Generating Unit Tests for Concurrent Classes

Sebastian Steenbuck
Saarland University – Computer Science
Saarbrücken, Germany
steenbuck@st.cs.uni-saarland.de

Gordon Fraser
University of Sheffield
Sheffield, UK
gordon.fraser@sheffield.ac.uk

***Abstract*—As computers become more and more powerful, programs are increasingly split up into multiple threads to leverage the power of multi-core CPUs. However, writing correct multi-threaded code is a hard problem, as the programmer has to ensure that all access to shared data is coordinated. Existing automated testing tools for multi-threaded code mainly focus on *re-executing* existing test cases with different schedules. In this paper, we introduce a novel coverage criterion that enforces concurrent execution of combinations of shared memory access points with different schedules, and present an approach that automatically *generates* test cases for this coverage criterion. Our CONSUITE prototype demonstrates that this approach can reliably reproduce known concurrency errors, and evaluation on nine complex open source classes revealed three previously unknown data-races.**

***Keywords*-concurrency coverage; search based software engineering; unit testing**

I. INTRODUCTION

The increasing use of multi-core processors makes it ever more important to use multi-threaded programming to leverage the power of multiple cores. However, writing correct multi-threaded code is a hard problem, as the programmer has to ensure that all access to shared data is coordinated. The coordination is usually done with some sort of locking – which in turn might lead to deadlocks [20]. Software testing is an important countermeasure to identify such concurrency issues in programs.

Writing test cases for concurrency errors is problematic for three reasons: (1) The scheduler might not be controllable by the programmer; this is the case in Java. (2) The programmer might not fully understand the consequences of multiple threads accessing the same memory. (3) The programmer might not think of some particular schedule that would be required to lead to a concurrency issue; e.g., he might know about a data race but assumes that one thread always needs much longer than another thread and therefore does not enforce the order of the threads [16].

For example, Figure 1 shows the `entrySet` method of the `HashMultimap.AsMap` class, containing bug #339[1]. If two threads access the same hashmap and call `entrySet` at the same time, a race condition can lead to erroneous behavior: If `entrySet` is `null` initially, then the first thread assigns `null` to `result` and should then check if `entrySet` was `null`. However, if the scheduler switches the context before the check can happen, the second thread may assign a value to `entrySet`. The first thread will evaluate the condition to false and return its local `result` variable, which is null.

```
1 public Set entrySet() {
2  Set result = entrySet;
3  return (entrySet == null) ?
4              entrySet = new AsMapEntries()
                   :
5              result;
6 }
```

Figure 1. Concurrency bug #339 in the `HashMultimap.AsMap` class in the Guava library: `entrySet()` may return `null` if two threads call the method at the same time.

```
HashMultimap multi0 = new HashMultimap();
HashMultimap.AsMap map0 = m.asMap();
```

Thread 1:	*Thread 2:*
`map0.entrySet();`	`map0.entrySet();`

Figure 2. Concurrent unit test produced by CONSUITE revealing the bug in Figure 1: Two threads independently access a shared `HashMultimap.AsMap` instance, and are executed with all interleavings of the three accesses to the `entrySet` field in Figure 1.

Automated test generation is important to simplify software testing, and modern test generation techniques can efficiently generate test data that exercise program code thoroughly. However, structural test generation approaches often assume that there is no concurrency (e.g., [9]). On the other hand, tools to automate concurrent testing often assume that tests already exist such that different schedules can be explored (e.g., [7], [18]).

In this paper we present a technique that generates both, *concurrent test cases* and *execution schedules*. All the technique needs as input is the bytecode of a class under test (CUT). It generates test cases such that combinations of memory accesses are exercised by different threads, and then explores different schedules for these combinations. For example, Figure 1 contains three synchronization points (memory accesses to a shared variable, definition in Section II-C) when accessing the `entryMap` member variable at Lines 2, 3, and 4. Our approach would create test cases

[1] http://code.google.com/p/guava-libraries/issues/detail?id=339

to cover all *pairs* and *triples* of such synchronization points, and would then execute these with all possible interleavings. Figure 2 shows such an example test case, and this test case easily reveals the data race, leading to a `null` return value.

In detail, the contributions of this paper are as follows:

- **Concurrency coverage:** We formalize a coverage criterion based on observations on real concurrency bugs (Section II).
- **Test generation:** We present a search-based technique to derive concurrent test cases that exercise a shared object with respect to the concurrency coverage criterion (Section III).
- **Evaluation:** We demonstrate on a set of concurrency benchmarks that the approach can reliably reproduce known concurrency issues (Section IV).
- **Real bugs:** We show the results of an evaluation on a set of open source classes, revealing several previously unknown data races (Section IV).

II. Background

A. Evolutionary Testing of Classes

Search-based testing applies meta-heuristic search techniques to test data generation [17]. A popular algorithm also used in this paper is a ge¡netic algorithm (GA), in which a population of candidate solutions is evolved towards satisfying a chosen coverage criteria. A fitness function guides the search in choosing individuals for reproduction, gradually improving the fitness values with each generation until a solution is found.

In this paper we consider object oriented software, for which tests are essentially small programs exercising the classes under test. A common representation is a variable length instruction sequence [9], [26] (method sequences). Each entry in such a sequence represents one statement in the code, such as calls to constructors, methods, or assignments to variables, member fields, or arrays. Recently, search-based testing has also been extended to whole test suite generation [9], where test suites are evolved with similar operators towards satisfying entire coverage criteria.

