Achieving High MAP-Coverage through Pattern Constraint Reduction

Yingquan Zhao, Zan Wang, *Member, IEEE,* Shuang Liu, *Member, IEEE,* Jun Sun, *Member, IEEE,* Junjie Chen, *Member, IEEE,* Xiang Chen, *Member, IEEE*

Abstract—Testing multi-threaded programs is challenging due to the enormous space of thread interleavings. Recently, a code coverage criterion for multi-threaded programs called MAP-coverage has been proposed and shown to be effective for testing concurrent programs. Existing approaches for achieving high MAP-coverage are based on random testing with simple heuristics, which is ineffective in systematically triggering rare thread interleavings. In this study, we propose a novel approach called pattern constraint reduction (PCR), which employs optimized constraint solving to generate thread interleavings for high MAP-coverage. The idea is to iteratively encode and solve path conditions to generate thread interleavings which are guaranteed to improve MAP-coverage. Furthermore, we effectively apply interpolation techniques to reduce the efforts of constraint solving by avoiding solving infeasible constraints. The experiment results on 20 benchmark programs show that our approach complements existing random testing based approaches when there are rare failure-inducing interleaving in the whole search space. Specifically, PCR finds concurrency bugs faster in 18 out of 20 programs, with an average speedup of 4.2x and a maximum speedup of 11.4x.

Index Terms—Concurrency bug detection, Constraint solving, Coverage criteria, Thread-safe class.

1 Introduction

Concurrency bugs are notoriously hard to detect and debug [1], [2], and would lead to high-profile system vulnerability [3]. Therefore, concurrent programs must be thoroughly tested. One of the fundamental challenges of testing concurrent programs is how to selectively explore the enormous search space of thread interleavings, which grow exponentially with the number of threads and the number of executed instructions.

Recently, a set of 17 generic memory-access patterns [4] has been shown to be associated with the root cause of concurrency bugs [4]. Based on this observation, several approaches based on memory-access patterns have been proposed for concurrent program testing [5] and fixing [6]. In particular, Wang *et al.* [5] proposed a new coverage criterion called MAP-coverage, for concurrent programs based on memory-access patterns. Intuitively, MAP-coverage measures the percentage of memory-access patterns covered on a set of test executions. Their evaluation results show that MAP-coverage is positively correlated with the bug-revealing effectiveness of a set of test executions. Inspired by the result, they also proposed a method called MAPTest, which randomly explores the space of thread interleavings based on heuristics to iteratively improve MAP-coverage.

- *Yingquan Zhao is with the College of Intelligence and Computing, Tianjin University, China. E-mail: zhaoyingquan@tju.edu.cn*
- *Zan Wang is with the College of Intelligence and Computing, Tianjin University, China. E-mail: wangzan@tju.edu.cn*
- *Shuang Liu is with the College of Intelligence and Computing, Tianjin University, China. E-mail: shuang.liu@tju.edu.cn*
- *Jun Sun is with the School of Information Systems, Singapore Management University, Singapore. E-mail: junsun@smu.edu.sg*
- *Junjie Chen is with the College of Intelligence and Computing, Tianjin University, China. E-mail: junjiechen@tju.edu.cn*
- *Xiang Chen is with the School of Information Science and Technology, Nantong University, China. E-mail: xchencs@ntu.edu.cn*

Corresponding author: Shuang Liu.

*Manuscript received November 4, 2020; revised ** **, ****.*

However, random testing of concurrent programs may result in exploring the same or similar thread interleavings many times, which have limited contribution to improve MAP-coverage, and may miss rare failure-inducing interleaving. Thus, it is desirable to have a systematic testing method, which avoids redundant thread interleavings and achieves high MAP-coverage. In the previous studies, researchers proposed some methods to solve the problem of redundant thread interleavings, including stateless model checking, partial order reduction (POR) [7], and maximal causality reduction (MCR) [8]. In particular, POR is applied (with stateless model checking for instance) to avoid thread interleavings that are stutter-equivalent to others. The latest proposed method called MCR [8] applies constraint solving techniques to systematically identify thread interleavings that are state-changing, i.e., that would result in a program state which is different from the explored ones. MCR has been shown to be effective in generating rare thread interleavings that reveal concurrency bugs. However, MCR still suffers from the scalability issue. The reason is that MCR aims to generate all state-changing thread interleavings, which is often enormous. Moreover, it may repeatedly solve constraints that have been proved to be unsatisfiable.

