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

Follow this and additional works at: [https://ink.library.smu.edu.sg/sis_research](https://ink.library.smu.edu.sg/sis_research)

Part of the Software Engineering Commons

**Citation**


This Conference Proceeding Article is brought to you for free and open access by the School of Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email library@smu.edu.sg.
Author
Zan WANG, Yingquan ZHAO, Shuang LIU, Jun SUN, Xiang CHEN, and Huarui LIN
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

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

Huairui Lin
College of Intelligence and Computing
Tianjin University, China
linhuarui@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, a given 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].

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...
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 ∈ M takes an optional sequence of input parameters, and accesses some variables in V for either reading or writing (which includes reading). We use Rm (and Wm) to denote the set of variables read (and written) by method m.

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 to be 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 t1 and t2; t1 executes the first suffix and t2 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 t2 must be executed after t1 executes line 4 and before t1 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 s4 (and s5, s9) represents a step which executes line 4 (and 5, 9) in Fig. 1. That is, thread t1 first reads the variable th with s4. Thread t2 then writes variable th with s5. Finally, thread t1 reads variable th with s8. 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.

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

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

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 $C_N^2 \times C_M^2 \times C_K^2$ where $C_N^m$ 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 $(t_a, s_i, \{x\}, \emptyset), (t_d, s_j, \emptyset, \{x\})$ 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

$$\sum_{x \in V} |I_x,R| \times |I_x,W|$$

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.

$$T_C = \sum_{x \in V} 2 \times |I_x,R| \times |I_x,W|$$

pattern 1–2

$$+ \sum_{x \in V} |I_x,R|^2$$

pattern 3

$$+ \sum_{x \in V} |I_x,W|^2$$

pattern 4

$$+ \sum_{x \in V} 3 \times |I_x,R| \times |I_x,W|^2$$

pattern 5–7

$$+ \sum_{x \in V} |I_x,W|^3$$

pattern 8

$$+ \sum_{x,y \in V} |I_x,W|^2 \times |I_y,W|^2$$

pattern 9–11

$$+ \sum_{x,y \in V} 6 \times |I_x,W| \times |I_x,R| \times |I_y,W| \times |I_y,R|$$

pattern 12–17

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 $p \in \text{patterns}(t)$. The MAP coverage of $T$ is measured as

$$\frac{|\bigcup_{t \in TE} \text{patterns}(t)|}{T_C}$$

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 $2 \times 2 \times 1^2 + 2^2 \times 1 + 3 \times 2 \times 2^2 + 1^3$. Thus, $T_C = 16$. Assume that there is only one test execution, i.e.,
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 $\frac{3}{7}$, 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 identify 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 the following patterns.

\[
\begin{align*}
\{t_a, s_4, \{th\}, \emptyset\}, \{t_b, s_9, \emptyset, \{th\}\} \\
\{t_a, s_5, \{th\}, \emptyset\}, \{t_b, s_9, \emptyset, \{th\}\} \\
\{t_a, s_9, \emptyset, \{th\}\}, \{t_b, s_4, \{th\}, \emptyset\} \\
\{t_a, s_9, \emptyset, \{th\}\}, \{t_a, s_9, \emptyset, \{th\}\}, \{t_b, s_5, \{th\}, \emptyset\} \\
\{t_a, s_5, \{th\}, \emptyset\}, \{t_b, s_9, \emptyset, \{th\}\}, \{t_a, s_4, \{th\}, \emptyset\} \\
\{t_a, s_9, \emptyset, \{th\}\}, \{t_a, s_9, \emptyset, \{th\}\}, \{t_b, s_4, \{th\}, \emptyset\}
\end{align*}
\]

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.
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
1 let N be an empty sequence;
2 if the input is a variable x then
3 add \((m, n)\) s.t. \(m \in M_{x,R}\) and \(n \in M_{x,W}\); //for pattern 1
4 add \((m, n)\) s.t. \(m \in M_{x,R}\) and \(n \in M_{x,R}\); //for pattern 2
5 add \((m, n)\) s.t. \(m \in M_{x,W}\) and \(n \in M_{x,W}\); //for pattern 3
6 add \((m, n, l)\) s.t. \(\{m, l\} \subseteq M_{x,R}\) and \(n \in M_{x,W}\); //for pattern 4
7 //for pattern 5
8 add \((m, n, l)\) s.t. \(\{m, n\} \subseteq M_{x,W}\) and \(l \in M_{x,R}\);
9 //for pattern 6
10 add \((m, n, l)\) s.t. \(\{m, l\} \subseteq M_{x,W}\) and \(n \in M_{x,R}\);
11 //for pattern 7
12 add \((m, n, l)\) s.t. \(m \subseteq M_{x,W}\) and \(\{n, l\} \in M_{x,R}\);
13 //for pattern 8
14 if the input is a pair of variables \(x, y\) then
15 add \((m, n, l, k)\) s.t. \(\{m, n\} \subseteq M_{x,W}\) and
16 \(\{l, k\} \subseteq M_{x,W}\); //for pattern 9
17 add \((m, n, l, k)\) s.t. \(\{m, l\} \subseteq M_{x,W}\) and
18 \(\{n, k\} \subseteq M_{x,W}\); //for pattern 10
19 add \((m, n, l, k)\) s.t. \(\{m, k\} \subseteq M_{x,W}\) and
20 \(\{n, l\} \subseteq M_{x,W}\); //for pattern 11
21 add \((m, n, l, k)\) s.t. \(m \subseteq M_{x,W}\) and \(n \subseteq M_{x,R}\)\) and
22 \(l \subseteq M_{x,R}\) and \(k \subseteq M_{y,W}\); //for pattern 12
23 add \((m, n, l, k)\) s.t. \(m \subseteq M_{x,W}\) and \(n \subseteq M_{x,R}\)\) and
24 \(l \subseteq M_{x,R}\) and \(k \subseteq M_{y,W}\); //for pattern 13
25 add \((m, n, l, k)\) s.t. \(m \subseteq M_{x,W}\) and \(n \subseteq M_{x,R}\)\) and
26 \(l \subseteq M_{x,W}\) and \(k \subseteq M_{y,W}\); //for pattern 14
27 add \((m, n, l, k)\) s.t. \(m \subseteq M_{x,R}\) and \(n \subseteq M_{y,W}\)\) and
28 \(l \subseteq M_{x,R}\) and \(k \subseteq M_{y,W}\); //for pattern 15
29 add \((m, n, l, k)\) s.t. \(m \subseteq M_{x,R}\) and \(n \subseteq M_{y,W}\)\) and
30 \(l \subseteq M_{y,W}\) and \(k \subseteq M_{x,R}\); //for pattern 16
31 add \((m, n, l, k)\) s.t. \(m \subseteq M_{x,R}\) and \(n \subseteq M_{y,W}\)\) and
32 \(l \subseteq M_{y,W}\) and \(k \subseteq M_{x,R}\); //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 \(s_1\), 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 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 \textit{Class Name} shows the name of class. Column \textit{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., \textit{public} or \textit{private}) of the variables, since all of the instance variables are by right shared. Column \textit{Methods} shows the number of public methods in the class including those from the super classes. Column \textit{LOC} shows the number of lines of code in the class including those from the super classes (counted using Statistic).\(^4\) Lastly, column \textit{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

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

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 hypothesis 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 Con-TeGe [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
The results of MAPTest’s compared to Random keyword) in the programs. The reason for that of 50%) is reached after 30 minutes, whereas for Random, MAPTest performs better in most of the cases (i.e., 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
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
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


