of three main steps. First, function $prioritizeCMP()$ selects, based on the coverage information gathered in $\mathcal{C}$ and $\mathcal{R}$ and the prioritization score, a method pair p_{prio} to focus on next and updates the tried count of p_{prio}, as discussed in Section IV-C. Second, function $generateTests()$ creates a set $\mathcal{T}$ of concurrent tests that focus on the prioritized pair p_{prio}, as discussed in Section IV-D. Third, the algorithm executes each test in $\mathcal{T}$ and checks whether it exposes a thread safety violation using the thread safety oracle (Section III). If the oracle reports a thread safety violation (line 10), then the algorithm terminates. After each test execution, the algorithm updates $\mathcal{C}$ to increase the covered counts of method pairs that have executed concurrently (line 13), as explained in Section IV-B.

There is one optimization that we have not discussed yet. Since there are often multiple method pairs with the same lowest prioritization score, as an optimization, CovCon computes the priorities once for all method pairs and stores all method pairs with the lowest score into a set $\mathcal{P}_{prio}$ of prioritized pairs. When Algorithm 2 calls $prioritizeCMP()$, the function checks whether $\mathcal{P}_{prio}$ is non-empty and if so, removes and returns a random element of $\mathcal{P}_{prio}$. Only if the set of pre-computed prioritized method pairs becomes empty, CovCon computes the priority scores for all method pairs and fills $\mathcal{P}_{prio}$ again. The benefit of this optimization is to significantly reduce the number of computations of prioritization scores. The downside is that prioritization is not always based on the most recent coverage information. We find that, in practice, the benefits of the optimization outweigh this disadvantage.

V. Implementation

The implementation of CovCon is available for download.[3] To instrument classes under test, CovCon builds upon the Eclipse Java Development Tools. The instrumented classes under test write an execution trace into a file, which is then analyzed to update the coverage information and to prioritize method pairs for the next round of test generation. The test generation is implemented based on an existing framework for generating concurrent tests.[4] To execute generated tests, we use the default scheduler of the Java VM, because this work focuses on test generation, not on exploring test executions. More sophisticated test execution approaches, such as systematic exploration of interleavings [31] or active testing [25] could be plugged into our implementation.

VI. Evaluation

To evaluate CovCon, we apply our implementation to 18 supposedly thread-safe classes with known, real-world concurrency bugs, which have been used to evaluate prior work [39], [55]. We compare the effectiveness and efficiency of CovCon to five state of the art approaches for generating concurrent unit tests [39], [42]–[44], [55]. To the best of our knowledge, this is the largest experimental comparison of approaches for generating concurrent tests, both in terms of number of classes and number of approaches.

A. Experimental Setup

Table II lists the classes under test. The concurrency bugs in these classes include data races, atomicity violations, and deadlocks, as shown in the "Bug" column of the table. The benchmarks include all classes used in [39], except for one because we could not obtain the source code, and all classes used in [55]. The table gives the number of lines of code, methods, and potential concurrent method pairs $\mathcal{P}$ for each class. These values include both the class itself and its superclasses (but not `java.lang.Object`) because this is the code tested by CovCon.

We compare our approach to random-based test generation using ConTeGe [39] and to coverage-based test generation using AutoConTest [55]. To compare our approach with sequential test-based approaches, we use Narada [44], Intruder [43], and Omen [42], which detect data races, atomicity violations, and deadlocks, respectively. In contrast, CovCon detects all three kinds of concurrency bugs. For a fair comparison, we combine the three existing tools into a single tool that starts all three individual tools in parallel and reports a bug as soon as one of the individual tools has found a bug. We call the combined sequential test-based approach Nainom, standing for Narada, Intruder, and Omen.[5] Since Nainom relies on sequential seed tests to generate concurrent tests, we need to provide sequential tests for each class under test. For a fair

[3] https://github.com/michaelpradel/ConTeGe/tree/CovCon

[4] https://github.com/michaelpradel/ConTeGe

[5] As an alternative to running the three tools in parallel, we also experiment with applying them in a round-robin fashion. Since the parallel version of Nainom is slightly more efficient, we report these results.

