11-2019

MAP-Coverage: A novel coverage criterion for testing thread-safe classes

Zan WANG

Yingquan ZHAO

Shuang LIU

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Xiang CHEN

See next page for additional authors

Citation

WANG, Zan; ZHAO, Yingquan; LIU, Shuang; SUN, Jun; CHEN, Xiang; and LIN, Huarui. MAP-Coverage: A novel coverage criterion for testing thread-safe classes. (2019). *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019), San Diego, California, United States, November 10-15*. 722-734. Research Collection School Of Information Systems.
Available at: https://ink.library.smu.edu.sg/sis_research/4964

MAP-Coverage: a Novel Coverage Criterion for Testing Thread-Safe Classes

Zan Wang
College of Intelligence and Computing
Tianjin University, China
wangzan@tju.edu.cn

Yingquan Zhao
College of Intelligence and Computing
Tianjin University, China
zhaoyingquan@tju.edu.cn

Shuang Liu*
College of Intelligence and Computing
Tianjin University, China
shuang.liu@tju.edu.cn

Jun Sun
School of Information Systems
Singapore Management University
junsun@smu.edu.sg

Xiang Chen
School of Information Science and Technology
Nantong University, China
xchencs@ntu.edu.cn

Huarui Lin
College of Intelligence and Computing
Tianjin University, China
linhuaruitju@tju.edu.cn

***Abstract*—Concurrent programs must be thoroughly tested, as concurrency bugs are notoriously hard to detect. Code coverage criteria can be used to quantify the richness of a test suite (e.g., whether a program has been tested sufficiently) or provide practical guidelines on test case generation (e.g., as objective functions used in program fuzzing engines). Traditional code coverage criteria are, however, designed for sequential programs and thus ineffective for concurrent programs. In this work, we introduce a novel code coverage criterion for testing thread-safe classes called MAP-coverage (short for memory-access patterns). The motivation is that concurrency bugs are often correlated with certain memory-access patterns, and thus it is desirable to comprehensively cover all memory-access patterns. Furthermore, we propose a testing method for maximizing MAP-coverage. Our method has been implemented as a self-contained toolkit, and the experimental results on 20 benchmark programs show that our toolkit outperforms existing testing methods. Lastly, we show empirically that there exists positive correlation between MAP-coverage and the effectiveness of a set of test executions.**

I. Introduction

Concurrency bugs are notoriously hard to detect and debug [2], [15], and therefore, concurrent programs must be thoroughly tested. Given a test suite for a concurrent program, the question is: how do we systematically measure the richness of the test suite? For sequential programs, this question has been answered in multiple ways. In the setting of black-box testing where the specification of the program is available, we can measure the quality of a test suite based on its coverage of the specification. For instance, given a specification in the form of use cases, we can measure the percentage of use cases that have been covered by the test suite. In the setting of white-box testing, a rich family of code coverage criteria has been proposed and adopted in practice, e.g., statement coverage, branch coverage, and path coverage. Such code coverage criteria not only facilitate measuring the richness of a test suite but also provide guidelines for automatic test generation [48].

*corresponding author

Existing code coverage criteria are however mostly designed for sequential programs, and therefore, ineffective for concurrent programs. Recently, there have been several attempts on designing new coverage criteria for concurrent programs. For instance, Taylor et al. [38] pioneered a hierarchy of concurrency coverage criteria. Bron et al. [3] presented coverage metrics that are useful for human developers to create concurrent tests. The latest study is that Choudhary et al. [6] proposed a coverage metric which measures the percentage of method pairs in a thread-safe class covered by a test suite. They further proposed a test generation method called CovCon, which aims to achieve high method-pair coverage. The results showed that CovCon outperforms previous related studies on a set of concurrent benchmark programs.

In this work, we propose a new coverage criterion for concurrent programs called MAP-coverage (short for memory-access pattern). Unlike previously proposed criteria, MAP-coverage measures the richness of a set of *test executions* instead of a set of test cases. A test execution is the sequence of atomic steps executed by a test case (i.e., a program under testing which sets up multiple threads and provides inputs for each method call) with a particular thread interleaving. Because the same test case may behave very differently with different thread interleavings, we conjecture that a measurement over a set of test executions would be more accurate in capturing what behaviors of a concurrent program have been covered. Unlike existing approaches on measuring coverage of thread-interleaving [39], MAP-coverage measures thread interleaving coverage in a highly abstract way. That is, MAP-coverage abstracts test executions using *memory-access patterns* and measures what memory-access patterns are covered. A memory-access pattern captures a pattern of how a shared variable is accessed by multiple threads. It has been shown that memory-access patterns are often associated with the essence of multi-threaded bugs [27], and, furthermore, concurrency bugs can often be reduced to one or more of a set of 17 generic memory-access patterns [40].

We propose a testing method called MAPTest, which aims

978-1-7281-2508-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ASE.2019.00073

to achieve high MAP-coverage. MAPTest works as follows. Given a class which is supposed to be thread-safe, it first automatically identifies all mutable shared state variables (including those which can be accessed through de-referencing). Next, MAPTest statically analyzes every public method in the class to identify the variables which are read/written by each method. Afterwards, MAPTest generates test cases which can potentially cover certain memory-access patterns. The test cases are then automatically instrumented and executed with controlled thread interleaving, while MAPTest monitors the memory-access patterns which are covered by the test executions. After that, MAPTest generates new test cases which are likely to cover those uncovered memory-access patterns. The process continues until a bug is discovered or a testing budget is exhausted. We remark that MAPTest is further integrated with the work in [6] to use the method-pair coverage as a heuristic in generating test cases.

MAPTest has been implemented in Java and evaluated on 20 Java classes, including all programs evaluated with CovCon [6]. The results showed that MAPTest can successfully detect thread-safety violations in all 20 programs. Compared to CovCon, MAPTest detected the bugs using less time on most of the test programs, i.e., with an average speedup of 17x and a maximum speedup of 193x. Furthermore, MAPTest's performance is consistent across different runs, though there exists randomness in this method. The results also show MAPTest is effective in achieving high MAP-coverage. Lastly, we conduct an empirical study to measure the correlation between MAP-coverage and bug detection capability of a set of test executions. The results show that they are correlated and are more so than the method-pair coverage.

In short, we make the following contributions in this work. First, we propose a new coverage criterion for concurrent programs called MAP-coverage. Second, we develop a testing method called MAPTest which aims to achieve high MAP-coverage. Lastly, we implement MAPTest and empirically show that MAPTest is effective in revealing concurrency bugs.

The remainders of the paper are organized as follows. In Section II, we define the problem to be addressed. In Section III, we present details of MAP coverage. In Section IV, we show how MAPTest works. MAPTest is evaluated in Section V. We review related work in Section VI and then conclude in Section VII.

II. Problem Definition

In this section, we define our problem. The input to our approach is a thread-safe class cl. A class is thread-safe if it behaves correctly when multiple threads are allowed to access methods in the class concurrently without additional synchronization or other coordination on the part of the calling code [13]. In this work, correctness refers to the absence of data races and atomicity violations. Without loss of generality, we assume class cl is composed of a set of mutable (instance or static) variables V and a set of public methods M. Each method $m \in M$ takes an optional sequence of input parameters, and accesses some variables in V for either reading or writing (which includes reading). We use R_m (and W_m) to denote the set of variables read (and written) by method m.

```
1. public abstract class AppenderSkeleton ... {
2.   protected Priority th;
3.   public boolean
     isAsSevereAsThreshold (Priority priority) {
4.     return ((th == null) ||
5.             priority.isGreaterOrEqual(th))
6.   }
7.
8.   public void setThreshold(Priority threshold) {
9.     this.th = threshold;
10.  }
```

Fig. 1: An example class

For example, Fig. 1 shows a class from Log4j[1] which is supposed to be thread-safe. For the sake of space, only two methods are shown. Using MAPTest, we identify a previously unknown bug in the class which intuitively can be explained as follows. At line 4, method $isAsSevereAsThreshold$ checks whether *th* is null or not and returns true if it is. Otherwise, *priority* is compared with *th* at line 5. If it so happens that after line 4 is executed and before line 5 is executed, another thread executes method *setThreshold* and sets *th* be to *null*, a *NullPointerException* is generated when line 5 is executed.