In this study, we propose a systematic approach to achieve high MAP-coverage. Our approach works as follows. It takes a concurrent program in the form of a thread-safe class and a set of test cases where multiple threads execute concurrently to access a shared object as inputs. Next, it systematically explores different interleavings of the threads in order to achieve high MAP-coverage with the provided test cases. In particular, each time it constructs and solves a constraint to generate an interleaving that is guaranteed to satisfy maximal causality reduction as well as exhibiting an unseen memory-access pattern. Furthermore, it records reasons of unsatisfiable constraints (in the form of interpolates) to avoid solving infeasible constraints. Our work can be viewed as an improvement of both MAPTest [5] and MCR [8]. Compared with MCR [8], we reduce the search space further by focusing

on generating different memory-access patterns. Moreover, we reduce the efforts of constraint solving with recorded infeasible constraints. Compared with MAPTest [5], our approach does not rely on 'chances' but rather systematically explores different memory-access patterns to achieve high MAP-coverage.

We conduct experiments with a benchmark consisting of 20 real-world concurrent programs. Compared with MAPTest [5], PCR achieves high MAP-coverage much faster on 13 programs. Moreover, PCR also exposes concurrency bugs faster. In particular, PCR is significantly better than MAPTest on 17 programs, achieving an average speedup of 4.2x and a maximum speedup of 11.4x.

In summary, we highlight our contributions as follows.

- We propose the approach PCR, which aims at achieving high MAP-coverage and concurrent bug detection efficiency.
- We evaluate PCR on 20 real-world concurrent programs, and the evaluation results demonstrate the efficiency and the effectiveness of our proposed approach.
- We implement our approach in a self-contained toolkit and make it open source for other researchers to replicate and follow our study[1].

The remaining of the paper is organized as follows. Section 2 introduces the relevant background of this study. Section 3 presents a motivating example to illustrate the main idea of our approach. Section 4 describes the details of the PCR algorithm. Section 5 reports our implementation details, the experiment settings, evaluation results, and the threats to validity. Section 6 discusses the related work and Section 7 concludes the paper.

2 PRELIMINARY

In this section, we review the relevant background.

2.1 Thread-safe Class

The test object of 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 [9]. In our study, correctness refers to the absence of data races and atomicity violations. Without loss of generality, we assume the 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 the method m.

2.2 Test cases

A test case for a class 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. 1(a) shows a test case for the class in Fig. 2. 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 afterward by executing different threads concurrently. In the example of Fig. 1(a), there are two threads t_1 and t_2; t_1 executes the first suffix, and t_2 executes the second suffix. Note that although this test case potentially reveals the bug. For instance, the Prefix in Fig. 1(a) first declares an instance of the tested class $AppenderAttachableImpl$ and does some initialization. Subsequently, the $isAttached$ method in Suffix1 is called by a thread. The $removeAllAppenders$ method in Suffix2 is called by a different thread. This test case triggers a $NullPointerException$ bug which we will describe in detail in Section 3. However, it is unlikely to always reveal this bug, if we just run this test case multiple times. The reason is that revealing this bug requires a strict constraint, i.e., the thread t_2 must be executed after thread t_1 executes line 6 and before t_1 executes line 8 or line 11, which is not easy to satisfy in a random way. Note that a test execution is a particular execution of a test case, i.e., a total order of statements in the test cases according to a specific thread interleaving.

1. https://github.com/sail-repos/PCR

TABLE 1: Generic memory-access patterns [4]

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	$(t_a, s_i, \emptyset, \{x\}), (t_b, s_j, \{y\}, \emptyset), (t_a, s_k, \emptyset, \{y\}), (t_b, s_l, \{x\}, \emptyset)$