B. Concurrent Testing

The majority of work on automated unit testing considers the case of single-threaded programs. In the context of multi-threaded programs, test automation usually focuses on the exploration of different schedules for existing program runs. For example, the DejaVu platform [4] allows the deterministic replay of a multi-threaded Java program. Other approaches assume existing tests such that different schedules for these tests can be tested. For example, Contest [7] allows the replay of program runs and the generation of new interleavings by adding random noise to the scheduler. Alternatively, stateless model checkers like CHESS [18] can systematically explore all thread interleavings for a given test case. Heuristics have been shown useful to speed up this exploration (e.g., [5]). Finding good schedules can also be interpreted as an optimization problem [8] solvable, for example, by a GA. LineUp [3] checks linearizability by comparing whether sequential behavior to interleaved behavior, but assumes a set of method calls given as input, i.e., it does not address the test generation problem.

There are only few approaches that try to generate not only schedules, but also test cases: Krena et al [13] proposed a search-based technique that optimizes schedules used by the Contest [7] tool. Ballerina [19] uses random test prefixes, and then explores two methods executed in parallel in two different threads. Sen and Agha [22] combined jCute [23] with race detection algorithms. The Ballerina tool [19] is the most similar to the approach described in this paper: It randomly invokes methods on a shared object under test in parallel. However, it cannot find concurrency issues that require execution of parallel sequences of code rather than individual method calls, and it cannot find data races based on more than two memory access points [16] (e.g., Figure 4).

Godefroid and Khurshid [11] used a GA in a model checking approach to find deadlocks; their fitness function aims to maximize the number of waiting threads. A similar approach is also followed by Alba et al [1]. These approaches search for individual program inputs rather than sequences of calls, and they are not driven by structural coverage criteria. In model checking partial order reduction [10] is used to reduce the number of schedules that need to be explored. In principle, partial order reduction could serve as an optimization of our test generation approach.

C. Concurrency Coverage

Systematic test generation is often guided by a coverage criterion that describes which aspects of a program a test suite needs to cover; e.g., statements or branches. To explore different interesting concurrency scenarios, dedicated coverage criteria for concurrency have been proposed. Synchronization coverage [2] was designed as an easy to understand concurrent coverage criterion such that testers would be able to apply it when manually writing unit tests. The coverage criterion requires that for each synchronization statement in the program (e.g., synchronize blocks, wait(), notify(), notifyAll() of an object instance and others) there is a test such that a thread is stopped from progressing at this statement.

To also consider shared data accesses that are not guarded by synchronization statements, Lu et al [15] defined a hierarchy of more rigorous criteria, including *all-interleaving coverage*, which requires that all feasible interleavings of shared access from all threads are covered; *thread pair interleaving* requires for a pair of threads that all feasible interleavings are covered; *single variable interleaving* requires that for one variable all accesses from any pair of threads are covered; finally, *partial interleaving* requires either coverage of definition-use pairs or consecutive execution of pairs of access points. In general, many of the existing criteria are either too weak or too complex to be of practical value in a testing context [21], [24].

Lu et al. [16] analyzed 105 concurrency bugs in open source applications, and learned that:

- 96% of the examined concurrency bugs are guaranteed to manifest if a certain partial order between two threads is enforced,
- 92% of the examined concurrency bugs are guaranteed to manifest if a certain partial order among no more than four memory accesses is enforced; 75% are guaranteed to manifest themselves with up to three memory accesses; all deadlocks were guaranteed to manifest themselves with up to three memory accesses,
- 97% of the examined deadlock bugs involved at most two variables, and 66% of the examined non-deadlock bugs involved only one variable.

This suggests that it is feasible to find most concurrency issues by exhaustively covering combinations of low numbers of memory access points, threads, and variables.

In the this paper, we use term *synchronization point* (SP) synonymously for a memory access to a shared variable. For example, in the case of Java bytecode this amounts to the GETFIELD, GETSTATIC as well as PUTFIELD and PUTSTATIC instructions. A *schedule* is a list $\langle(t_1, s_1), \ldots, (t_n, s_n)\rangle$ where $t_i \in [1, \ldots, m], s_i \in [1, \ldots, n]$. For example, (t_1, s_1) with $t_1 = 1$ and $s_1 = 3$ means thread 1 executes the instruction associated with the synchronization point 3. The example in Figure 1 has three synchronization points: $s_1 = 1$ and $s_2 = 2$ in line 2 and 3 for the GETFIELD bytecode instructions used to load the `entrySet` and $s_3 = 3$ in line 4 for the underlying PUTFIELD instruction. With two threads $(t_1 = 1, t_2 = 2)$ the schedule $\langle(t_1, s_1), (t_2, s_3)\rangle$ is a failure enforcing schedule, while $\langle(t_1, s_3), (t_2, s_1)\rangle$ is not.

We define *concurrency coverage* as a parameterized criterion that requires covering all possible schedules for sets of threads, variables, and synchronization points:

Definition 1 (Concurrency coverage): All feasible combinations of n synchronization points for each group of v variables, being accessed by m threads.

In a program which only one thread can access at a time, no such schedule can be executed. Therefore we also count schedules that cannot be executed due to deliberate synchronization as covered. Concurrency coverage fits between thread pair interleaving and partial interleaving coverage [15]. The number of schedules in the criteria can be larger or smaller as in single variable interleaving, depending on the tested code. As Lu et al. [16] determined that more than 70% of all concurrency bugs manifest if a certain partial order between three main memory accesses to each combination of two variables is enforced, in this paper we consider concurrency coverage for the case of $n = 3$, $m = 2$, and $v = 2$.

Algorithm 1 Test suite generation conceptional view.