A. Test cases

A test case for class cl is a (concurrent) program which invokes one or more public methods in M possibly through multiple threads. A test suite is a collection of multiple test cases. For instance, Fig. 2(a) shows a test case for the class in Fig. 1. It is written in the form of one prefix and multiple suffixes. Intuitively, the prefix is a sequential part of the test case which is executed first and then multiple suffixes are executed afterwards by different threads concurrently. In this example, there are two threads t_1 and t_2; t_1 executes the first suffix and t_2 executes the second. Note that although this test case potentially reveals the bug, it is unlikely to, if we simply run this test case multiple times. The reason is that thread t_2 must be executed after t_1 executes line 4 and before t_1 executes line 5, so that the bug can be revealed.

A test execution is a sequence of atomic steps which are executed during the execution of a test case with a particular thread interleaving. Without loss of generality, we assume that each step in a test execution is of the form (t, i, R, W) where t is a thread identifier; i is an atomic instruction; R is a set of variables read by the instruction; and W is a set of variables written by the instruction. For instance, Fig. 2(b) shows a few steps of a test execution of the test case on the left, where s_4 (and s_5, s_9) represents a step which executes line 4 (and 5, 9) in Fig. 1. That is, thread t_1 first reads the variable th with s_4. Thread t_2 then writes variable th with s_9. Finally, thread t_1 reads variable th with s_5. A test case may result in multiple test executions due to different thread interleavings. For instance, the test case shown in Fig. 2 may result in multiple test executions, and only some of them result in a *NullPointerException*.

[1] http://logging.apache.org/log4j/1.2/index.html

```
1. Prefix:
2.     NullAppender var0 = new NullAppender();
3.     Priority var1 = Priority.DEBUG;
4.
5. Suffix1:
6.     var0.isAsSevereAsThreshold(var1);
7. Suffxi2:
8.     var0.setThreshold(null);
```

(a) A test case

#	thread t_1:	thread t_2:
s_1	$(t_1, s_4, \{th\}, \emptyset)$	
s_2		$(t_2, s_9, \emptyset, \{th\})$
s_3	$(t_1, s_5, \{th\}, \emptyset)$	

(b) A test execution

Fig. 2: Sample test case and test execution

B. Coverage

A coverage criterion can be useful in multiple ways. For instance, it provides a quantitatively measure for the 'richness' of a test suite. Ideally, the higher the coverage achieved by a test suite, the more likely bugs in the program are revealed.[2] For another instance, coverage can be used as an objective function in automatic test generation (e.g., the AFL fuzzer is designed to maximize branch coverage [47]). Thus, a rich family of code coverage criteria have been proposed, e.g., statement coverage, branch coverage and path coverage. Details of these criteria can be found in a survey [48].

Unfortunately, most existing code coverage criteria are defined for sequential programs and are thus ineffective for concurrent programs. For instance, one execution of the test case in Fig. 2 covers all statements in the relevant methods of the program shown in Fig. 1 and yet it is unlikely to reveal the bug as we explained above. In recent years, multiple code coverage criteria dedicated to concurrent programs have been proposed [38], [3], [6]. The latest study in [6] proposes a coverage metric called the method-pair coverage, which measures the frequency of concurrent executions of method pairs. While it makes sense to exercise different method pairs concurrently, it may not be sufficient since CovCon is based on test cases, not test executions. For instance, in our experiment, CovCon generates the test case in Fig. 2 correctly and yet fails to reveal the bug even if we run the test case with CovCon ten times. Alternatively, we can use thread interleaving coverage as a measurement [37], [39]. The problem is that there are often many (i.e., exponential in the number of scheduling points) thread interleavings. Covering all of them would be costly and wasteful, since many of them could be considered equivalent in terms of revealing the bug. For instance, given the program in Fig. 2(a), there are many interleavings, many of which are equivalently irrelevant to the bug.

In the following, we identify 4 requirements on a code coverage criterion for thread-safe classes, which will be used to guide our work.

R1 Unlike code coverage for sequential programs, we conjecture that a coverage criterion for a concurrent program should be a function from a set of test executions to a measurement (e.g., a number from 0 to 1), instead of a function from a set of test cases to a measurement. This is because the same test case may result in different test executions under different thread interleavings, and thus a measurement based on the test cases, without considering the thread interleaving, is likely misleading.

R2 The coverage should be correlated with the bug-revealing effectiveness of the test executions, i.e., given a program, a set of test executions which achieve higher coverage should be more likely to reveal bugs in the program.

R3 Given a set of test executions, the coverage should be relatively easy to measure. Otherwise, the coverage would not be able to provide instant feedback to, for instance, tools for test case generation on-the-fly.

R4 Last, it should be intuitive so that software testers are likely to use it as a guideline for creating test cases.

[2]There are studies on whether this is true even for commonly used code coverage criteria for sequential programs [16]. Code coverage criteria, however, remain relevant unless we have better alternatives.

TABLE I: Generic memory-access patterns [27]

ID	Memory-Access Pattern
1	$(t_a, s_i, \{x\}, \emptyset), (t_b, s_j, \emptyset, \{x\})$
2	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \{x\}, \emptyset)$
3	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \emptyset, \{x\})$
4	$(t_a, s_i, \{x\}, \emptyset), (t_b, s_j, \emptyset, \{x\}), (t_a, s_k, \{x\}, \emptyset)$
5	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \emptyset, \{x\}), (t_a, s_k, \{x\}, \emptyset)$
6	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \{x\}, \emptyset), (t_a, s_k, \emptyset, \{x\})$
7	$(t_a, s_i, \{x\}, \emptyset), (t_b, s_j, \emptyset, \{x\}), (t_a, s_k, \emptyset, \{x\})$
8	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \emptyset, \{x\}), (t_a, s_k, \emptyset, \{x\})$
9	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \emptyset, \{x\}), (t_b, s_k, \emptyset, \{y\}), (t_a, s_l, \emptyset, \{y\})$
10	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \emptyset, \{y\}), (t_b, s_k, \emptyset, \{x\}), (t_a, s_l, \emptyset, \{y\})$
11	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \emptyset, \{y\}), (t_a, s_k, \emptyset, \{y\}), (t_b, s_l, \emptyset, \{x\})$
12	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \{x\}, \emptyset), (t_b, s_k, \{y\}, \emptyset), (t_a, s_l, \emptyset, \{y\})$
13	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \{y\}, \emptyset), (t_b, s_k, \{x\}, \emptyset), (t_a, s_l, \emptyset, \{y\})$
14	$(t_a, s_i, \{x\}, \emptyset), (t_b, s_j, \emptyset, \{x\}), (t_b, s_k, \emptyset, \{y\}), (t_a, s_l, \{y\}, \emptyset)$
15	$(t_a, s_i, \{x\}, \emptyset), (t_b, s_j, \emptyset, \{y\}), (t_b, s_k, \emptyset, \{x\}), (t_a, s_l, \{y\}, \emptyset)$
16	$(t_a, s_i, \{x\}, \emptyset), (t_b, s_j, \emptyset, \{y\}), (t_a, s_k, \{y\}, \emptyset), (t_b, s_l, \emptyset, \{x\})$
17	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \{y\}, \emptyset), (t_a, s_k, \emptyset, \{y\}), (t_b, s_l, \{x\}, \emptyset)$

III. MAP Coverage

In this section, we define MAP-coverage and discuss why it is meaningful for testing thread-safe classes. In the following, we fix a thread-safe class cl with a set of mutable variables V and a set of public methods M.

A. Memory-Access Patterns

Intuitively, a memory-access pattern is a pattern describing how a shared variable is accessed by multiple threads. It has been shown that memory-access patterns are often associated

Algorithm 1: Identify patterns from a test execution

```
1 let patterns_x be an empty sequence of steps;
2 for each step e in the execution do
3     if e reads x by thread a thread t_a then
4         for each subsequent step e' do
5             if e' writes x by thread t_b s.t. t_a ≠ t_b then
6                 add (e, e') into patterns_x;
```

with the essence of multi-threaded bugs [27]. Memory-access patterns can be viewed as an abstraction of the test execution, which allows us to ignore irrelevant details and yet preserve the cause of the multi-threaded bug. A memory access pattern is represented in the form of a sequence of steps. In this work, we adopt the set of 17 memory access patterns defined in [27], shown in Table I where the second column shows the sequence of steps in the memory-access pattern. It is shown that this set is complete [40] under a certain assumption, as concurrency bugs can be reduced to one or more of these patterns [21]. For instance, the fourth pattern is a scenario with three steps. First, one thread t reads a variable x. Second, a different thread writes x. Lastly, thread t reads variable x again. Note that this pattern is presented in the executions of the program shown in Fig. 1 and is very relevant to the bug.