2.3 Memory-Access Patterns

A memory-access pattern is a pattern describing how multiple threads access one or more shared variables. It has been shown that memory-access patterns are often associated with the essence of concurrency bugs [4]. Memory-access patterns can be viewed as an abstraction of the test execution, which allows us to ignore irrelevant information, yet preserve the root cause of the concurrency bug. We adopt the set of 17 memory-access patterns defined by Park *et al.* [4], shown in Table 1. A memory-access pattern is represented in the form of a sequence of steps, as shown in the second column of Table 1. Taking the fourth pattern in Table 1 as an example, it contains three steps. First, thread t_a reads the variable x at instruction s_i. Second, thread t_b writes x at s_j. Finally, thread t_a reads x again at s_k. Note that this pattern presents in the execution trace in Fig. 1(b) and is relevant to the bug. Vaziri *et al.* [11] prove that this set is complete under a certain assumption. That is, the root cause of concurrency bugs can be attributed to one or more of these patterns [6].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3144480, IEEE Transactions on Software Engineering


```
1. Prefix:
2.   AppenderAttachableImpl var0 = new AppenderAttachableImpl();
3.   ConsoleAppender var1 = new ConsoleAppender();
4.   var0.addAppender(var1);
5.
6. Suffix1:
7.   boolean var3 = var0.isAttached(var1);
8. Suffxi2:
9.   var0.removeAllAppenders();
```

(a) The test case

gid	thread t_1:	thread t_2:
i_1	$(t_1, s_6, \{aL\}, \emptyset)$	
i_2		$(t_2, s_{19}, \{aL\}, \emptyset)$
i_3	$(t_1, s_8, \{aL\}, \emptyset)$	
i_4		$(t_2, s_{20}, \{aL\}, \emptyset)$
i_5	$(t_1, s_{11}, \{aL\}, \emptyset)$	
i_6		$(t_2, s_{22}, \{aL\}, \emptyset)$
i_7		$(t_2, s_{25}, \{aL\}, \emptyset)$
i_8		$(t_2, s_{26}, \emptyset, \{aL\})$

(b) The test execution trace

Fig. 1: An example test case and a test execution trace for Fig. 2

```
 1.public class AppenderAttachableImpl ... {
 2.
 3.    protected Vector aL;
 4.
 5.    public boolean isAttached(Appender appender)
     {
 6.        if( aL == null || appender == null)
 7.            return false;
 8.        int size = aL.size();
 9.        Appender a;
10.        for(int i = 0; i < size; i++) {
11.            a = aL.elementAt(i);
12.            if(a == appender)
13.                return true;
14.        }
15.        return false;
16.    }
17.
18.    public void removeAllAppenders() {
19.        if(aL != null) {
20.            int len = aL.size();
21.            for(int i = 0; i < len; i++) {
22.                Appender a = aL.elementAt(i);
23.                a.close();
24.            }
25.            aL.removeAllElemeznts();
26.            aL = null;
27.        }
28.    }
29.}
```

Fig. 2: An example class adopted from `Log4j1` [10]

2.4 MAP-Coverage

Given a thread-safe class, the MAP-coverage of a set of test executions TE is calculated by the number of patterns covered by TE over the total number of memory-access patterns. Due to the difficulty of precisely identifying the total number of memory-access patterns (just like the difficulty of knowing how many statements are reachable when we compute statement coverage), the total number of memory-access patterns are over-estimated as the number of all read operations multiply the number of all write operations. The number of the first pattern in Table 1 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.

With a similar (over) estimation on the other patterns, the total number of patterns T_C is then computed as follows.

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

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 TE is defined as:

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

where T_C is defined as above.

3 Motivating Example

Fig. 2 shows a motivation example adopted from `Log4j1` [10] which is supposedly thread-safe. For the sake of space, we only show two methods that are related to the concurrency bug. At line 6, method $isAttached$ checks whether aL and $appender$ are $null$. If neither is $null$, aL is read at lines 8 and 11 of the method. At line 19, method $removeAllAppenders$ first checks whether aL is $null$. If it is not $null$, the elements in aL are cleared through lines 20, 22, and 25 of the method, and then set aL to $null$ at line 26. Therefore, if a thread checks that aL and $appender$ are not $null$ at line 6 of the method $isAttach$ and has not yet executed line 8 or line 11, another thread executes method $removeAllAppenders$ and sets aL to $null$ at line 26, a $NullPointerException$ is generated when line 8 or line 11 is executed.