Require: Class C
Require: Number of synchronization points n
Require: Number of threads m
Require: Number of variables v
Ensure: Test Suite T

procedure GENERATESUITE((C, n, m, v))
    $p \leftarrow$GENERATEPREFIXSEQUENCE(C)
    $N \leftarrow$GETSYNCHRONIZATIONPOINTS(C)
    $S \leftarrow$ GENERATESCHEDULES(n, m, v, N)
    $P \leftarrow$ GENERATEPARTIALGOALS(S, m)
    $S \leftarrow$ remove statically infeasible schedules from S
    $A \leftarrow$ generate method sequences for P
    $T \leftarrow \{\}$
    for $s \in S$ **do**
        for *sequences* $\in$ GETMATCHINGS(s, A) **do**
            if seq. execution of seq reaches s **then**
                if par. execution of seq reaches s **then**
                    $T \leftarrow T \cup \{(p, sequences, s)\}$
    return T;

III. GENERATING CONCURRENCY TESTS

Unlike synchronization coverage [25], which was specifically designed in order to cater for the needs of developers when manually writing unit tests, concurrency coverage is likely to produce far more coverage goals that need to be exercised by tests. Thus, the aim of our test generation approach is to automatically produce test sets that achieve high concurrency coverage.

Algorithm 1 depicts the approach at a high level. The output of the algorithm is a set of test cases, where each test is a triple consisting of a prefix sequence p that creates a shared object, a set of m method sequences performing operations on the shared object, and a schedule s which determines how the m threads are interleaved. This set of tests covers as many as possible of all n combinations of synchronization points of v variables with m threads.

The algorithm first generates a simple prefix sequence that creates an instance of the class under test C – in most cases, this is simply a constructor call and some setup of required complex parameters. Then, the first step of the analysis consists of determining all test goals, i.e., all schedules that the coverage criterion requires for the given parameters m, n, v on the class C. Based on the set of schedules S, the algorithm then determines a set of individual method sequences that need to be generated. For this, the schedules are summarized in a tree-like data structure; this is described in detail in Section III-A.

The result of this analysis is a set of sub-goals P, each of which is a single method sequence that covers different synchronization points of class C. As described in Section III-B, we use a GA to derive these sequences.

Finally, given the set of method sequences A we can

Algorithm 2 Schedule generation.

Require: Number of synch. points n in a schedule
Require: Number of threads m
Require: Number of variables v
Require: Set of synchronization points N
Ensure: Schedules S

procedure GENERATESCHEDULES(n, m, v, N)
  $G \leftarrow [1, \ldots, m] \times [1, \ldots, |N|]$
  $S \leftarrow G$
  for 2 to n **do**
    $S \leftarrow S \cup S \times G$
  clean(S) //remove schedules with only one thread
  return S;

iterate over all target schedules S and try to assemble a concurrent test consisting of m individual method sequences in parallel. The procedure GETMATCHINGS in Algorithm 1 selects all combination of the method sequences in A which reach the points needed by the threads in s. Therefore $sequences$ is a list of method sequences. Whose first elements reaches the points thread t_1 should visit. This thread consists of first executing the individual threads in sequence to verify if the synchronization points are still covered that way, and then executing them in the target schedule s. This step is described in Section III-C.

A. Determining Coverage Goals

The first step of the test generation approach is to determine the set of synchronization points N for a given class C. A memory access happens every time a field of C is read from or written to. In our scenario (Java bytecode) these are the read (GETFIELD, GETSTATIC) and write (PUTFIELD, PUTSTATIC) bytecode instructions.

Based on these synchronization points, the target schedules can be determined. For m threads, $|N|$ synchronization points and schedule length n, in the worst case (all $|N|$ synchronization points access the same variable) there are $(|N| \cdot m)^n$ schedules. In practice, the number of schedules depends on how many memory access points each variable has. Algorithm 2 illustrates schedules generation.

The number of schedules can become very large, but not all schedules are feasible: Some infeasible schedules can easily be detected using static analysis. E.g., consider the schedule $\langle(t_1, s_1), (t_1, s_2), (t_2, s_3)\rangle$: If s_1 and s_2 are in the same method, and s_2 dominates s_1, then there exists no test that can satisfy this schedule, under the assumption that both s_1 and s_2 have to be in the same method call. We identify and eliminate such infeasible schedules by checking for every synchronization point if there is a path in the control flow graph from this point to the other points that the schedule requires in the same thread and method.

To satisfy the individual schedules, we need to create method sequences such that we can assemble combinations of m different sequences in a way that all synchronization points described in the schedule can be matched by their corresponding threads. We first determine the combinations of synchronization points that the individual method sequences need to cover. The upper bound on schedules is $(|M| \cdot m)^n$, but as many of the schedules are very similar the number of actual sequences required is much lower. For example, consider the schedules $\langle(t_1, s_1), (t_2, s_2), (t_1, s_3)\rangle$ and $\langle(t_1, s_1), (t_1, s_3), (t_2, s_2)\rangle$: They differ only in terms of the scheduling, but not in terms of the memory access points.

Algorithm 3 Test goal generation.

Let $f(t_i, \langle(t_1, s_1), \ldots, (t_n, s_n)\rangle) \rightarrow [(s_{x_1}), \ldots, (s_{x_y})]$ be a function, which selects all the y synchronization points belonging to thread t_i from a schedule and returns them as an ordered list.

Require: Set of schedules S
Require: Number of threads t
Ensure: Set of partial goals G

procedure GENERATEPARTIALGOALS(S, t)
  $G \leftarrow$ empty tree
  for $\langle(t_1, s_1), \ldots, (t_i, s_i)\rangle \in S$ **do**
    for $t \in \{1, \ldots, t\}$ **do**
      add($G, f(t, \langle(t_1, s_1), \ldots, (t_i, s_j)\rangle)$)
  return Paths(G);

To determine the actual set of method sequences that we require to cover all goals, we determine a set of *partial* goals, where each partial goal consists of a sequence of synchronization points that need to be covered by an individual method sequence. As illustrated in Algorithm 3 illustrates, we generate a tree G of synchronization points, where each path from the root to a leaf equals the projection of a schedule on one thread. The set of partial goals is determined as the set of paths in the tree G.