comparison with CovCon, which does not require creating any sequential tests, and to avoid biasing Nainom's behavior by providing a particular set of manually written sequential tests, we generate sequential tests via feedback-directed, random test generation, similar to Randoop [35]. The generated tests call each public method of the class under test at least once, as suggested by the authors of Nainom.[6]

The efficiency of automated concurrency testing depends not only on the generated tests but also on how these tests are executed. Both CovCon and ConTeGe execute tests using the default scheduler of the Java VM. In contrast, Nainom and AutoTestGen build upon more sophisticated approaches for exploring interleavings that are likely to reveal concurrency bugs quicker [27], [29], [38], [47]. Since targeted strategies to explore the space of possible interleavings are known to outperform repeated execution [31], we expect this difference in the setups to work in favor of Nainom and AutoConTest.

For all approaches, we measure the time to detect a bug as the total time required by each tool, i.e., including test generation and test execution or exploration time. We use a timeout of one hour per run on a particular class. Because all approaches involve non-deterministic decisions, we repeat each experiment ten times with a different random seed. To determine whether the time required by two approaches to detect a bug differs in a statistically significant way, we compute the confidence intervals (90% confidence) of the measured times and check whether these intervals overlap. All experiments are done on an Intel Core i7-4790 CPU with eight 3.6GHz cores and 32GB memory, running Ubuntu 14.04 LTS (64 bit).

B. Bug Finding Capabilities

CovCon detects thread safety bugs in 17 of the 18 classes under test. The detected bugs include data races, atomicity violations, and deadlocks. In comparison, the existing ConTeGe, Nainom, and AutoConTest approaches fail to detect a bug in 3, 10, and 14 classes, respectively. Table II lists the average time required to detect a bug, where "*3,600*" marks that an approach fails to find a bug within the one-hour timeout for all ten runs of the approach. A reason why AutoConTest, which shares the idea of coverage-directed test generation, fails to detect various bugs is that it focuses on a particular kind of atomicity violation. In contrast, CovCon uses a generic coverage goal suitable for any kind of concurrency bug.

C. Efficiency

An important criterion for applying automated concurrency testing in practice is the time required to detect a bug. We compare this time for CovCon and the state of the art approaches. On average, CovCon requires 314 seconds to find a bug in a given class. In contrast, all existing approaches require significantly more time and often exceed our one-hour timeout, as we discuss in detail in the following.

Figure 2 shows, for each class under test, the average speedup of CovCon over the existing approaches. Speedup

[6]Personal communication, September 2015.

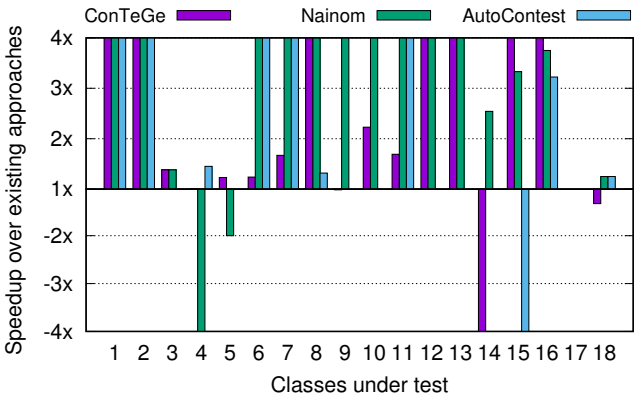


Fig. 2: Speedup of CovCon over existing approaches in terms of the average time required to detect a bug. Note that several speedup factors exceed the displayed $[-4, 4]$ range.

here means the average time taken by the slower approach divided by the average time taken by the faster approach. We present cases where CovCon is the faster approach as a positive speedup and cases where CovCon is a slower approach as a negative speedup. Several speedups exceeds the $[-4, 4]$ range that is shown in the figure. For example, for class 1 (`Logger`), CovCon is at least 4x faster than all three existing approaches. For class 15 (`Vector`), CovCon is at least 4x faster than ConTeGe, 3.3x faster than Nainom, and at least 4x slower than AutoConTest. Overall, CovCon clearly outperforms the existing approaches. We discuss exceptions to this overall result below.