The existing approach MAPTest [5] first statically analyzes the test class's bytecode instructions and then finds the potential memory-access patterns. In this example, there are three read instructions on aL in the $isAttached$ method (lines 6, 8 and 11), four read instructions (lines 19, 20, 22 and 25) and one write instruction on aL (line 26) in the $removeAllAppenders$ method. Therefore, MAPTest estimates that there are nine memory-access patterns, as shown in Fig. 3, and then generates test cases which are randomly executed in order to trigger the patterns. However, due to the limitation of static analysis, there are redundant and

Pattern of length 2
p_1: $((t_1, s_6, \{aL\}, \emptyset), (t_2, s_{26}, \emptyset, \{aL\}))$
p_2: $((t_2, s_{26}, \emptyset, \{aL\}), (t_1, s_6, \{aL\}, \emptyset))$
p_3: $((t_1, s_8, \{aL\}, \emptyset), (t_2, s_{26}, \emptyset, \{aL\}))$
p_4: $((t_2, s_{26}, \emptyset, \{aL\}), (t_1, s_8, \{aL\}, \emptyset))$
p_5: $((t_1, s_{11}, \{aL\}, \emptyset), (t_2, s_{26}, \emptyset, \{aL\}))$
p_6: $((t_2, s_{26}, \emptyset, \{aL\}), (t_1, s_{11}, \{aL\}, \emptyset))$
Pattern of length 3
p_7: $((t_1, s_6, \{aL\}, \emptyset), (t_2, s_{26}, \emptyset, \{aL\}), (t_1, s_8, \{aL\}, \emptyset))$
p_8: $((t_1, s_6, \{aL\}, \emptyset), (t_2, s_{26}, \emptyset, \{aL\}), (t_1, s_{11}, \{aL\}, \emptyset))$
p_9: $((t_1, s_8, \{aL\}, \emptyset), (t_2, s_{26}, \emptyset, \{aL\}), (t_1, s_{11}, \{aL\}, \emptyset))$

Fig. 3: The memory-access patterns generated for the example class by MAPTest (p_1 - p_9) and PCR (highlited in blue)

t_1 *happens-before* :	($i_1 < i_3$)
t_2 *happens-before* :	($i_2 < i_4$), ($i_4 < i_6$), ($i_6 < i_7$), ($i_7 < i_8$)
Pattern Constraint :	($i_1 < i_8$), ($i_8 < i_3$)

Fig. 4: Constraints generated based on Fig. 1(b) and p_7 of Fig. 3

meaningless memory-access patterns in the estimated results. For patterns of length 3, p_8 represents a relaxed set of constraints, which can be replaced by the situations of p_7 and p_9. We refer to this situation as the interrupted pattern, and the details are introduced in Section 4.2. Furthermore, since MAPTest is based on random testing, it can not guarantee to cover any of the patterns. For instance, given the test case shown in Fig. 1(a), which can potentially cover all the memory-access patterns in Fig. 3 with proper scheduling. However, we conduct an experiment with MAPTest on this test case and MAPTest fails to cover any of the patterns in most runs. As a result, MAPTest took 97.7% more scheduling than PCR to trigger the failure.

In the following, we show how PCR works to cover as many patterns as possible from a given test case. Fig. 1(a) is an example test case for the class in Fig. 2. PCR obtains the given test case's execution information from the dynamic execution results and then identifies the memory-access patterns that have not been covered yet. PCR first randomly executes the test case, and gets an initial execution trace, as is shown in Fig. 1(b), the gid indicates the global id of each event in the execution trace. With the execution trace, PCR constructs five memory-access patterns, i.e., p_2, p_4, p_6, p_7, and p_9 which are highlighted in blue in Fig. 3. Thanks to dynamic execution, PCR is able to precisely identify the memory-access patterns, in contrast to the over-estimation of MAPTest.

After obtaining all patterns, PCR first selects a target uncovered pattern and constructs the constraints required to cover the pattern. Note that the longer patterns are selected first since the more constraints contained in the pattern the more specific the generated schedule is. The complete list of constraints that are required to cover the pattern are shown in Fig. 4, including the happens-before constraints of t_1, the happens-before constraints of t_2 and the pattern constraints. The happen-before constraint restricts the execution order of events from the same thread, i.e., those events must follow the same order as that of the original execution trace. For example, the happens-before constraint of t_1 requires that $i_1 < i_3$, which means that i_1 must be executed before i_3. According to the sequential consistency memory model [8], the execution results of i_3 depends on the execution results of i_1. PCR also constructs pattern constraints based on the patterns that are generated from the given execution trace. In this example, according to pattern p_7, i_8 needs to be executed in between i_1 and i_3. Therefore, PCR generates two constraints, i.e., $i_1 < i_8$ and $i_8 < i_3$. After obtaining all the above constraints, PCR applies an SMT-Solver [12] to compute a new schedule that satisfies all the constraints. A feasible schedule obtained from the constraint solving result is shown as follows:

$$i_2 < i_4 < i_6 < i_7 < i_1 < i_8 < i_3$$

According to this schedule, after the thread t_1 checks that aL is not NULL at i_1, the thread t_2 assigns aL to NULL at i_8, and when the thread t_1 is scheduled to execute i_3, a $NullPointerException$ will be triggered.

Algorithm 1: PCR: overall algorithm

```
1 given a testcase tc for cl;
2 let Φ_u ← ∅ be the set of unsatisfiable constraints, initially
    empty;
3 let FT ← patternConstraintExplore(tc, ∅, Φ_u);
4 print FT;
```

Algorithm 2: *patternConstraintExplore*(tc,schedule,Φ_u)

```
1 let trace ← execute(tc, schedule);
2 let P ← patternConstraintCreation(trace);
3 for each pattern p in P do
4 |   let Φ be the constraints obtained for trace on p;
5 |   let s ← patternConstraintSolving(Φ, Φ_u);
6 |   schedules.add(s)
7 for each schedule s in schedules do
8 |   let FT ← patternConstraintExplore(tc,s, Φ_u);
9 return FT;
```

4 Approach

In this section, we introduce the details of our proposed approach called Pattern Constraint Reduction (PCR). PCR aims to generate effective thread scheduling to achieve high MAP-coverage and consequently expose bugs in thread-safe classes. In particular, given a test case for a thread-safe class, PCR first executes it randomly to obtain an execution trace (i.e., thread scheduling) and then generates a set of constraints to explore uncovered patterns to achieve higher MAP-coverage. The input of PCR is a test case for a class that is supposed to be thread-safe. The output is a test execution that triggers the concurrency bug, which violates thread-safety and a test report that summarizes the achieved MAP-coverage.

The overall algorithm of PCR is shown in Algorithm 1. The input is a test case tc designed to expose concurrency bugs in the thread-safe class cl. The test case can be obtained in different ways. One is to manually design test cases by randomly calling methods in the thread-safe class under test. In this work, we adapt existing test case generation tools (e.g., which is a part of MAPTest [5]), which automatically generates test cases. Each test case consists of a prefix and two suffixes, as is shown in Fig. 1(a). Specifically, the prefix is used to instantiate an object cl for the thread-safe class and randomly call some methods to change the declared object's state. The two suffixes are used to set up two threads t_a and t_b to call methods in a method pair correspondingly and iteratively generate the parameters needed for all methods. The two threads access the shared object cl declared in the prefix

Algorithm 3: *patternConstraintCreation*($trace$)

```
1  identify all shared variables V in trace;
2  let patterns ← ∅;
3  for each variable x in V do
4      identify I_{x,R} and I_{x,W} for each variable x;
5      //Add patterns of length 2
6      for each s_i in I_{x,R} and each s_j in I_{x,W} do
7          if thread(s_i) ≠ thread(s_j) then
8              add (s_i,s_j) into patterns;
9              add (s_j,s_i) into patterns;
10     for each pair of s_i and s_j in I_{x,W} do
11         if thread(s_i) ≠ thread(s_j) and s_i != s_j then
12             add (s_i,s_j) into patterns;
13             add (s_j,s_i) into patterns;
14     //Add patterns of length 3
15     for each pair of patterns < p_1, p_2 > in patterns(x) do
16         if thread(p_1.s_1) == thread(p_2.s_2) and
17         p_1.s_1 ≠ p_2.s_2 and p_1.s_2 == p_2.s_1 then
18             add (p_1.s_1, p_1.s_2, p_2.s_1) into patterns;
19 //Add patterns of length 4
20 for each variable x, y ∈ V do
21     for each pattern p_1 ∈ patterns(x) and each pattern p_2 ∈
       patterns(y) do
22         if p_1.length == 2 and p_2.length == 2 and
23         thread(p_1.s_1) == thread(p_2.s_2) and
24         thread(p_1.s_2) == thread(p_2.s_1) then
25             //x, x, y, y
26             add (p_1.s_1, p_1.s_2, p_2.s_1, p_2.s_2) into patterns;
27             //x, y, x, y
28             add (p_1.s_1, p_2.s_1, p_1.s_2, p_2.s_2) into patterns;
29             //x, y, y, x
30             add (p_1.s_1, p_2.s_1, p_2.s_2, p_1.s_2) into patterns;
31 for each pattern p in patterns do
32     if p satisfies the filtering rules in Table 2 then
33         remove p in patterns;
34 return patterns;
```