Given a test execution, we can systematically identify the set of memory-access patterns it contains by pattern matching. That is, for each variable $x \in V$ and every pattern in Table I, we first match the first step in the pattern with a step in the test execution, and then match the second step in the pattern with a subsequent step in the test execution and so on, until all steps in the pattern are matched. For instance, Algorithm 1 shows how to identify instances of the first pattern in Table I from a given test execution with regards to variable x.

Since there are at most four steps, concerning two threads and at most two variables, in any of the memory-access patterns, the number of memory-access patterns in a test execution is bounded by $C_N^2 * C_M^2 * C_K^4$ where C_m^n is the number of combinations, where N is the number of shared variables; M is the number of threads; and K is the total number of steps in the test execution. Given a test execution t, we write $patterns(t)$ to denote the set of memory-access patterns in t (or equivalently, covered by t). Note that the thread identifiers only matter when we decide whether they are the same thread when we do the pattern matching. For instance, two instances of the first pattern in Table I $(t_a, s_i, \{x\}, \emptyset), (t_b, s_j, \emptyset, \{x\})$ and $(t_c, s_i, \{x\}, \emptyset), (t_d, s_j, \emptyset, \{x\})$ are considered the same, since they only differ by the identifiers of the threads. In the following, we assume that only one for all equivalent instances of the patterns is kept.

B. MAP Coverage

Given a thread-safe class, the MAP-coverage of a set of test executions is then calculated by the number of patterns covered by t over the total number of memory-access patterns. In general, identifying all feasible memory-access patterns of a program is challenging, just like identifying all reachable statements (for calculating statement coverage) is challenging. Nonetheless, just like we can (over)estimate the number of reachable statements by counting the total number of statements in a given program, we can (over)estimate the number of memory-access patterns as we explain below. The number of the first pattern in Table I is estimated as

$$\Sigma_{x \in V} |I_{x,R}| * |I_{x,W}| \tag{1}$$

where x is a variable in V; $I_{x,R}$ is the set of atomic instructions in the program which read x; $I_{x,W}$ is the set of atomic instructions in the program which write x; $|S|$ is the size of a set S. This is because it is in general possible to set up a thread to execute any of the instructions which read x and set up another thread to execute any of the instructions which write x afterwards. Similarly, we can estimate the number of other patterns accordingly. The total number of patterns, denoted as T_C, is then calculated as follows.

$$\begin{aligned}
T_C = {} & \Sigma_{x \in V} 2 * |I_{x,R}| * |I_{x,W}| && \text{pattern 1–2} \\
& + \Sigma_{x \in V} |I_{x,W}|^2 && \text{pattern 3} \\
& + \Sigma_{x \in V} |I_{x,R}|^2 * |I_{x,W}| && \text{pattern 4} \\
& + \Sigma_{x \in V} 3 * |I_{x,R}| * |I_{x,W}|^2 && \text{pattern 5–7} \\
& + \Sigma_{x \in V} |I_{x,W}|^3 && \text{pattern 8} \\
& + \Sigma_{\{x,y\} \subseteq V} 3 * |I_{x,W}|^2 * |I_{y,W}|^2 && \text{pattern 9–11} \\
& + \Sigma_{\{x,y\} \subseteq V} 6 * |I_{x,W}| * |I_{x,R}| * |I_{y,W}| * |I_{y,R}| && \\
& && \text{pattern 12–17}
\end{aligned}$$

We remark applying the above formula to calculate T_C in practice is still non-trivial. For instance, identifying V, $I_{x,R}$ and $I_{x,W}$ would require aliasing analysis, which is known to be challenging. Furthermore, the above formula over estimates the number of possible patterns as (1) not every instruction reading/writing v may be feasible and (2) not every pair of instructions reading/writing v may be executed concurrently (e.g., due to happens-before constraints). We leave it to future work on developing more precise estimations of T_C.

Definition 1: *(MAP-Coverage)* Let TP be a concurrent program; V be a set of shared mutable variables in TP; and TE be a set of test executions. We say that a memory-access pattern p is covered if and only if there exists at least one test execution $t \in TE$ such that $p \in patterns(t)$. The MAP coverage of T is measured as

$$\frac{|\cup_{t \in TE} patterns(t)|}{T_C} \tag{2}$$

where T_C is defined as above.

For instance, given the program show in Fig. 1 (without considering variables and methods which are not shown), T_C is calculated as follows. As there are two read instructions and one write instruction on variable th, T_C is $2 * 2 * 1 + 1^2 + 2^2 * 1 + 3 * 2 * 1^2 + 1^3$. Thus, $T_C = 16$. Assume that there is only one test execution, i.e.,

Pattern 1
$((t_1, s_4, \{th\}, \emptyset), (t_2, s_9, \emptyset, \{th\}))$
Pattern 2
$((t_2, s_9, \emptyset, \{th\}), (t_1, s_5, \{th\}, \emptyset))$
Pattern 4
$((t_1, s_4, \{th\}, \emptyset), (t_2, s_9, \emptyset, \{th\}), (t_1, s_5, \{th\}, \emptyset))$

Fig. 3: Memory-Access Patterns

the one shown in Fig. 2(b), we can systematically obtain the patterns in the test execution as discussed in Algorithm 1. The results are shown in Fig. 3, i.e., three patterns are covered. The MAP-coverage then can be calculated as $\frac{3}{16}$, i.e., this execution covers 18.75% of the memory-access patterns.

IV. MAP Coverage Guided Test Generation

In this section, we develop a testing method called MAPTest, which aims to achieve high MAP-coverage. The input is a class which is supposed to be thread-safe. The output is a set of test cases, a set of test executions generated based on the test cases, as well as a test report which summarizes the achieved MAP-coverage. The overall algorithm is shown in Algorithm 2. At line 1, we statically identity all the shared mutable variables in the given class. Note that V not only includes all those variables declared in the class but also those which can be accessed through de-referencing or due to inheritance. Furthermore, for each method m, we identify R_m (or W_m) which is the set of all variables in V that method m reads (or writes). Note that we use existing approaches for aliasing analysis. Function $identifyPatterns$ is then invoked at line 2 to identify all potential memory-access patterns.

Function $identifyPatterns$ takes the program as input and returns a set of all possible memory-access patterns. The details are shown in Algorithm 3. At line 1, for every variable x in V, we identify all instructions (in all methods) which read/write x. This is done using traditional techniques including aliasing analysis and data flow analysis. Next, for each variable (or pair of variables), we identify potential memory-access patterns on the variable (or the variable pair). For instance, at line 5, for every instruction which reads x and every instruction which writes x, we add an instance of pattern type 1 into $patterns$. The resulting $patterns$ then contains all possible patterns. Note that $patterns$ may contain patterns which are infeasible due to happens-before [20] constraints among the instructions. That is, if a pattern in $patterns$ has a step executing instruction i and then a step executing instruction j, whereas j can only happen before i, the pattern is infeasible. In MAPTest, we implement a standard happens-before inference procedure [11] to help prune infeasible patterns from $patterns$.

For the example in Fig. 1, assume that we only focus on th, there are two instructions s_4 and s_5 in $I_{th,R}$ (i.e., line 4 and 5) and one instruction s_9 in $I_{th,W}$ (i.e., line 9). We form

Algorithm 2: MAPTest: overall algorithm

```
1  identify V, R_m and W_m for each method m in M;
2  let patterns := identifyPatterns(cl);
3  while patterns is not empty do
4      if there is a pattern on x only in patterns then
5          let N := selectMethods with input x;
6      else if there is a pattern on x, y in patterns then
7          let N := selectMethods with input x, y;
8      let tc := buildTestCase(N);
9      let TE := textExecute(tc);
10     print tc and TE;
11     remove patterns(te) for each te ∈ TE from patterns;
```

the following patterns.

$(t_a, s_4, \{th\}, \emptyset), (t_b, s_9, \emptyset, \{th\})$
$(t_a, s_5, \{th\}, \emptyset), (t_b, s_9, \emptyset, \{th\})$
$(t_a, s_9, \emptyset, \{th\}), (t_b, s_4, \{th\}, \emptyset)$
$(t_a, s_9, \emptyset, \{th\}), (t_b, s_5, \{th\}, \emptyset)$
$(t_a, s_4, \{th\}, \emptyset), (t_b, s_9, \emptyset, \{th\}), (t_a, s_5, \{th\}, \emptyset)$
$(t_a, s_5, \{th\}, \emptyset), (t_b, s_9, \emptyset, \{th\}), (t_a, s_4, \{th\}, \emptyset)$
$(t_a, s_9, \emptyset, \{th\}), (t_b, s_9, \emptyset, \{th\}), (t_a, s_4, \{th\}, \emptyset)$
$\ldots$

Note that the same instruction can appear in multiple steps. For instance, to form an instance of pattern 5, a thread t_a must perform two writes on variable th (in the first and second step), whereas in this example, there is only one instruction which writes th (i.e., s_9). As a result, s_9 appears twice at the last line of the above example, which means method *setThreshold* is to be called twice.