Table II provides detailed execution times and speedup factors for all classes. The columns in the "Time to bug" block show how long each approach takes to detect a bug in a particular class. For example, CovCon requires 159 seconds to detect a bug in `Logger`, whereas ConTeGe takes 2,810 seconds, on average. We mark classes that cannot be analyzed by a particular tool because the tool crashes with "—". The last block in Table II shows the speedup of CovCon over the existing approaches. For example, CovCon is 17.67 times faster in finding a bug in `Logger` than ConTeGe. Because we stop the existing approaches for several classes after the one-hour timeout, the speedup is an underapproximation. For example, CovCon is *at least* 22.64 times faster than Nainom for class `Logger`. To ease the presentation, the table color-codes the speedups, where a darker colors means a higher speedup. We also mark which speedups are statistically significant by printing these speedups in bold.

Contrary to most classes, there are three classes where an existing approach clearly outperforms CovCon. The `XStream` class is an outlier in the comparison with ConTeGe. The reason is that CovCon's prioritization mechanism is less successful than for the other classes in steering testing toward higher interleaving coverage (Section VI-F). The `Vector` 1.1.7 class stands out in the comparison with AutoConTest. The reason is that this class suffers from an atomicity violation that is unlikely to be triggered yet perfectly matches the bug

TABLE II: Classes under test and the time required to find bugs in them with CovCon and existing approach. "*3,600*" indicates a timeout. The last columns color-code the speedup of CovCon over the existing approaches into three categories: Faster ($>$ 1.0x and $<$2x), Much faster ($\geq$ 2x and $<$4x), A lot faster ($\geq$ 4x). Speedups printed in bold are statistically significant.

ID	Class	LoC	Meth.	CMPs	Bug	Time to bug (seconds, avg.)				Speedup of CovCon over ..		
						CovCon	ConTeGe	Nainom	Auto-ConTest	ConTeGe	Nainom	Auto-ConTest
1	BufferedInputStream (JDK 1.1)	239	9	45	Atom.	3	178	149	743	**59.33**	**49.67**	247.67
2	Logger (JDK 1.4.1)	531	44	990	Atom.	159	2,810	*3,600*	*3,600*	**17.67**	**22.64**	**22.64**
3	SynchronizedMap (JDK 1.4.2)	79	15	120	Deadl.	2,602	*3,600*	*3,600*	—	**1.38**	**1.38**	—
4	ConcurrentHashMap (JDK 1.6.0)	972	22	253	Atom.	2,477	2,504	346	*3,600*	1.01	**-7.16**	**1.45**
5	StringBuffer (JDK 1.6.0)	789	52	1,378	Atom.	560	689	280	—	1.23	-2.00	—
6	TimeSeries (JFreeChart 0.98)	359	41	861	Race	59	73	385	*3,600*	1.24	**6.53**	**61.02**
7	XYSeries (JFreeChart 0.9.8)	200	25	325	Race	9	15	174	*3,600*	1.67	**19.33**	**400.00**
8	NumberAxis (JFreeChart 0.9.12)	1,662	110	6,105	Atom.	72	374	*3,600*	95	**5.19**	**50.00**	1.32
9	PeriodAxis (JFreeChart 1.0.1)	1,975	125	7,875	Race	26	25	*3,600*	—	-1.04	**138.46**	—
10	XYPlot (JFreeChart 1.0.9)	3,080	217	23,653	Race	325	722	*3,600*	—	2.22	**11.08**	—
11	Day (JFreeChart 1.0.13)	267	26	351	Race	26	44	217	*3,600*	1.69	**8.35**	**138.46**
12	PerUserPoolDataSource (DBCP 1.4)	719	65	2,145	Race	28	738	*3,600*	—	**26.36**	**128.57**	—
13	SharedPoolDataSource (DBCP 1.4)	546	51	1,326	Race	17	433	*3,600*	—	**25.47**	**211.76**	—
14	XStream (XStream 1.4.1)	926	66	2,211	Race	1,416	55	*3,600*	—	**-25.76**	**2.54**	—
15	Vector (JDK 1.1.7)	216	24	300	Atom.	793	*3,600*	2,638	103	**4.54**	**3.33**	**-7.70**
16	Vector (JDK 1.4.2)	786	45	1035	Atom.	134	612	503	432	4.57	3.75	3.22
17	IntRange (Apache Commons 2.4)	276	26	351	Atom.	*3,600*	*3,600*	*3,600*	*3,600*	1.00	1.00	1.00
18	AsMap (Google Commons 1.0)	409	15	120	Atom.	2,881	2,166	*3,600*	*3,600*	-1.33	1.25	1.25

pattern that AutoConTest targets. Finally, Naimon is faster than CovCon in detecting a bug in `ConcurrentHashMap`. Despite these outliers, the overall results show CovCon to be significantly more efficient than state of the art approaches.