B. Satisfying Partial Goals

Given a set of partial goals, the next step in the algorithm is to derive sequences of method calls to satisfy them. A partial goal is a sequence of synchronization points, and a sequence of method calls satisfies a partial goal if it executes these synchronization points in order.

To produce a sequence of method calls that reaches one particular synchronization point, we use a GA. The fitness function to drive the GA towards covering one synchronization point is based on the well-established approach-level and branch distance metrics (see e.g., [17]). The approach level represents the number of unsatisfied control dependencies towards reaching the target statement, whereas the branch distance estimates how close the point of diversion was towards evaluation as required. There are standard rules to derive the branch distance [17]. The overall fitness function for one synchronization point s_i thus is:

$$d(t, s_i) = \text{approach level} + \alpha(\text{branch distance})$$

Here, α is a normalization function in the range $[0, 1]$, and the optimization needs to minimize this fitness value. A partial goal is a sequence of synchronization points $g = \langle s_1, \ldots s_n \rangle$. The fitness function for a partial goal is thus a combination of the individual fitness values for the synchronization points:

$$\text{fitness}(t, \langle s_1, \ldots, s_n \rangle) = \sum_{i=1}^{n} \begin{cases} 1 & \text{if } i > 1 \text{ and } \\ & \exists x < i : s_x \text{was} \\ & \text{not covered,} \\ \alpha(d(t, s_i)) & \text{otherwise} \end{cases}$$

This fitness function estimates the distance towards the next synchronization point not yet covered. For example, if the first synchronization point s_1 is not covered, then the fitness function consists of the distance $d(t, s_1)$ towards reaching s_1, and for each remaining synchronization point we add 1. Once s_1 is covered its fitness is 0, and the overall fitness consists of $d(t, s_2)$ plus 1 for every synchronization point thereafter; and so on.

Because the number of partial goals can be large and many of them can be similar, we apply a further optimization and do not search for individual partial goals, but *sets* of partial goals. This is based on the idea of whole test suite generation [9], and means that the individuals of the GA are not sequences of method calls, but sets of sequences of method calls. For operations on sets of sequences of method calls we refer to [9]. If the number of partial goals is too large, it is possible to consider subsets of a chosen size X of all partial goals, which are randomly selected out of the set of partial goals not already covered. Once the GA has covered a certain amount of partial goals or the fitness has not improved after a predetermined number of generations the GA would restart with a new set of partial goals, and the best individuals of the GA are retained to seed further iterations of the search. The fitness function for a set of partial goals $P = \{p_1, \ldots, p_n\}$ on a set of test cases $T = \{t_1, \ldots, t_m\}$ is calculated as follows:

$$\text{fitness}(T, P) = \sum_{i=1}^{n} (min_{j=1}^{m}(\text{fitness}(t_j, p_i)))$$

C. Assembling Concurrent Tests

After the second step, we now have at least one sequential test for each partial goal, except for the cases where the search failed; due to coincidental coverage the number of tests for each partial goal in practice is usually higher than one. In the last step of the algorithm we combine an m-tuple of sequences for each schedule $s \in S$ to generate the concurrent tests.

This assembly step works as follows: First, we select m sequences of method calls that together cover all the synchronization points required by the chosen schedule s. These tests are executed per thread on a shared object of the CUT, which is generated in terms of the prefix sequence p. This sequential execution determines whether the synchronization points in s that were covered by the sequences in isolation are still covered when the tests are combined. For example, a call to `empty()` on a container class might result in a different result if a previous test has already put some object into the container. So for the schedule $\langle(t_1, s_1), (t_2, s_2), (t1, s_3)\rangle$, we would try to find two tests: One which covers $\langle s_1, s_3 \rangle$, and one that covers $\langle s_2 \rangle$, if both are executed sequentially. If the synchronization points are no longer covered by the schedule, we drop this combination and attempt a different one. As the behaviour changes if the tests are executed with some tighter interleaving, we generate multiple candidate tests for each schedule.

IV. EVALUATION

To analyze the effectiveness of our approach we have implemented our CONSUITE prototype as an extension to the EVOSUITE [9] unit test generation tool and performed a set of experiments. The aim of these experiments is to demonstrate that it is able to find concurrency issues, and to gain insights on its efficiency and scalability.

A. Experimental Setup

CONSUITE is a search-based Java test generation tool that works on the bytecode level, i.e., no source code is required for its operation. Instrumentation for concurrency coverage is also performed at the bytecode level. In particular, data accesses are performed at all GETFIELD/GETSTATIC and PUTFIELD/PUTSTATIC bytecode instructions. Consequently, the bytecode instrumentation consists of additional tracing calls inserted before each of these instructions; this instrumentation serves for both, to track coverage and to influence the scheduling.

The GA used to generate method sequences for partial goals was configured to run for a total of 20 minutes for all partial goals per class. The GA created sets of method sequences targeting all partial goals at the same time; the size of the sets during the search is variable. All other parameters of the GA were set to their defaults according to the underlying EVOSUITE. To accommodate for the randomness of the approach, each experiment was repeated five times, and all values listed in this paper are averaged over all runs.