Next, the loop from lines 3–11 in Algorithm 2 aims to generate test executions to cover the patterns in $patterns$. In particular, lines 4–7 first calls function $selectMethods$ to identity a sequence of up to 4 methods which can be used to build a test case for exercising certain pattern. The details of function $selectMethods$ are shown in Algorithm 4. We write $M_{x,R}$ to denote the set of methods which contain at least one instruction reading variable x; and $M_{x,W}$ to denote the set of methods which contain at least one instruction writing variable x. There are two cases. One is that there are uncovered patterns which involve one variable x (i.e., lines 2–6). The other is that there are uncovered patterns which involve two variables x and y (i.e., lines 7–9). In order to cover a pattern concerning with only one variable x, a sequence of two (for patterns 1–3) or three (for patterns 4–8) methods are identified. For instance, to identify a sequence of two methods for pattern 1, we first identify a method which contains an instruction which reads x and then one that writes x.

For the example shown in Fig. 1, given the variable th, method $selectMethods$ first identifies a pair of methods which potentially cover pattern 1. First, a method reading th is selected, e.g., method $isAsSevereAsThreshold$. Next, a method writing th is selected, e.g., method $setThreshold$. Similarly, sequences of methods are identified for other patterns.

Algorithm 3: Algorithm $identifyPatterns(cl)$

```
1  identify I_{x,R} and I_{x,W} for each variable x;
2  for each x in V do
3      let patterns be ∅;
4      for each s_i in I_{x,R} and each s_j in I_{x,W} do
5          add (t_a, s_i, {x}, ∅), (t_b, s_j, ∅, {x}) into patterns; //for pattern 1
6          add (t_a, s_j, ∅, {x}), (t_b, s_i, {x}, ∅) into patterns; //for pattern 2
7      for each pair of s_i and s_j in I_{x,W} do
8          add (t_a, s_i, ∅, {x}), (t_b, s_j, ∅, {x}) into patterns; //for pattern 3
9      for each s_i in I_{x,R}, s_j in I_{x,W}, and s_k in I_{x,R} do
10         add (t_a, s_i, {x}, ∅), (t_b, s_j, ∅, {x}), (t_a, s_k, {x}, ∅) into patterns; //for pattern 4
11     for each s_i in I_{x,W}, s_j in I_{x,W}, and s_k in I_{x,R} do
12         add (t_a, s_i, ∅, {x}), (t_b, s_j, ∅, {x}), (t_a, s_k, {x}, ∅) into patterns; //for pattern 5
13         add (t_a, s_i, ∅, {x}), (t_b, s_k, {x}, ∅), (t_a, s_j, ∅, {x}) into patterns; //for pattern 6
14         add (t_a, s_k, {x}, ∅), (t_b, s_i, ∅, {x}), (t_a, s_j, ∅, {x}) into patterns; //for pattern 7
15     for each s_i in I_{x,W}, s_j in I_{x,W}, and s_k in I_{x,W} do
16         add (t_a, s_i, ∅, {x}), (t_b, s_j, ∅, {x}), (t_a, s_k, ∅, {x}) into patterns; //for pattern 8
17 for each x in V and y in V do
18     for each s_i in I_{x,W}, s_j in I_{x,W}, s_k in I_{y,W}, and s_l in I_{y,W} do
19         add (t_a, s_i, ∅, {x}), (t_b, s_j, ∅, {x}), (t_b, s_k, ∅, {y}), (t_a, s_l, ∅, {y}) into patterns; //for pattern 9
20         add (t_a, s_i, ∅, {x}), (t_b, s_k, ∅, {y}), (t_b, s_j, ∅, {x}), (t_a, s_l, ∅, {y}) into patterns; //for pattern 10
21         add (t_a, s_i, ∅, {x}), (t_b, s_k, ∅, {y}), (t_a, s_l, ∅, {y}), (t_b, s_j, ∅, {x}) into patterns; //for pattern 11
22     for each s_i in I_{x,W}, s_j in I_{x,R}, s_k in I_{y,R}, and s_l in I_{y,W} do
23         add (t_a, s_i, ∅, {x}), (t_b, s_j, {x}, ∅), (t_b, s_k, {y}, ∅), (t_a, s_l, ∅, {y}) into patterns; //for pattern 12
24         add (t_a, s_i, ∅, {x}), (t_b, s_k, {y}, ∅), (t_b, s_j, {x}, ∅), (t_a, s_l, ∅, {y}) into patterns; //for pattern 13
25         add (t_a, s_j, {x}, ∅), (t_b, s_i, ∅, {x}), (t_b, s_l, ∅, {y}), (t_a, s_k, {y}, ∅) into patterns; //for pattern 14
26         add (t_a, s_j, {x}, ∅), (t_b, s_l, ∅, {y}), (t_b, s_i, ∅, {x}), (t_a, s_k, {y}, ∅) into patterns; //for pattern 15
27         add (t_a, s_j, {x}, ∅), (t_b, s_l, ∅, {y}), (t_a, s_k, {y}, ∅), (t_b, s_i, ∅, {x}) into patterns; //for pattern 16
28         add (t_a, s_i, ∅, {x}), (t_b, s_k, {y}, ∅), (t_a, s_l, ∅, {y}), (t_b, s_j, {x}, ∅) into patterns; //for pattern 17
29 return patterns;
```

If there are multiple methods containing instructions for reading or writing a variable, the choice can be resolved in different ways. Either it could be completely random, or we can adopt existing heuristics like the method-pair coverage. In the latter case, we resort to CovCon to recommend a ranked list of method pairs and check whether there exists a method pair which satisfies our requirement. If there is, we choose the top method pair; otherwise we choose randomly. Details on how method pairs are ranked can be found in [6]. For instance, to obtain a method pair for covering pattern 1 on variable th, we search through the ranked list of method pairs from CovCon. The pair $(isAsSevereAsThreshold, setThreshold)$ is selected, as method $isAsSevereAsThreshold$ contains an instruction reading th and $setThreshold$ contains an instruction writing th. Note that the same method can be selected more than once sometimes. Hereafter, we use MAPTest-random to denote an implementation of MAPTest which selects methods randomly and MAPTest to denote the one which selects methods accordingly to CovCon's recommendations.

After the methods are selected, at line 8 in Algorithm 2, function $buildTestCase$ is called to build a test case. That is, we set up an object c of type cl and two threads, t_a and t_b, which share a reference of c. Note that this is sufficient given that all patterns shown in Table I involve two threads only. Thread t_a and t_b are then set up to invoke the sequence of methods in N in an alternating order. Note that we generate parameters for each method call (including the constructor of cl) using traditional methods [25].

For instance, a test case for the class in Fig. 1 is shown in Fig. 2. First, an object of type $NullAppender$ is created in the prefix, which is a sequence of sequential code to instantiate the thread-safe class as well as randomly call methods to change the state of the shared variable. Then, based on the methods selected by $selectedMethods$, two suffixes are created to call the methods. The suffixes form the concurrent part of the test case, i.e., they are executed by different threads concurrently.

The constructed test case is then executed with a controlled scheduler at line 9 in Algorithm 2 to obtain a set of test executions TE through function $textExecute$. Function $textExecute$ (detailed in Algorithm 5) takes a test case as input, and aims to generate a set of test executions which cover as many memory-access patterns as possible. At line 1, we first identify a set of patterns which could potentially be exercised by the test case, in a way very similar to Algorithm 1 (by assuming that the program contains only the methods called in the test case). Afterwards, we aim to generate one thread interleaving for covering every pattern.