TABLE 2: Memory-access pattern filtering rules

No.	Rules	Description
1	**Non-defined**	The pattern is not defined in Table 1
2	**Covered**	The pattern has been covered in the current or previous trace
3	**Happens-before**	The events originated from the same thread violate the happens-before relation
4	**Old-state**	The pattern can not generate a new program state
5	**Interrupted**	The pattern was interrupted by a write instruction
6	**Redundant**	The pattern is redundant

concurrently. We use Φ_u to store unsatisfiable constraints (line 2) to avoid infeasible thread interleavings. Then the given test case is explored with the $patternConstraintExplore$ algorithm (i.e., Algorithm 2).

4.1 Test Case Exploration

Algorithm 2 first executes the given test case with a random schedule to obtain a test execution, then identifies patterns from the test execution, which may change the program states, and then generates schedules guided by the patterns. In particular,

Algorithm 4: *patternConstraintSolving* (Φ, Φ_u)

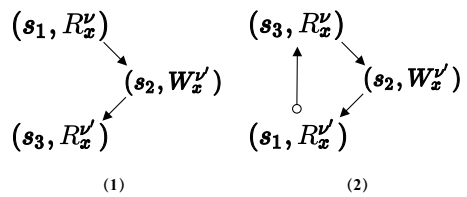
```
1  if ∃φ_u ∈ Φ_u such that φ_u ∈ Φ then
2      return null;
3  let s, φ_u ← SolveConstraints (Φ);
4  if s ≠ null then
5      return s;
6  else
7      Φ_u.add(φ_u) ;
8      return null;
```

it first executes the test case tc on the program according to the given schedule (initially empty) and obtains an execution trace $trace$ (line 1). Then we obtain a set of patterns P with Algorithm 3 (line 2). The patterns in P do not occur in $trace$, and they involve instructions, which read/write a different value on a shared variable. For each pattern $p \in P$, we obtain the set of pattern constraint Φ for $trace$ according to p (line 4). The pattern constraint is constituted with the must-happen-before constraint, the lock-mutual-exclusion constraint, and the data-validity constraint. Intuitively, the happen-before constraints make sure that the correct execution order of events from the same thread. The lock-mutual-exclusion constraints make sure that the correct acquisition and release of synchronization locks. The data-validity constraints ensure that the newly generated schedule is feasible. We refer to the readers to MCR [13] for details on how these constraints can be systematically obtained. We also add the pattern constraint, which is in essence a happens-before constraint on the instructions of the pattern. For each of the patterns, PCR explores the possible thread interleavings through the $patternConstraintSolving$ algorithm (line 5), which returns the generated thread schedule on the given test case, if any. The generated schedule is added to the set of schedules (line 6) for each pattern in P. Then we execute the test case tc on all of the generated schedules until a failing test execution FT is triggered or all of the schedules are explored (lines 7-8). The test execution results are returned (line 9). Note that the for loop in lines 3-6 and lines 7-8 can be executed in parallel. In addition, we maintain a pool of explored schedules and check on the newly generated schedules to avoid duplicated executions.