D. Number of Tests

The overall effectiveness of automated testing depends both on how effective individual tests are and on how long it takes to generate such tests [3]. CovCon may generate and execute several hundreds or even thousands of tests before finding a bug, whereas Nainom and AutoConTest typically require less then a hundred tests. However, as our results in Section VI-C show, CovCon clearly outperforms Nainom and AutoConTest in terms of the overall effectiveness because the higher effectiveness of their generated tests comes at a significantly higher cost for generating each test.

E. Comparison with Naive Prioritization

We compare our prioritization of method pairs (Section IV-C) with a naive prioritization that always selects the method pair that has been tried the least number of times. If multiple such pairs exist, the approach randomly picks from all pairs with the minimum tried count. The efficiency of this naive prioritization is either similar or worse than for our prioritization. For example, the time required to find a bug with the naive prioritization is 450x, 13x, and 5x higher than with CovCon for `BufferedInputStream`, `NumberAxis`, and `Day`, respectively.

F. Steering Toward Uncovered Interleavings

To validate our hypothesis that CovCon effectively steers toward tests that cover not-yet-covered interleavings, we compare the interleaving coverage achieved when executing tests

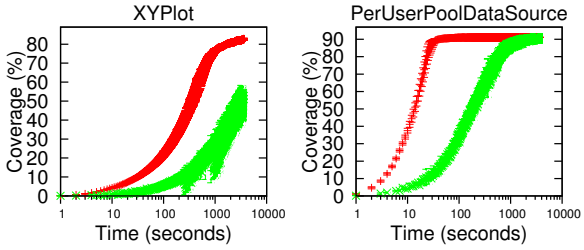


Fig. 3: Interleaving coverage over time for two representative classes. The upper, red curve is for CovCon; the lower, green curve is for ConTeGe. The x axis is log-scaled.

generated by CovCon and the random-based ConTeGe. Each experiment runs either until 100% coverage is reached or until one hour is over, and each experiment is repeated 10 times. Figure 3 shows the results for two representative classes. Each graph gives the percentage of covered concurrent method pairs over time, and it provides the mean and confidence interval for each data point. The results show that a coverage-guided approach indeed achieves higher interleaving coverage in less time. For most classes, the interleaving coverage achieved through both automated testing approaches saturates at some point, and CovCon reaches this point earlier than ConTeGe. Note that reaching 100% coverage may be infeasible, e.g., when two methods cannot execute concurrently because they are protected by the same lock.

G. Breakdown of Overall Execution Times

Figure 4 shows how much time CovCon spends on (i) generating tests, (ii) executing tests, (iii) analyzing the execution trace to update coverage information and prioritizing concurrent method pairs, and (iv) checking for thread safety

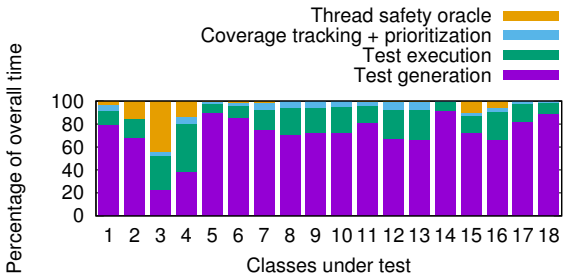


Fig. 4: Breakdown of time spent in different parts of CovCon.

violations by executing linearizations of concurrent test executions. Most time is spent in the test generation part of the approach, which underlines the need to steer this part toward tests that explore not yet covered behavior. Class 3 (`SynchronizedMap`) is an outlier because an unusually high amount of time is spent in the thread safety oracle. The reason is that about 10% of all generated tests cause an exception or a deadlock, so the thread safety oracle must check an unusually high number of linearizations to check whether the misbehavior is due to a thread safety violation.