B. Concurrency Bug Benchmarks

Our first set of experiments is performed on known concurrency bug examples taken from the Software Infrastructure Repository (SIR) [6]. Table I lists the details; each example is minimized to reveal one particular concurrency fault. The aim of the first experiment is to determine if our approach is able to generate tests that would reveal these bugs. In total SIR contains 25 Java concurrency bug examples, out of which we used the 12 listed in Table I; the other examples did not qualify for our experiments because a) they required access to files, which is problematic for

Table II
OPEN SOURCE CONCURRENT CLASSES USED FOR EVALUATION

Example		Project	LOC	SP	Schedules	P. goal pairs	Partial goals
CH	`ConcurrentHashMap`	java.util	1,323	162	290,607	96,869	4,174
FT	`FastTreeMap`	Commons Collections	827	148	1,418,919	472,973	6,066
FH	`FastHashMap`	Commons Collections	718	106	469,095	156,365	2,898
SB	`StaticBin1D`	Colt	301	100	28,176	9,392	704
AM	`AbstractMultiMap$AsMap`	Guava	150	47	2,859	953	117
MA	`MemoryAwareConcurrentReadMap`	Groovy	369	61	316,332	105,444	3,525
FA	`FileAppender`	Log4j	463	33	13,766	4,589	543
CR	`ConcurrentReaderHashMap`	Groovy	1,231	136	820,992	273,664	9,050
CW	`CopyOnWriteArrayList`	EDU.oswego.cs	1,199	114	448,902	149,634	2,825

Table I
CONCURRENCY BUGS FROM THE SIR

Example	LOC	SP	Bug	Revealing Tests
Account	116	4	Data Race	7
Airline	52	12	Data Race	1
Allocation Vector	151	15	Data Race	8
Sleeping Barber	154	27	Deadlock	10
Deadlock	41	14	Deadlock	14
Dining	101	12	Deadlock	4
LinkedList	130	37	Data Race	22
NestedMonitor	139	2	Deadlock	1
ReadersWriters	170	40	Data Race	4
Reorder	42	10	Data Race	6
TwoStage	62	11	Data Race	2
WrongLock	40	6	Data Race	4

automated test generation in general, or b) they are examples of a producer-consumer pattern, which is not the scenario our approach is targeting (i.e., it would require to generate additional threads during test generation, or it would require the test generator to handle additional threads generated during test execution).

The last columns of Table I summarizes the results of the experiments on the SIR examples. Each of the SIR examples contains an oracle that allows us to detect if the bugs the example represents was triggered by a test. As the examples are minimized with respect to these bugs, they are not representative with respect to the number of tests or the effort that goes into generating them, therefore we only focus on whether our approach leads to tests that reveal the bugs. As can be seen in Table I, for each of the examples there is at least one test case that triggers the bug; for the LinkedList example there are 22 test cases.

In our experiments, our CONSUITE *prototype revealed **all** benchmark concurrency bugs.*

C. Open Source Concurrent Classes

As the SIR examples are minimized with respect to the concurrency bugs they contain, they are not representative in terms of the scalability and effectiveness of the approach. Therefore we performed a second set of experiments on a selected number of concurrent open source classes, with the aim to gain insight on the level of coverage that can be achieved with our approach, the number of potential problems detected, and the performance. Details of these classes are given in Table II. The number of test targets (schedules) prescribed by the concurrency coverage criterion can become very large, and ranges from 2,859 in the `AbstractMultiMap$AsMap` subclass to 1,418,919 for the `FastTreeMap`. This describes the total number of schedules to be executed; the number of unique synchronization point combinations is significantly lower. The column labelled "P. goal pairs" in Table II refers to unique combinations of partial goal pairs (pairs as we used $m = 2$ for the evaluation). This number is lower than the number of schedules, as several schedules of a combination of n memory access points in m threads can be done with the same m-tuple of individual method sequences. Finally, the partial goals are those that the GA has to satisfy.

Table III
RESULTING NUMBER OF TEST CASES

Example	MS	MS Pairs	Executed	Schedules
CH	251	589	446,535	150,670
FT	241	5,273	5,902,732	986,128
FH	159	799	1,093,958	345,965
SB	20	19	97,881	28,176
AM	13	42	2,587	2,040
MA	–	–	–	–
FA	91	542	28,798	7,239
CR	178	1,517	1,876,288	176,054
CW	155	716	1,915,757	448,838

These numbers are not comparable to the small numbers of coverage goals one would get for traditional structural criteria like branch coverage. However, we argue that this is a feasible number, as the approach is targeting unit tests of concurrent classes, and unit tests are generally supposed to run fairly quick. Furthermore, the resulting test set will not be reported to the user in its entirety, unlike, e.g., a branch coverage test suite, where the objective is often to provide a complete test set to the developer, who then is supposed to add test oracles. In the concurrency coverage scenario, however, the user would only see those tests that lead to actual concurrency issues such as deadlocks or data races, and these can be further post-processed such that only unique concurrency issues are reported to the users. As will be seen later in Table V this number is much lower.

Table III summarizes the results in terms of the number

Table IV
COVERAGE RESULTS ON OPEN SOURCE CLASSES

Example	P. goal pairs	Partial goals	Schedules
CH	59%	53%	52%
FT	78%	73%	70%
FH	85%	77%	74%
SB	100%	100%	100%
AM	83%	73%	72%
MA	–	–	–
FA	64%	54%	53%
CR	38%	22%	22%
CW	100%	100%	100%

of produced tests. It can be seen that the actual number of method sequences (MS) generated is much lower than the number of partial goals. This is mainly because a single method sequence usually covers more than one partial goal. The number of method sequence pairs is also significantly lower than the number of goals. The column labelled "Executed" denotes the actual number of test cases executed, which is significantly larger than the number of actually covered schedules ("Schedules"). This is because we usually need to execute several different combinations of method sequences until the interleaved execution properly reaches all synchronization points as required by the target schedule. Table III and the following tables list no results for the `MemoryAwareConcurrentReadMap` (MA) class, as CONSUITE detected a deadlock in this class early on. This deadlock appeared for so many different schedules that we had to cancel test generation, as the deadlock detection slows down test execution such that a full exploration would have taken too long. In practice, one would fix the deadlock and then restart CONSUITE on the fixed class.