This is achieved with a scheduler which is controlled through code instrumentation. That is, we first statically insert a scheduling point in front of every instruction that reads

Algorithm 4: Algorithm *selectMethods*

```
let N be an empty sequence;
if the input is a variable x then
    add (m, n) s.t. m ∈ M_{x,R} and n ∈ M_{x,W}; //for pattern 1
    add (m, n) s.t. m ∈ M_{x,W} and n ∈ M_{x,R}; //for pattern 2
    add (m, n) s.t. m ∈ M_{x,W} and n ∈ M_{x,W}; //for pattern 3
    add (m, n, l) s.t. {m, l} ⊆ M_{x,R} and n ∈ M_{x,W};
    //for pattern 4
    add (m, n, l) s.t. {m, n} ⊆ M_{x,W} and l ∈ M_{x,R};
    //for pattern 5
    add (m, n, l) s.t. {m, l} ⊆ M_{x,W} and n ∈ M_{x,R};
    //for pattern 6
    add (m, n, l) s.t. m ⊆ M_{x,R} and {n, l} ∈ M_{x,W};
    //for pattern 7
    add (m, n, l) s.t. {m, n, l} ⊆ M_{x,W}; //for pattern 8
if the input is a pair of variables x, y then
    add (m, n, l, k) s.t. {m, n} ⊆ M_{x,W} and
      {l, k} ⊆ M_{y,W}; //for pattern 9
    add (m, n, l, k) s.t. {m, l} ⊆ M_{x,W} and
      {n, k} ⊆ M_{y,W}; //for pattern 10
    add (m, n, l, k) s.t. {m, k} ⊆ M_{x,W} and
      {n, l} ⊆ M_{y,W}; //for pattern 11
    add (m, n, l, k) s.t. m ⊆ M_{x,W} and n ⊆ M_{x,R} and
      l ⊆ M_{y,R} and k ⊆ M_{y,W}; //for pattern 12
    add (m, n, l, k) s.t. m ⊆ M_{x,W} and n ⊆ M_{y,R} and
      l ⊆ M_{x,R} and k ⊆ M_{y,W}; //for pattern 13
    add (m, n, l, k) s.t. m ⊆ M_{x,R} and n ⊆ M_{x,W} and
      l ⊆ M_{y,W} and k ⊆ M_{y,R}; //for pattern 14
    add (m, n, l, k) s.t. m ⊆ M_{x,R} and n ⊆ M_{y,W} and
      l ⊆ M_{x,W} and k ⊆ M_{y,R}; //for pattern 15
    add (m, n, l, k) s.t. m ⊆ M_{x,R} and n ⊆ M_{y,W} and
      l ⊆ M_{y,R} and k ⊆ M_{x,W}; //for pattern 16
    add (m, n, l, k) s.t. m ⊆ M_{x,W} and n ⊆ M_{y,R} and
      l ⊆ M_{y,W} and k ⊆ M_{x,R}; //for pattern 17
```

or writes a shared variable (at the bytecode level). During dynamic execution, we use a daemon thread to control the thread scheduling. That is, the daemon thread suspends all threads when a scheduling point is reached, and then selects a suspended thread to continue. If there is more than one thread which can be scheduled next to execute, we eagerly schedule the thread which would execute a step in a pattern, e.g., if the step requires executing an instruction i and a thread t is to execute i next, t is scheduled. If multiple threads can be scheduled and none of them would exercise immediately a step in the pattern, a thread is selected at random.

For instance, given the test case in Fig. 2(a) and the pattern 4 in Fig. 3, after thread t_1 reads th at s_4, we reach a scheduling point where either t_1 proceeds to read th (by executing line 5) or t_2 proceeds to write th (by executing line 9). According to the pattern, the next step should be a different thread writing th, and thus thread t_2 is scheduled. After t_2 executes s_9, another scheduling point is reached and t_1 is scheduled for the same reason. As a result, the pattern is covered.

At line 10 in Algorithm 2, the test case and the test executions are printed as a part of the report. Lastly, all patterns covered by any test execution in TE are removed from $patterns$. The loop continues until $patterns$ becomes empty (or it reaches time out). Note that Algorithms 3, 4 and 5 always terminate. However, Algorithm 5 does not guarantee

Algorithm 5: Algorithm *testExecute*

```
let P be the set of patterns which can be potentially covered
  by the test case;
for each pattern p in P do
    generate a thread interleaving which eagerly schedules
      the steps in p;
execute the test case according to the generated thread
  interleaving to obtain a set of test executions TE;
return TE;
```

that the resultant test execution will cover the pattern and, as a result, the termination of Algorithm 2 cannot be guaranteed.

V. Implementation and Evaluation

MAPTest is implemented based on JDK 8 and it is open source.[3] It is built on top of the Java bytecode analysis and modification tool ASM, which is used to insert code for controlling the thread interleaving at the bytecode level. MAPTest uses the framework $ConTeGe$ [31] to generate test cases.

A. Evaluation

To evaluate the relevance of MAP-coverage and the effectiveness of MAPTest, we conduct a set of experiments to answer the following research questions (RQs).

1) RQ1: Does MAPTest reveal bugs effectively?
2) RQ2: Does MAPTest achieve high MAP-coverage?
3) RQ3: Is MAP-coverage correlated with bug-revealing effectiveness of test executions?

Our test subjects are a set of 20 buggy 'thread-safe' classes collected from various sources including all test subjects from [6]. These examples are widely used in previous studies [6], [31]. Table II shows the details of these classes. Column *Class Name* shows the name of class. Column *Fields* shows the number of variables defined in the class, including those from its super classes. Note that we do not distinguish the access permission (e.g., *public* or *private*) of the variables, since all of the instance variables are by right shared. Column *Methods* shows the number of public methods in the class including those from the super classes. Column *LOC* shows the number of lines of code in the class including those from the super classes (counted using Statistic).[4] Lastly, column *Bug* shows the type of bug in the class. MAPTest uses a simple oracle which monitors unexpected exceptions, including assertion failures if there are assertions in the program.

All results of the experiments presented below are obtained on a machine with two octa-core CPUs Intel(R) Xeon(R) CPU E5-2640 @ 2.60GHz and 125G memory, running CentOS Linux7.4 (64 bit). The timeout is set to be 1 hour for each run. To minimize the impact of randomness, each experiment is repeated for 10 times independently with different random seeds and we report the average result.

[3]https://github.com/MAPCoverage/Map-Coverage
[4]https://plugins.jetbrains.com/plugin/4509-statistic

TABLE II: Benchmarks Description

ID	Project	Version	Package	Class Name	Fields	Methods	LOC	Bug
V1	Apache DBCP	1.4	org.apache.commons.dbcp.datasources	PerUserPoolDataSource	35	65	682	Data race
V2			org.apache.commons.dbcp.datasources	SharedPoolDataSource	30	51	516	Atomicity
V3	JDK	1.1	java.io	BufferedInputStream	7	9	237	Atomicity
V4		1.6.0	java.util	ConcurrentHashMap	15	29	1007	Atomicity
V5		1.6.0	java.util	HashTable	14	31	558	Data race
V6		1.4.1	java.util.logging	Logger	18	44	530	Atomicity
V7		1.6.0	java.lang	StringBuffer	5	52	845	Atomicity
V8		1.4.2	java.util	SynchronizedMap	6	26	79	Deadlock
V9		1.1.7	java.util	Vector	3	22	177	Atomicity
V10		1.4.2	java.util	Vector	5	51	660	Atomicity
V11	JFreeChart	1.0.13	org.jfree.data.time	Day	20	26	271	Data race
V12		0.9.12	org.jfree.chart.axis	NumberAxis	43	110	1637	Atomicity
V13		1.01	org.jfree.chart.axis	PeriodAxis	45	125	1681	Data race
V14		0.98	org.jfree.data.time	TimerSeries	12	41	331	Data race
V15		1.09	org.jfree.chart.plot	XYPlot	84	217	2788	Data race
V16		0.98	org.jfree.data	XYSeries	7	25	198	Data race
V17	Log4j	1.2.13	org.apache.log4j.helpers	AppenderAttachableImpl	1	8	92	Data race
V18			org.apache.log4j	FileAppender	7	33	410	Atomicity
V19			org.apache.log4j.varia	NullAppender	8	19	138	Atomicity
V20	Xstream	1.4.1	com.thoughtworks.xstream	Xstream	88	66	798	Atomicity