VII. Related Work

a) Test Generation: Section VI experimentally compares our work to recent random-based [39], sequential test-based [42]–[44], and coverage-based [55] generators of concurrent tests. Other existing approaches include the random-based Ballerina [33] and the coverage-based ConSuite [51]. Conceptually, CovCon improves upon random-based approaches by using interleaving coverage to steer the test generation toward infrequently tested interleavings. In contrast to sequential test-based approaches, CovCon does not require sequential tests but has the class under test as its only input. The main differences of our work to existing coverage-based approaches are (i) to use a conceptually and computationally simpler coverage metric and (ii) to work well for arbitrary kinds of concurrency bugs.

SpeedGun [40] generates concurrent tests that expose performance problems. Schimmel et al. [46] combine test generation with a static pre-analysis of the code to find methods that access shared data. One could integrate such a static analysis into CovCon to reduce the set of concurrent methods pairs to consider. Claessen et al. propose a test generator for Erlang programs [10]. It requires a user to provide a finite state model of the tested program, whereas our approach is fully automatic. MultithreadedTC [41] and IMUnit [22] are unit testing frameworks to manually write concurrent unit tests. In contrast, CovCon generates tests automatically. Besides concurrent tests, there is a long history of approaches for generating sequential tests, e.g., based on random testing [9], [12], [35], symbolic execution [58], [62], concolic execution [7], [19], [48], and genetic algorithms [18].

b) Interleaving Coverage: Several concurrency-related coverage metrics have been proposed. Taylor et al. pioneered by proposing a hierarchy of concurrency coverage criteria [53]. Bron et al. discuss coverage metrics that are useful for human developers that write concurrent tests [4]. Other work adapts the definition-use path coverage criterion to concurrent programs [64]. Lu et al. theoretically analyze the cost of seven interleaving coverage criteria [30]. Our work differs from these approaches by using interleaving coverage as a feedback mechanism for generating concurrent tests. The coverage criterion that CovCon builds on, concurrent method pairs, is inspired by Deng et al.'s concurrent function pairs [13]. They use the criterion to select from pre-defined inputs to analyze with dynamic concurrency bug detectors, whereas we use it to generate new inputs. Since the design goals are different, how they measure coverage and use coverage to conduct prioritization are also completely different from us.

c) Schedule Exploration: A single program and input may have many different schedules; approaches for exhaustive [28], [57], bounded [31], change-based [23], [54], and heuristic [8], [11] exploration of schedules have been proposed. Active testing searches for potential concurrency bugs with a static or dynamic analysis and forces schedules that may trigger the potential bugs [6], [17], [25], [27], [29], [37], [38], [47], [50], [66], [67]. Other work uses concurrent coverage metrics to select and prioritize schedules to explore [21], [52], [60], [65]. Selecting schedules and generating tests are two orthogonal problems. CovCon can be combined with existing schedule exploration approaches, and such a combination may further reduce the time required to detect bugs.

d) Static Analysis, Synthesis, Verification: One alternative to testing concurrent software are synthesis and verification [20], [49], [56]. These approaches rely on a formal specification that describes the desired behavior of the synthesized program. Instead, CovCon can be directly applied to existing thread-safe classes. Another alternative are static analyses, e.g., to detect data races [59] or deadlocks [32], [61]. In contrast to those, CovCon covers multiple kinds of concurrency bugs and guarantees to report only true positives.

VIII. Conclusion

This paper presents CovCon, an efficient, coverage-guided generator of concurrent tests. The main idea is to measure which interleavings are not yet covered by existing tests and to generate tests likely to cover these interleavings. Key to the success of CovCon is the insight that concurrent method pairs provide an inexpensive coverage metric that is effective in steering test generation toward not yet covered interleavings. An implementation of the CovCon approach reveals 17 concurrency bugs in 18 thread-safe classes from popular Java libraries. Furthermore, we show that CovCon reduces the time to detect bugs over existing random-based and sequential test-based approaches in 38 of 47 cases, with speedups of at least 4x in 22 of 47 cases.