Table IV lists how many of the schedules and partial goals have been covered by the tests listed in Table III. We can see a small decrease from method sequences to combinations, which arises when the combination leads to individual synchronization points no longer being reached in sequential execution. The decrease from combinations to actually executed schedules arises when an interleaved execution leads to different synchronization points than the sequential execution. In all cases, this loss is relatively small. The largest potential for improvement of coverage lies in individual method sequences – this could quite easily be increased simply by giving the GA more time to perform the search; in our experiments it was set to 20 minutes, which in the larger classes simply was not enough to achieve higher coverage.

> *In our experiments, our* CONSUITE *prototype achieved 68% concurrency coverage on average.*

D. Data Race Detection

There are several ways to select the interesting test cases out of the large amount produced by our approach. If there are automated oracles in terms of assertions or contracts, then violations of these would be candidates. This was done for the SIR examples above; in the case of the open source classes selected there are no such oracles except for deadlocks or generic object contracts (e.g., there should be no undeclared exceptions or program crashes). An alternative way to report concurrency issues is to monitor data races. We use an approach based on the happens-before relation [14].

To find data races, we instrumented all inter-thread actions[2] in the tested code. As we are only interested in data races in the tested code we did not instrument any other classes (e.g., in java.utils), therefore the overhead of the data race detection was negligible compared to the overhead of the instrumentation already in place to monitor and force context switches. When the control flow leaves instrumented code, we assumed synchronization actions on all objects.

```
1 public Object put(Object key,Object value){
2  if (fast) {
3   synchronized (this) {
4    TreeMap temp = (TreeMap) map.clone();
5    Object result = temp.put(key, value);
6    map = temp;
7    return (result);
8   }
9  } else {
10  synchronized (map) {
11   return (map.put(key, value));
12  }
13 }
14 }
```

Figure 3. Concurrency bug detected by CONSUITE in the `FastTreeMap` class in Apache Commons Collection: `put(Object, Object)` may lose an update if two threads call the method at the same time.

CONSUITE was able to find data races in all but three classes: The `ConcurrentHashMap` from the `java.util.concurrent` package is very complex; much of this complexity originates from rigorous treatment of potential and past concurrency issues. All data races we found in the `ConcurrentHashMap` are, to the best of our knowledge, benign. They are optimizations, where a result is produced without a lock and is checked afterwards with the read of a volatile version number. CONSUITE was also not able to find any data races on the `StaticBin1D` class from the Colt library and the `CopyOnWriteArrayList`. Manual investigation of these classes confirms that synchronization seems to be correctly implemented.

CONSUITE was able to find real unreported bugs in the `FastTreeMap` and `FastHashMap` classes of the current stable version of Apache Commons Collections. For example, Figure 3 shows the `put` method of `FastTreeMap`, containing a data race detected by CONSUITE. If two threads (as shown in CONSUITE's test case in Figure 4) access the same `FastTreeMap` and call `put` at the same time, a race condition can lead to erroneous behavior: If the first thread starts execution and runs until Line 5 in `put`, and then the

[2] http://java.sun.com/docs/books/jls/third_edition/html/memory.html

```
FastTreeMap map0 = new FastTreeMap();
```

Thread 1:	Thread 2:
`map0.setFast(true);`	`map0.setFast(false);`
`map0.put(1,1);`	`map0.put(2,2);`

Figure 4. Concurrent unit test produced by CONSUITE, revealing the bug in Figure 3: Two threads independently access a shared `FastTreeMap` instance, and are executed with all interleavings needed for concurrency coverage of the three synchronization points in Figure 3. The method `setFast` is a wrapper around `fast` without any synchronization.

second thread executes all its commands it puts the value into the initial backing TreeMap `map`. When the first thread starts execution again it will override the reference `map` with `temp`. As `temp` does not contain the value that the second thread added, that value is lost.

The bug shown in Figure 3 is one manifestation of the failed synchronization attempt. Another interesting version of that bug, which manifests as a data race, involves the Java memory model[3]. The method `get()` is not synchronized, if fast is true. If two threads access one instance of the `FastTreeMap` class, one calls `setFast(true)` once and after that always calls `get('A')` until a value for 'A' is found, while the other calls `put('A', 'B')` once, the compiler (which in the Java memory model also includes the memory hierarchy) is free to never return a value different from `NULL` to the first thread. Because of errors like this, it is not enough to check for linearizability [19]. The compiler has some degrees of freedom and may choose to only display this error on a production system, e.g., if the production system has more processor caches. The `FastTreeMap` and `FastHashMap` classes both have further real concurrency issues; however, we did not investigate all reported data races in detail, as the first issue we considered revealed a design error that leads to the same type of data race at several locations in the classes.

The `AbstractMultiMap$AsMap` from the Guava project contains a known bug, and we deliberately selected a version of the class before this data races was fixed. CONSUITE was able to find the data race without problems (see Figure 1 and 2). The `FileAppender` of the Log4j 1.0.2 library contains a known data race that leads to a NullPointerException, and is thus easy to discover, even without data race detection.