TABLE III: Results compared with CovCon and MAPTest-Random

ID	MAPTest		CovCon		MAPTest-Random		Comparison					
							M. over C.		M. over M-R.		M-R. over C.	
	Time(s)	Success Rate	Time(s)	Success Rate	Time(s)	Success Rate	Speedup	p-value	Speedup	p-value	Speedup	p-value
V1	25.3	100%	30.0	100%	93.0	100%	**1.18**	0.475	**3.67**	**0.032**	−3.10	**0.028**
V2	20.3	100%	18.2	100%	85.9	100%	−1.12	0.441	**4.23**	**0.009**	−4.72	**0.009**
V3	1.0	100%	4.5	100%	0.6	100%	**4.50**	**0.009**	−1.67	0.386	**7.50**	**0.012**
V4	147.0	100%	2811.5	40%	572.0	100%	**19.13**	**0.006**	**3.89**	0.083	**4.92**	**0.011**
V5	6.6	100%	433.1	90%	3.5	100%	**65.62**	**0.011**	−1.89	0.767	**123.74**	**0.008**
V6	90.4	100%	287.8	100%	934.6	100%	**3.18**	**0.041**	**10.34**	**0.008**	−3.25	0.083
V7	257.5	100%	1710.2	90%	674.7	100%	**6.64**	**0.041**	**2.62**	0.359	**2.53**	0.221
V8	480.5	100%	404.5	100%	1235.5	100%	−1.19	0.476	**2.57**	**0.032**	−3.05	**0.032**
V9	237.3	100%	309.7	100%	3600.0	0%	**1.31**	0.838	**15.17**	**0.006**	−11.62	**0.006**
V10	86.7	100%	413.4	90%	2636.2	40%	**4.77**	0.541	**30.41**	**0.006**	−6.38	**0.018**
V11	89.3	100%	127.0	100%	129.8	100%	**1.42**	0.359	**1.45**	0.359	−1.02	0.838
V12	31.1	100%	71.8	100%	2542.6	50%	**2.30**	**0.041**	**81.76**	**0.006**	−35.41	**0.006**
V13	36.8	100%	56.0	100%	26.8	100%	**1.52**	0.415	−1.37	0.221	**2.09**	0.076
V14	53.7	100%	53.8	100%	332.9	100%	**1.00**	0.683	**6.19**	0.097	−6.19	0.154
V15	27.7	100%	439.4	100%	2339.7	70%	**15.86**	**0.006**	**84.47**	**0.006**	−5.32	**0.008**
V16	9.2	100%	9.6	100%	55.3	100%	**1.04**	1.000	**6.01**	**0.006**	−5.76	**0.032**
V17	5.7	100%	20.9	100%	1.4	100%	**3.67**	**0.008**	−4.07	**0.012**	**14.93**	**0.006**
V18	18.7	100%	3600.0	0%	15.0	100%	**192.51**	**0.006**	−1.25	0.799	**240.00**	**0.006**
V19	239.3	100%	3600.0	0%	234.7	100%	**15.04**	**0.006**	−1.02	1.000	**15.34**	**0.006**
V20	1210.5	100%	2316.3	70%	2647.8	50%	**1.91**	**0.032**	**2.19**	**0.014**	−1.14	0.639

RQ1: Does MAPTest reveal bugs effectively? In order to answer this question, we systematically apply MAPTest to every class and measure the time elapsed before the first bug is revealed (by any execution). To examine whether there is a statistically significant difference between two methods, we use the Wilcoxon signed-rank test [44]. The significance level is set to 0.05. If p-value is smaller than 0.05, we reject the null hypotheses which means that the difference between two methods is statistically significant; otherwise we accept the null hypothesis which indicates the difference is not statistically significant.

For a baseline comparison, we compare MAPTest with the state-of-the-art approach CovCon (which has been shown [6] to significantly outperform other approaches like ConTeGe [31], AutoConTest [39] and Narada [32]). The results of the experiments are shown in Table III, where column *Time* shows the average time spent on revealing a bug, and *Success Rate* shows how many times (out of 10 runs) a bug is successfully revealed. The last six columns report the speedup and statistical test comparison between different testing methods, where column *M. over C.* is the results of MAPTest's compared to that of CovCon.

We have the following observations based on the results. First, MAPTest successfully revealed bugs in all 20 programs, whereas CovCon failed in two cases. Second, MAPTest outperforms CovCon in 18 out of 20 cases (significantly better than CovCon in 11 cases) and performs similarly in the remaining two cases. Overall, MAPTest achieves an average

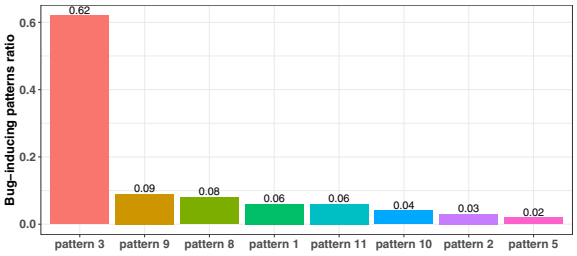

Fig. 4: Most bug-inducing patterns

speedup of 17 times and a maximum speedup of 193 times over CovCon. Third, MAPTest performs consistently across different runs for the same program, e.g., MAPTest reveals a bug in every run for every program, whereas CovCon may sometimes miss the bug for 5 out of 18 programs.

We additionally compare MAPTest with MAPTest-Random to see whether testing guided by MAP-coverage alone (without using the method-pair coverage heuristic) is useful. Note that MAPTest-Random selects methods randomly in Algorithm 4 rather than applying the method-pair coverage as a guideline. The results are summarized in Table III, where column *M. over M-R.* represents the results of MAPTest's compared to MAPTest-Random and column *M-R. over C.* represents the results of MAPTest-Random's compared to CovCon. Comparing MAPTest-Random and CovCon, MAPTest-Random performs noticeably (i.e., more than 2 times) better for 8 programs (significantly better in 6 programs) and performs noticeably worse for 10 programs. Furthermore, MAPTest-Random and CovCon complement each other, as they perform better on different programs. Comparing MAPTest and MAPTest-Random, MAPTest performs better in most of the cases (i.e., 75%) and performs noticeably worse for one program. We conclude that MAPTest effectively reveals concurrency bugs and furthermore combining MAP-coverage and method-pair coverage is effective.

We further analyze which are the memory-access patterns that successfully trigger the bug in each case. The results are shown in Fig. 4. The results show that some memory-access patterns trigger more bugs. The most bug-triggering pattern is "pattern 3", i.e., concurrent write-write on a shared variable, which accounts more than half of the cases. This observation suggest that it might be useful to prioritize certain patterns during testing, which we will explore in the future works.

RQ2: Does MAPTest achieve high MAP-coverage? In order to answer this question, we systematically apply MAPTest to every class for a total of 30 minutes and measure the MAP coverage achieved over time. The results are shown on the left of Fig. 5, where the vertical axis is the MAP-coverage achieved. Note that we run MAPTest on each program for 2 hours and take the number of patterns covered by them to be the total number of patterns, since it is in general non-trivial to know exactly how many patterns are there. The results show that the MAP coverage increases rapidly for all programs initially and continues to increase for most of the programs. The rate of increment, however, varies from program to program. For most of the programs, a high MAP-coverage ($> 50\%$) is reached after 30 minutes, whereas for 6 programs, the MAP-coverage remains low for 30 minutes. We conjecture that the reason for the latter is that MAPTest is unable to control the thread interleaving as expected to cover certain patterns.

To show the effect of selecting methods and controlling the thread interleaving in MAPTest, we additionally implement a test engine (called *Random*) which randomly selects method pairs for generating test cases and executes them without controlling thread interleaving. We then measure the difference between MAPTest and *Random*'s MAP-coverage over time. The results are shown on the right of Fig. 5, where the vertical axis is the value of MAPTest's MAP coverage *minus* that of *Random*. We can find that the difference increases monotonically over time. Furthermore, the trend for each program is similar to that of the figure on the left. This suggests that MAPTest improves MAP-coverage effectively as expected, whereas a random testing engine is ineffective at covering different memory-access patterns.

RQ3: Is MAP-coverage correlated with bug-revealing effectiveness of test executions? In order to answer this question, we design an experiment which is inspired by the study in [16]. The general idea is to measure the correlation between the MAP-coverage of a set of test executions with the number of bugs those test executions revealed. First, we systematically inject additional bugs into the test programs by removing all locking mechanisms (e.g., by tracking lock objects and the *synchronized* keyword) in the programs. The reason for injecting bugs is that there is typically only one bug in the test program, which is insufficient for calculating correlation. We then run each test program for two hours, record the total number of different bugs encountered as the total number of bugs. Two failed test executions are considered to reveal the same bug if the same exception is observed from the same instruction. We also record all the memory-access patterns observed during the two hours as the total number of patterns. We then run each program independently ten times with different random seeds, each time for 20 minutes. We record the achieved MAP-coverage and the number of bugs revealed after 5 minutes, 10 minutes, 15 minutes and 20 minutes respectively. This gives us a total of 40 data points for each program and 800 in total.