Acknowledgments

We thank the anonymous reviewers and Andrew Habib for their comments. Our research has been supported by the DFG within ConcSys, by the BMBF and the HMWK within CRISP, by NSF grants CCF-1439091 and CNS-1563956, and by the CERES Center for Unstoppable Computing.

REFERENCES

[1] C. Artho, K. Havelund, and A. Biere. High-level data races. *Software Testing, Verification and Reliability*, 13(4):207–227, 2003.

[2] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A framework for automated testing of JavaScript web applications. In *ICSE*, pages 571–580, 2011.

[3] M. Böhme and S. Paul. On the efficiency of automated testing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 632–642, 2014.

[4] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur. Applications of synchronization coverage. In *Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 206–212. ACM, 2005.

[5] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-Up: a complete and automatic linearizability checker. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 330–340. ACM, 2010.

[6] J. Burnim, Koushik, and S. C. Stergiou. Testing concurrent programs on relaxed memory models. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 122–132, 2011.

[7] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224. USENIX, 2008.

[8] J. Chen and S. MacDonald. Testing concurrent programs using value schedules. In *Conference on Automated Software Engineering (ASE)*, pages 313–322. ACM, 2007.

[9] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. ARTOO: adaptive random testing for object-oriented software. In *International Conference on Software Engineering (ICSE)*, pages 71–80. ACM, 2008.

[10] K. Claessen, M. Palka, N. Smallbone, J. Hughes, H. Svensson, T. Arts, and U. T. Wiger. Finding race conditions in Erlang with QuickCheck and PULSE. In *International Conference on Functional programming (ICFP)*, pages 149–160. ACM, 2009.

[11] K. E. Coons, S. Burckhardt, and M. Musuvathi. GAMBIT: effective unit testing for concurrency libraries. In *Symposium on Principles and Practice of Parallel Programming, (PPOPP)*, pages 15–24. ACM, 2010.

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

[13] D. Deng, W. Zhang, and S. Lu. Efficient concurrency-bug detection across inputs. In *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 785–802. ACM, 2013.

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

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

[16] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 121–133. ACM, 2009.

[17] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 293–303. ACM, 2008.

[18] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 147–158. ACM, 2010.

[19] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 213–223. ACM, 2005.

[20] P. Hawkins, A. Aiken, K. Fisher, M. C. Rinard, and M. Sagiv. Concurrent data representation synthesis. In *PLDI*, pages 417–428, 2012.

[21] S. Hong, J. Ahn, S. Park, M. Kim, and M. J. Harrold. Testing concurrent programs to achieve high synchronization coverage. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 210–220. ACM, 2012.

[22] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, and D. Marinov. Improved multithreaded unit testing. In *European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 223–233, 2011.

[23] V. Jagannath, Q. Luo, and D. Marinov. Change-aware preemption prioritization. In *ISSTA*, pages 133–143, 2011.

[24] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang. OCAT: object capture-based automated testing. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 159–170. ACM, 2010.

[25] P. Joshi, M. Naik, C.-S. Park, and K. Sen. CalFuzzer: An extensible active testing framework for concurrent programs. In *Conference on Computer Aided Verification*, pages 675–681. Springer, 2009.

[26] P. Joshi, M. Naik, K. Sen, and D. Gay. An effective dynamic analysis for detecting generalized deadlocks. In *Symposium on Foundations of Software Engineering (FSE)*, pages 327–336. ACM, 2010.

[27] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 110–120. ACM, 2009.

[28] K. Kim, T. Yavuz-Kahveci, and B. A. Sanders. Precise data race detection in a relaxed memory model using heuristic-based model checking. In *Conference on Automated Software Engineering (ASE)*, pages 495–499. IEEE, 2009.

[29] Z. Lai, S.-C. Cheung, and W. K. Chan. Detecting atomic-set serializability violations in multithreaded programs through active randomized testing. In *International Conference on Software Engineering (ICSE)*, pages 235–244. ACM, 2010.

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

[31] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *Symposium on Operating Systems Design and Implementation*, pages 267–280. USENIX, 2008.