Finally, in the `MemoryAwareConcurrentReadMap` class from the Groovy project CONSUITE found a deadlock. The `ConcurrentReaderHashMap` from the same project contains a data race that allows the number of elements contained in the map to differ arbitrarily from the number that is reported. Further investigation of the class revealed that it is no longer used, so we did not report this particular bug.

> *In our experiments, our* CONSUITE *prototype revealed three new and three known real concurrency issues.*

[3] http://java.sun.com/docs/books/jls/third_edition/html/memory.html

Table V
DATA RACES FOUND BY CONSUITE

Example	Data Races	Unique Data Races
CH	127,641	42
FT	8,005,102	341
FH	2,236,636	218
SB	0	0
AM	2,195	5
MA	–	–
FA	65,708	28
CR	461,002	23
CW	0	0

Table V lists the numbers of data races detected in detail. The number of data races given refers to the number of tests that revealed a data race; these data races were then minimized with respect to the synchronization point, such that at the end only the small number of unique data races was left to investigate.

E. Concurrency Coverage vs. Random Tests

As a sanity check with respect to the optimization of the concurrency coverage criterion, we ran another set of experiments where we configured CONSUITE to produce random tests. From these we generated combined tests as described in Section III-C, so this experiment does not represent a true random testing approach where schedules would be randomized as well, but should give an upper bound on what is possible with random testing. We used the number of executed tests by CONSUITE as a stopping criterion for the random approach, and the length of the tests was on average the same.

For reasons of space we cannot provide the full data of these experiments, but only summarize our findings. As expected, random tests achieve significantly lower concurrency coverage: On average, random testing was able to cover 50.5% of the partial goal pairs (vs. 75.9% by CONSUITE), 41.0% of the partial goals (vs. 69.0%), and thus in total only 26.6% of all schedules (vs. 67.9%). The number of detected data races is also lower than in the test sets produced with respect to concurrency coverage; CONSUITE detected 657 unique data races in total, whereas the random tests revealed 574. Consequently, we can conclude that the results achieved with CONSUITE are not simply due to the large number of tests, but also due to the underlying coverage criterion.

> *In our experiments, concurrency coverage tests detected 14.5% more data races than random tests.*

In general, random tests are good at detecting shallow bugs; the example in Figure 3 was not found by the random tests. The data race results consist of two groups: For the `ConcurrentHashMap`, `AbstractMultiMap$AsMap`, and `ConcurrentReaderHashMap` examples random tests find nearly as many data races as evolved tests. These classes are either very small (`AbstractMultiMap$AsMap`) or the data races are

Table VI
Synchronization coverage

Example	Random Tests	Conc. Coverage Tests
CH	92%	92%
FT	95%	95%
FH	94%	94%
SB	64%	67%
AM	100%	100%
MA	–	–
FA	100%	100%
CR	91%	91%
CW	94%	100%

```
public void trimToSize() {
  synchronized(this){}
}
```

Figure 5. Example of a default method implementation in `StaticBin1D`.

intentional to improve performance. In this case the data race is often an optimistic execute of some logic and the complex case is only triggered if this fails; this is the case for all the `ConcurrentHashMap` data races. The one unintentional data race in the `ConcurrentReaderHashMap` class is caused by the `clear` function which is easy to cover, as it only has one branch. The other group consists of classes in which the data races are in more complex methods, e.g., `FastTreeMap`, `FastHashMap`, and `FileAppender`. In these cases ConSuite was able to find data races which the random approach could not.

F. Synchronization Coverage

To set our concurrency coverage criterion into context, we further measured the synchronization coverage of the resulting test sets. As shown in Table VI, synchronization coverage was, except for the `StaticBin1D`, upward of 90%. These results are the same for the randomly generated tests and the evolved tests. Synchronization coverage was designed to be reachable 100% of the time [25], therefore it is not unexpected that we can reach this coverage. For the `StaticBin1D` class the synchronization coverage is only 67% as some methods contain only default implementations as shown in Figure 5. There are no memory access points in the synchronized block and therefore ConSuite generates no schedules which wait inside the synchronized block. In general, the missing 5% to 9% coverage are an artifact of our implementation (e.g., the `hashCode` method is excluded by the underlying EvoSuite).

> *In our experiments,* ConSuite *achieved more than 95% synchronization coverage on average.*

G. Performance

To shed some light on the scalability of the approach, Table VII lists the time that went into test generation. The first step of generation of sequences of method calls was always fixed to 20 minutes and is therefore not listed in the table. The column labelled "Sequential" describes the time spent on the sequential execution of pairs of tests (see Section III-C), and "Combined" denotes the time spent on executing tests with different schedules. As also suggested by the large number of test executions shown in Table III, there is room for optimization in the latter part.

Table VII
Test generation time on open source concurrent classes (in seconds); the GA used to satisfy partial goals was set to 20 minutes in all examples.

Example	Sequential	Combined	Total
CH	23	591	614
FT	1,282	9,427	10,709
FH	60	1,740	1,800
SB	<1	420	420
AM	<1	3	3
MA	<1	–	–
FA	1	44	45
CR	233	5,540	5,773

> *In our experiments, generating and executing a concurrency coverage test suite took from 20 minutes up to 3 hours.*

H. Threats to Validity

Our experiments are subject to the following threats to *construct validity*: We measured the performance in terms of the achieved coverage and the number of data races found. In practice, there might be other factors that need to be considered, such as the readability and understandability of the resulting tests in order to explain the detected concurrency issues, and the high costs of the approach might outweigh the benefits.

Threats to *internal validity* might come from how the empirical study was carried out. To reduce the probability of having faults in our ConSuite prototype, it has been carefully tested. Threats to *external validity* regard the generalization to other types of software. Our set of evaluation classes is small, and has a strong emphasis on container classes. However, the application area of the approach is unit testing of classes under external concurrency, and we see no reason why the approach should not generalize to other classes in this domain.