Afterwards, we calculate the relationship between MAP-coverage and bug coverage using Kendall correlation coefficients [18]. Kendall correlation is chosen (instead of Pearson Product Moment Correlation [28]) as it has fewer assumptions, e.g., it does not assume that the variables are linearly related or the data has a normal distribution. A Kendall correlation coefficient ranges from -1 to 1, where a positive value means positively correlated and a negative value means negatively correlated. According to the definition of correlation in Guildford scale [1], an absolute value of less than 0.4 means that the (positive or negative) correlation is *low*; an absolute value

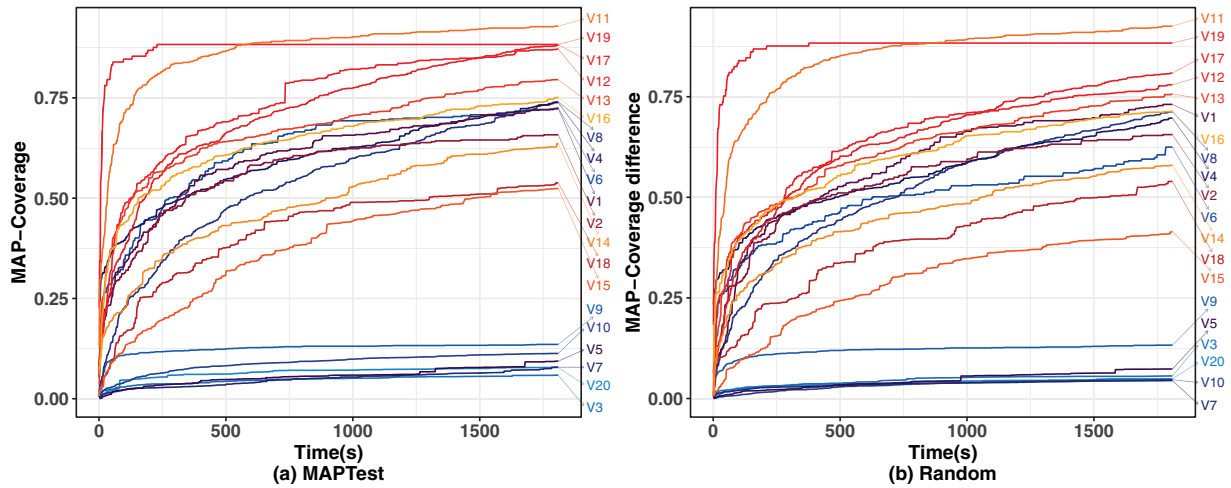

Fig. 5: MAP coverage achieved over time

between 0.4 and 0.7 means that the correlation is $moderate$; and otherwise the correlation is $high$.

For a baseline comparison, we measure the method-pair coverage and calculate the correlation between the method-pair coverage and the bug coverage similarly. That is, we first run each program using CovCon for 2 hours to obtain the total number of method pairs. We then run each program using CovCon ten times, each time 20 minutes to record the achieved method-pair coverage and the number of bugs revealed after 5, 10, 15 and 20 minutes.

The results are shown in Table IV, where the second, third and fourth columns show the total number of method-pairs, patterns and bugs (covered after executing 2 hours). Note that for 3 programs, no new bugs are injected, as there are no locks in these programs. The last two columns show the Kendall correlation coefficients between MAP-coverage and the number of bugs revealed, and that between the method-pair coverage and the number of bugs revealed. Note that an entry "—" means that the same number of patterns/method-pairs are covered in the 10 independent runs (and thus the Kendall correlation coefficients cannot be calculated), and the "*" means that the method did not find any bug. We present the average correlation values across all programs in the last row. We can observe that there are moderate correlations for 6 programs and high correlation for 1 program in the case of MAP-coverage, whereas there are moderate correlations for 3 programs in the case of method-pair coverage. On average, MAP-coverage shows a correlation of 0.43, which is much stronger than that of the method-pair coverage, which is 0.13. We thus conclude that MAP-coverage is reasonably correlated to the bug-revealing effectiveness of test executions.

Limitations. While the above experimental results show that MAPTest outperforms existing techniques, it still has some limitations which require future research. First, it may not be easy for test engineers to manually design test cases to achieve high MAP-coverage, mainly due to the difficulty in controlling the scheduling. Given a memory-access pattern, say on a certain shared variable x, one way is to manually create a test case for covering the pattern is to introduce multiple threads reading/writing x concurrently (which increases the chance of exhibiting the pattern). Furthermore, explicit thread control (like thread yield, sleep, and so on) could be used to enforce certain ordering of reading/writing the variable according to the pattern. We do acknowledge that this could be time-consuming and labor-intensive. Second, it is difficult in general to estimate the total number of feasible patterns. The estimation in Def. 1 is based on a simple static analysis and thus may be far from accurate. Knowing more precisely whether certain patterns are possible requires us to perform more complicated static analysis, which will be left

TABLE IV: Correlation Results of MAPTest and CovCon

ID	Method Pairs	Patterns	Bugs	Kendall(τ)	
				MAPTest	CovCon
V1	2145	691	1	0.19	0.20
V2	1326	462	1	0.38	0.27
V3	45	3960	18	0.61	0.32
V4	435	3608	1	0.16	0.16
V5	496	8619	10	0.74	0.19
V6	990	469	2	0.44	0.35
V7	1485	15205	42	0.28	0.54
V8	351	2354	2	0.20	0.11
V9	253	1742	17	0.29	0.10
V10	1326	7504	35	0.33	0.65
V11	351	1347	11	0.54	—
V12	6105	1560	2	0.31	0.42
V13	7875	4576	4	−0.09	0.32
V14	861	1596	9	0.53	0.39
V15	23653	24093	9	0.28	0.33
V16	325	1344	12	0.34	0.22
V17	36	1295	16	0.56	—
V18	561	1585	7	0.37	0.18
V19	190	137	2	0.56	*
V20	2211	4930	5	−0.08	0.18
All:	—	—	—	0.43	0.13

to future work. Lastly, MAPTest's employs some heuristics (e.g., the one from CovCon) to achieve high MAP-coverage, which may or may not work in general.

Threats to validity. The above evaluation suffers from two threats to validity. First, while we tried our best to collect benchmark programs, the number of programs is limited and thus it is not clear whether our conclusion above extends to other programs. Furthermore, it remains to be tested whether MAPTest performs well on large thread-safe classes which have more than a few thousand of lines of code. As other classes may interact indirectly with a given class, the scope of the analysis may need to be extended to several classes or packages. Second, although we tried our best to eliminate bugs in our implementation or effect of randomness, we cannot be completely sure.

VI. Related Work

This work is closely related to work on code coverage criteria for multi-threaded programs.

CovCon [6] generates test cases which are likely to cover these uncovered method pairs by analyzing the recorded executions to extract method pairs that are frequently executed. ConSuite [37] statically analyzes the set of thread interleavings and examines the record of executions to check if a particular thread interleaving is covered. ConSuite then applies genetic algorithms to generate tests that can cover more interleaving. AutoConTest [39] considers calling context information, dynamically and iteratively computes the coverage requirements, generates sequential tests based on sequential coverage, and assembles sequential tests into concurrent tests.

HaPSet [43] is a coverage-guided concurrency testing algorithm. The idea is to gather the ordering constraints in the program and to guide the testing of the program through analyzing the constraints. TSA [14] aims to achieve high synchronization coverage of concurrent programs by generating thread scheduling to cover uncovered coverage requirements.

Yang et al. [45] proposed a def-use pair coverage based on all-du-path coverage. Kena et al. [19] proposed a method to deriving new coverage metrics for testing concurrent software based on existing dynamic or static analysis approaches such as Eraser [33] and GoldiLocks [8]. They expanded multiple existing concurrent test coverage metrics, e.g., ConcurPairs, Definition-use coverage and Synchronisation pair coverage.

Our work is different from the above work as we proposed a new coverage criterion called MAP-coverage. MAP-coverage is more abstract than thread-interleaving coverage and is more bug-related than method-pair coverage. The experiment results show that MAPTest works more effectively than state-of-the-art approaches.

This work is broadly related to work on detecting concurrency bugs [10], [42], [22]. Research on concurrency bug detection usually focuses on three sub-problems, i.e., (1) how to improve the efficiency of the detection; (2) how to improve the effectiveness of the detection; and (3) how to reduce false positives. Happens-before analysis [23], [29] and lockset algorithms [9], [24], [7] are classic approaches for concurrency bug detection, which are widely used to detect bugs of concurrent programs. For instance, RaceChecker [23] is a data race detector which uses happens-before relation to prune infeasible races before reporting potential races to be verified. Eraser [33] proposed a lockset algorithm to detect bugs in lock-based multi-threaded programs by monitoring every shared memory references and locking behaviours. The lockset algorithm is refined for reducing overhead and false positives in [9], [7], [24], [41]. There are proposals [46], [42], [8], [5], [30] on combining the lockset algorithm with happens-before methods. However, due to limitations of static analysis methods, the lockset algorithm and the happens-before analysis methods often suffer from false positives compared to dynamic concurrency bug detection methods.