[32] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *International Conference on Software Engineering (ICSE)*, pages 386–396. IEEE, 2009.

[33] 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 *International Conference on Software Engineering (ICSE)*, pages 727–737, 2012.

[34] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 167–178. ACM, 2003.

[35] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *International Conference on Software Engineering (ICSE)*, pages 75–84. IEEE, 2007.

[36] R. Pandita, T. Xie, N. Tillmann, and J. de Halleux. Guided test generation for coverage criteria. In *International Conference on Software Maintenance (ICSM)*, pages 1–10, 2010.

[37] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *Symposium on Foundations of Software Engineering (FSE)*, pages 135–145. ACM, 2008.

[38] S. Park, S. Lu, and Y. Zhou. CTrigger: exposing atomicity violation bugs from their hiding places. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 25–36. ACM, 2009.

[39] M. Pradel and T. R. Gross. Fully automatic and precise detection of thread safety violations. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 521–530, 2012.

[40] M. Pradel, M. Huggler, and T. R. Gross. Performance regression testing of concurrent classes. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 13–25, 2014.

[41] W. Pugh and N. Ayewah. Unit testing concurrent software. In *Conference on Automated Software Engineering (ASE)*, pages 513–516. ACM, 2007.

[42] M. Samak and M. K. Ramanathan. Multithreaded test synthesis for deadlock detection. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 473–489, 2014.

[43] M. Samak and M. K. Ramanathan. Synthesizing tests for detecting atomicity violations. In *ESEC/FSE*, pages 131–142, 2015.

[44] M. Samak, M. K. Ramanathan, and S. Jagannathan. Synthesizing racy tests. In *PLDI*, pages 175–185, 2015.

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

[46] J. Schimmel, K. Molitorisz, A. Jannesari, and W. F. Tichy. Automatic generation of parallel unit tests. In *Workshop on Automation of Software Test (AST)*, pages 40–46, 2013.

[47] K. Sen. Race directed random testing of concurrent programs. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 11–21. ACM, 2008.

[48] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 263–272. ACM, 2005.

[49] A. Solar-Lezama, C. G. Jones, and R. Bodík. Sketching concurrent data structures. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 136–148. ACM, 2008.

[50] F. Sorrentino, A. Farzan, and P. Madhusudan. PENELOPE: weaving threads to expose atomicity violations. In *Symposium on Foundations of Software Engineering (FSE)*, pages 37–46. ACM, 2010.

[51] S. Steenbuck and G. Fraser. Generating unit tests for concurrent classes. In *International Conference on Software Testing, Verification and Validation (ICST)*, 2013.

[52] S. Tasharofi, M. Pradel, Y. Lin, and R. Johnson. Bita: Coverage-guided, automatic testing of actor programs. In *Conference on Automated Software Engineering (ASE)*, 2013.

[53] R. N. Taylor, D. L. Levine, and C. D. Kelly. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering*, 18(3):206–215, 1992.

[54] V. Terragni, S. Cheung, and C. Zhang. RECONTEST: effective regression testing of concurrent programs. In *ICSE*, pages 246–256, 2015.

[55] V. Terragni and S.-C. Cheung. Coverage-driven test code generation for concurrent classes. In *ICSE*, 2016.

[56] M. T. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 327–338, 2010.

[57] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[58] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 97–107. ACM, 2004.

[59] C. von Praun and T. R. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *Conference on Programming Languages Design and Implementation*, pages 115–128. ACM, 2003.

[60] C. Wang, M. Said, and A. Gupta. Coverage guided systematic concurrency testing. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 221–230, 2011.

[61] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for java libraries. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 602–629. Springer, 2005.

[62] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 365–381. Springer, 2005.

[63] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad hoc synchronization considered harmful. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 163–176. USENIX, 2010.

[64] C.-S. D. Yang, A. L. Souter, and L. L. Pollock. All-du-path coverage for parallel programs. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 153–162, 1998.

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

[66] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. W. Reps. ConSeq: detecting concurrency bugs through sequential errors. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 251–264, 2011.

[67] W. Zhang, C. Sun, and S. Lu. ConMem: detecting severe concurrency bugs through an effect-oriented approach. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 179–192. ACM, 2010.