V. Conclusions

In this paper, we have introduced concurrency coverage as a result of the insights gained in observations of real concurrency bugs [16]. The ConSuite tool implements a technique to automatically generate concurrent tests that satisfy this criterion. Although the number of tests required by the criterion can be very large, only few suspicious test cases need to be reported to the user. In our evaluation we revealed several previously unknown concurrency bugs in open source classes, demonstrating the effectiveness of the approach.

The CONSUITE prototype has potential for several optimizations: While generating individual method sequences for the partial goals we keep track of which other goals are covered accidentally by the current result. When running individual schedules, however, we do not perform this optimization because the overhead of the analysis would be too large on our current implementation. This results in the large numbers in Table III, and has significant potential for improvement, as a run likely satisfies more then one schedule. Furthermore, there may be cases where the GA is not able to derive a method sequence for a combination of synchronization points, yet during execution of a different test this particular schedule would be covered; currently, CONSUITE would miss such cases. The achieved levels of concurrency coverage showed that there is also room for improvement on the test generation part.

There are also ample opportunities for future work. The CONSUITE tool is not yet optimized towards producing readable test cases; how to best represent multi-threaded unit tests is a problem that is still actively researched (e.g., [12]). A related question is how to select representative regression test sets from the large concurrency coverage test sets.

Finally, in this paper we considered the case of external concurrency when testing individual classes. The concurrency coverage criterion presented is not limited to this scenario, but in principle can also be applied to non-unit testing scenarios and internal concurrency.

Acknowledgments. Thanks to Clemens Hammacher for comments on an earlier version of this paper, and Kiran Lakhotia for discussions on synchronization coverage. This project has been funded by a Google Focused Research Award on "Test Amplification".

REFERENCES

[1] E. Alba, F. Chicano, M. Ferreira, and J. Gomez-Pulido, "Finding deadlocks in large concurrent java programs using genetic algorithms," in *Proc. GECCO'08*. ACM, 2008, pp. 1735–1742.

[2] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur, "Applications of synchronization coverage," in *Proc. PPoPP*. ACM, 2005, p. 212.

[3] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan, "Line-up: a complete and automatic linearizability checker," *SIGPLAN Not.*, vol. 45, no. 6, pp. 330–340, Jun. 2010.

[4] J. Choi, B. Alpern, T. Ngo, M. Sridharan, and J. Vlissides, "A perturbation-free replay platform for cross-optimized multithreaded applications," in *Proc. IPDPS*. IEEE, 2001, pp. 10–pp.

[5] K. E. Coons, S. Burckhardt, and M. Musuvathi, "Gambit: effective unit testing for concurrency libraries," in *Proc. PPoPP'10*. ACM, 2010, pp. 15–24.

[6] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, pp. 405–435, October 2005.

[7] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur, "Multithreaded java program test generation," *IBM Systems Journal*, vol. 41, no. 1, pp. 111–125, 2002.

[8] Y. Eytani, "Concurrent java test generation as a search problem," *Electron. Notes Theor. Comput. Sci.*, vol. 144, pp. 57–72, May 2006.

[9] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions in Software Engineering*, 2013.

[10] P. Godefroid, *Partial-order methods for the verification of concurrent systems—an approach to the state-explosion problem*, ser. Lecture Notes in Computer Science. Springer, 1996, vol. 1032.

[11] P. Godefroid and S. Khurshid, "Exploring very large state spaces using genetic algorithms," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 6, pp. 117–127, 2004.

[12] V. Jagannath, M. Gligoric, D. Jin, Q. Luo, G. Rosu, and D. Marinov, "Improved multithreaded unit testing," in *Proc. ESEC/FSE '11*. ACM, 2011, pp. 223–233.

[13] B. Křena, Z. Letko, T. Vojnar, and S. Ur, "A platform for search-based testing of concurrent software," in *Proc. PADTAD '10*. ACM, 2010, pp. 48–58.

[14] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[15] S. Lu, W. Jiang, and Y. Zhou, "A study of interleaving coverage criteria," in *Proc. ESEC/FSE*. ACM, 2007, pp. 533–536.

[16] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *ACM SIGPLAN Notices*, vol. 43, no. 3. ACM, 2008, pp. 329–339.

[17] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[18] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs," in *Proc. OSDI'08*. USENIX Association, 2008, pp. 267–280.

[19] 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 *Proc. ICSE*, 2012.

[20] J. Ousterhout, "Why threads are a bad idea (for most purposes)," in *Presentation given at the 1996 Usenix Annual Technical Conference*, 1996.

[21] G. Ramalingam, "Context-sensitive synchronization-sensitive analysis is undecidable," *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 2, pp. 416–430, 2000.

[22] K. Sen and G. Agha, "A race-detection and flipping algorithm for automated testing of multi-threaded programs," in *Proc. HVC'06*. Springer-Verlag, 2007, pp. 166–182.

[23] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *Proc. ESEC/FSE-13*. ACM, 2005, pp. 263–272.

[24] E. Sherman, M. B. Dwyer, and S. Elbaum, "Saturation-based testing of concurrent programs," in *Proc. ESEC/FSE'09*. ACM, 2009, pp. 53–62.

[25] R. N. Taylor, D. L. Levine, and C. D. Kelly, "Structural testing of concurrent programs," *IEEE Trans. Softw. Eng.*, vol. 18, no. 3, pp. 206–215, Mar. 1992.

[26] P. Tonella, "Evolutionary testing of classes," in *Proc. ISSTA*, 2004, pp. 119–128.