A false positive means a thread interaction that has nothing to do with defects is considered as an error. Static detection technology does not execute the program, but analyzes the source code to detect defects. So the detector cannot determine the happens-before and alias information correctly. That may lead to false positives.

In concurrent bug detection, random testing is applied as well. Methods proposed in [34], [26], [35], [17] are based on random testing and propose to 'optimize' the random scheduler in certain way for better detecting concurrency bugs. PCT [4] is a randomized scheduling method, which uses a disciplined schedule-randomization technique to provide efficient probabilistic guarantees of finding bugs. Random testing suffers from the problem of redundant exploration, i.e., the same (non-buggy) trace may be executed multiple times, which makes it hard to find bugs that hide in rare schedules.

This work is remotely related to various studies on code coverage in general, e.g., [12], [36].

VII. Conclusion

We conclude this with a discussion on whether MAP-coverage and MAPTest satisfy the requirements we established in Section II. Requirement R1 is satisfied by definition. Requirement R2 is evaluated in Section V. According to the experimental results, MAP-coverage is positively correlated with bug-coverage, although not strongly correlated. This motivates us to further investigate what is correlated with bug-coverage in the future. Requirement R3 is satisfied, as we have shown how to efficiently estimate the total number of memory-access patterns and how to obtain a set of memory-access patterns given a test execution. Lastly, we would argue that requirement R4 is partly satisfied, as creating test cases for high MAP-coverage roughly translates to creating test cases which maximize different ways of accessing shared variables concurrently.

VIII. Acknowledgment

This work is partially funded by projects 61872263, U1836214, 61802275 from National Natural Science Foundation of China, and the Singapore Ministry of Education Research Found, grant number: MOE2016-T2-2-123.

REFERENCES

[1] J B. Stroud. Fundamental statistics in psychology and education. *Journal of Educational Psychology*, 42:318, 05 1951.

[2] Francesco Adalberto Bianchi, Alessandro Margara, and Mauro Pezzè. A survey of recent trends in testing concurrent software systems. *IEEE Trans. Software Eng.*, 44(8):747–783, 2018.

[3] Arkady Bron, Eitan Farchi, Yonit Magid, Yarden Nir, and Shmuel Ur. Applications of synchronization coverage. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2005, June 15-17, 2005, Chicago, IL, USA*, pages 206–212, 2005.

[4] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ACM Sigplan Notices*, volume 45, pages 167–178. ACM, 2010.

[5] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. *ACM Sigplan Notices*, 37(5):258–269, 2002.

[6] Ankit Choudhary, Shan Lu, and Michael Pradel. Efficient detection of thread safety violations via coverage-guided generation of concurrent tests. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 266–277, 2017.

[7] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Precise race detection and efficient model checking using locksets. Technical report, Microsoft Tech Report, 2006.

[8] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: a race and transaction-aware Java runtime. In *ACM SIGPLAN Notices*, volume 42, pages 245–255. ACM, 2007.

[9] Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 237–252. ACM, 2003.

[10] Cormac Flanagan and Stephen N. Freund. FastTrack: efficient and precise dynamic race detection. In *ACM Sigplan Notices*, volume 44, pages 121–133. ACM, 2009.

[11] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, pages 234–245, 2002.

[12] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. Guidelines for coverage-based comparisons of non-adequate test suites. *ACM Trans. Softw. Eng. Methodol.*, 24(4):22:1–22:33, 2015.

[13] Brian Goetz, Tim Peierls, Joshua J. Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[14] Shin Hong, Jaemin Ahn, Sangmin Park, Moonzoo Kim, and Mary Jean Harrold. Testing concurrent programs to achieve high synchronization coverage. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 210–220, New York, NY, USA, 2012. ACM.

[15] Jeff Huang and Charles Zhang. Debugging concurrent software: Advances and challenges. *J. Comput. Sci. Technol.*, 31(5):861–868, 2016.

[16] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 435–445, 2014.

[17] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *ACM Sigplan Notices*, volume 44, pages 110–120. ACM, 2009.

[18] Maurice G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

[19] Bohuslav Krena, Zdenek Letko, and Tomás Vojnar. Coverage metrics for saturation-based and search-based testing of concurrent software. In *Runtime Verification - Second International Conference, RV 2011, San Francisco, CA, USA, September 27-30, 2011, Revised Selected Papers*, pages 177–192, 2011.

[20] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[21] Huarui Lin, Zan Wang, Shuang Liu, Jun Sun, Dongdi Zhang, and Guangning Wei. Pfix: Fixing concurrency bugs based on memory access patterns. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 589–600, New York, NY, USA, 2018. ACM.

[22] Shuang Liu, Guangdong Bai, Jun Sun, and Jin Song Dong. Towards using concurrent java api correctly. In *2016 21st International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 219–222, 11 2016.

[23] Kai Lu, Zhendong Wu, Xiaoping Wang, Chen Chen, and Xu Zhou. Racechecker: efficient identification of harmful data races. In *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*, pages 78–85. IEEE, 2015.

[24] Hiroyasu Nishiyama. Detecting data races using dynamic escape analysis based on read barrier. In *Virtual Machine Research and Technology Symposium*, pages 127–138, 2004.

[25] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-Directed Random Test Generation. In *ICSE*, pages 75–84, 2007.

[26] Chang-Seo Park and Koushik Sen. Randomized active atomicity violation detection in concurrent programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 135–145. ACM, 2008.

[27] Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold. UNICORN: a unified approach for localizing non-deadlock concurrency bugs. *Softw. Test., Verif. Reliab.*, 25(3):167–190, 2015.

[28] Karl Pearson. Notes on the history of correlation. *Biometrika*, 13(1):25–45, 1920.

[29] Dejan Perkovic and Peter J Keleher. Online data-race detection via coherency guarantees. In *OSDI*, volume 96, pages 47–57, 1996.

[30] Eli Pozniansky and Assaf Schuster. Multirace: efficient on-the-fly data race detection in multithreaded C++ programs. *Concurrency and Computation: Practice and Experience*, 19(3):327–340, 2007.

[31] Michael Pradel and Thomas R. Gross. Fully automatic and precise detection of thread safety violations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 521–530, 2012.

[32] Malavika Samak, Murali Krishna Ramanathan, and Suresh Jagannathan. Synthesizing racy tests. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 175–185, 2015.

[33] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[34] Koushik Sen. Effective random testing of concurrent programs. In *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*, pages 323–332. ACM, 2007.

[35] Koushik Sen. Race directed random testing of concurrent programs. *ACM Sigplan Notices*, 43(6):11–21, 2008.

[36] Matt Staats, Michael W. Whalen, Ajitha Rajan, and Mats Per Erik Heimdahl. Coverage metrics for requirements-based testing: Evaluation of effectiveness. In *Second NASA Formal Methods Symposium - NFM 2010, Washington D.C., USA, April 13-15, 2010. Proceedings*, pages 161–170, 2010.

[37] Sebastian Steenbuck and Gordon Fraser. Generating unit tests for concurrent classes. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*, pages 144–153, 2013.

[38] Richard N. Taylor, David L. Levine, and Cheryl D. Kelly. Structural testing of concurrent programs. *IEEE Trans. Software Eng.*, 18(3):206–215, 1992.

[39] Valerio Terragni and Shing-Chi Cheung. Coverage-driven test code generation for concurrent classes. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 1121–1132, 2016.

[40] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 334–345, 2006.

[41] Christoph von Praun and Thomas R. Gross. Object race detection. In *Acm Sigplan Notices*, volume 36, pages 70–82. ACM, 2001.

[42] Christoph von Praun and Thomas R. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *ACM Sigplan Notices*, volume 38, pages 115–128. ACM, 2003.

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

[44] Frank Wilcoxon. *Individual Comparisons by Ranking Methods*, pages 196–202. Springer New York, New York, NY, 1992.

[45] Cheer-Sun D. Yang, Amie L. Souter, and Lori L. Pollock. All-du-path coverage for parallel programs. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 1998, Clearwater Beach, Florida, USA, March 2-5, 1998*, pages 153–162, 1998.

[46] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 221–234. ACM, 2005.

[47] Michał Zalewski. AFL. http://lcamtuf.coredump.cx/afl/.

[48] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.