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-threading 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. 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 might assign a value to entrySet. The first thread will evaluate the condition to false and return its local result variable, which is null.

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

```java
public Set entrySet() {
    Set result = entrySet;
    return (entrySet == null) ?
        entrySet = new AsMapEntries() :
        result;
}
```

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.

```java
HashMultimap multi0 = new HashMultimap();
HashMultimap.AsMap map0 = m.asMap();
Thread 1:
    map0.entrySet();
Thread 2:
    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.

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 genetic algorithm (GA), in which a population of candidate solutions is evolved towards satisfying a chosen coverage criterion. 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).

Godfrey 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 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 \(\{(t_1, s_1), \ldots, (t_n, s_n)\}\) 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 \(\{(t_1, s_1), (t_2, s_2)\}\) is a failure enforcing schedule, while \(\{(t_1, s_3), (t_2, s_1)\}\) 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 \(T\)

1: **procedure** GENERATESUITE\((C, n, m, v)\)
2: \(p \leftarrow\) GENERATEPREFIXSEQUENCE\((C)\)
3: \(N \leftarrow\) GETSYNCHRONIZATIONPOINTS\((C)\)
4: \(S \leftarrow\) GENERATESCHEDULES\((n, m, v, N)\)
5: \(P \leftarrow\) GENERATEPARTIALGOALS\((S, m)\)
6: \(S \leftarrow\) remove statically infeasible schedules from \(S\)
7: \(A \leftarrow\) generate method sequences for \(P\)
8: \(T \leftarrow\) \{
9:  \textbf{for} \(s \in S\) \textbf{do}
10:      \textbf{for} sequences \(\in\) GETMATCHINGS\((s, A)\) \textbf{do}
11:          \textbf{if} seq. execution of seq reaches \(s\) \textbf{then}
12:              \(T \leftarrow T \cup \{(p, \text{sequences}, s)\}\)
13: \(\}
14: \textbf{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
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 \( \text{sequences} \) 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 \( \langle (t_1, s_1), (t_2, s_2), (t_2, s_3) \rangle \): 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.

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 that need to be covered by an individual method sequence. As illustrated in Algorithm 3 illustrates, we 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 \( G \).

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 \( S \)

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

Algorithm 3 Test goal generation.

Let \( f(t, \langle (t_1, s_1), \ldots, (t_n, s_n) \rangle) \rightarrow \langle (s_{x_1}), \ldots, (s_{x_y}) \rangle \) 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 \( G \)

1: procedure GENERATEPARTIALGOALS\((S, t)\)
2: \( G \leftarrow \) empty tree
3: for \( \langle (t_1, s_1), \ldots, (t_i, s_i) \rangle \in S \) do
4: for \( t \in \{1, \ldots, t_i\} \) do
5: add\((G, f(t, \langle (t_1, s_1), \ldots, (t_i, s_j) \rangle))\)
6: return Paths\((G)\).
Here, $\alpha$ 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:

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

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 $\langle (t_1, s_1), (t_2, s_2), (t_1, s_3) \rangle$, 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

EvoSuite 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

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

Table III
RESULTING NUMBER OF TEST CASES

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

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.

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
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.

\(\text{CONSUITE}\) was able to find data races in all but three classes: The ConcurrentHashMap class in 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 StaticBinID class from the Colt library and the CopyOnWriteArrayList. Manual investigation of these classes confirms that synchronization seems to be correctly implemented.

\(\text{CONSUITE}\) 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 \(\text{CONSUITE}\). If two threads (as shown in \(\text{CONSUITE}\)'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

```java
public Object put(Object key, Object value) {
    if (fast) {
        synchronized (this) {
            TreeMap temp = (TreeMap) map.clone();
            Object result = temp.put(key, value);
            map = temp;
        }
        return (result);
    } else {
        synchronized (map) {
            return (map.put(key, value));
        }
    }
}
```

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

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 \(\text{CONSUITE}\) prototype achieved 68% concurrency coverage on average.

FastTreeMap map0 = new FastTreeMap();

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

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 expected tests, but also due to the underlying coverage criterion.

Table V lists the numbers of data races detected in detail.

<table>
<thead>
<tr>
<th>Example</th>
<th>Data Races</th>
<th>Unique Data Races</th>
</tr>
</thead>
<tbody>
<tr>
<td>CH</td>
<td>127,641</td>
<td>42</td>
</tr>
<tr>
<td>FT</td>
<td>8,005,102</td>
<td>341</td>
</tr>
<tr>
<td>FH</td>
<td>2,236,636</td>
<td>218</td>
</tr>
<tr>
<td>SB</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>AM</td>
<td>2,195</td>
<td>5</td>
</tr>
<tr>
<td>MA</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>FA</td>
<td>65,708</td>
<td>28</td>
</tr>
<tr>
<td>CR</td>
<td>461,002</td>
<td>23</td>
</tr>
<tr>
<td>CW</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table V.

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

Synchronization coverage was, except for the StaticBin1D class, upward of 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 CONSUTE 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 EVO tool). In our experiments, CONSUTE 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.

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 CONSUTE 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 CONSUTE 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 than 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