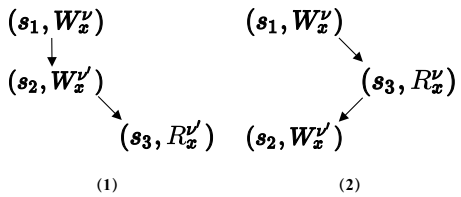
4.2 Pattern Constraint Creation

Algorithm 3 shows the steps of identifying patterns that satisfy the pattern constraints shown in Table 1. Our algorithm first identifies all read and write instructions on all shared variables (line 4), and then constructs pattern candidates of length 2 (lines 6-13). $I_{x,R}$ represents the instruction reads on variable x, and $I_{x,W}$ represents the instruction writes on variable x. We add all read-write, write-read and write-write instruction pairs on the same shared variable from different threads into the set of pattern candidates. Based on the pattern candidates of length 2, we construct pattern candidates of length 3 (lines 15-18) and pattern candidates of length 4 (lines 20-30). Note that patterns of length 3 and 4 are constructed only based on feasible length-2 patterns, i.e., lines 16-17, lines 22-24 where s_1 and s_2 are the first and second instruction in a length 2 pattern candidate. $p_1.s_1$ is the first instruction of the first pattern candidate of length 2. This allows us to reduce the implementation cost and runtime overhead of constructing patterns of length 3 and 4. In general, it can be easily shown that all the patterns of length 3 and 4 can be obtained by combining two patterns of length 2

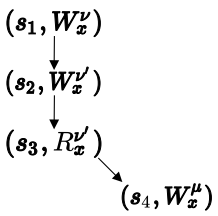
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3144480, IEEE Transactions on Software Engineering



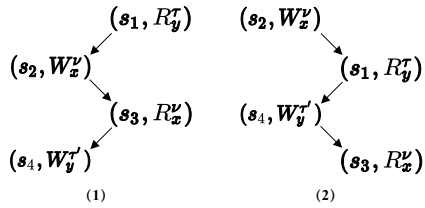
(a) Illustration on the happens-before rule



(b) Illustration on the read different value rule



(c) Illustration on the skip interrupted pattern rule



(d) Illustration on the redundant exchange rule

Fig. 5: Memory-Access Pattern filtering rules

in general. For instance, p_9 is the combination of two instances of p_3 in Table 1. After obtaining all the pattern candidates, We then check the pattern candidates with the pattern constraints and remove those patterns that do not satisfy the constraints (lines 31-33).

To reduce the cost of constraint solving, we propose six filtering rules, based on which the generated pattern constraints are filtered. We check the pattern constraint candidates following the order of the rules listed in Table 2. We first use **Rule 1** to filter the patterns that are not defined in Table 1 (Non-defined), and use **Rule 2** to filter the patterns that are already covered in existing traces (Covered). Then we further use the other four rules to filter the patterns with the following steps.

Rule 3: Satisfy the happens-before relation. The instructions from the same thread in the generated pattern must satisfy the happens-before relation, i.e., the instructions from the same thread must obey the original sequential order. For instance in Fig. 5(a), we can construct a pattern of length 3 $(s_3, R_x^\nu), (s_2, W_x^{\nu'}), (s_1, R_x^{\nu'})$ (i.e., (2) of Fig. 5(a)) based on the length 2 patterns $(s_3, R_x^v), (s_2, W_x^{\nu'})$ and $(s_2, W_x^{\nu'}), (s_1, R_x^{\nu'})$, and this pattern has not been covered. (s_1, R_x^ν) represents the read operation on the variable x by the instruction s_1, which reads the value v. However, based on the happens-before relation rule, the newly constructed pattern is invalid since the two read instructions (s_3, R_x^v) and $(s_1, R_x^{\nu'})$ are from the same thread. Under the sequential consistent memory model [8], the instructions of the same thread must satisfy the happens-before relation rule, i.e., the read operation s_1 must happen before the read operation s_3.

Rule 4: Generate a new program state. The generated thread interleaving should have a different output value, i.e., the read instruction reads a different value (on the targeted shared variable) from the original test execution, or the write operation changes orders. One example is illustrated in Fig. 5(b), where the read operation $(s_3, R_x^{\nu'})$ reads the value ν' from the write operation $(s_2, W_x^{\nu'})$. To guarantee the read operation $(s_3, R_x^{\nu'})$ can read a different value (i.e., $\nu \neq \nu'$), we need to construct a pattern of length 3 to ensure that the read operation of $(s_3, R_x^{\nu'})$ is executed after (s_1, W_x^ν) and before $(s_2, W_x^{\nu'})$.

Rule 5: Skip interrupted patterns. The generated patterns should avoid being interrupted by other patterns. Take the execution trace shown in Fig. 5(c) as an example, suppose we want to construct a pattern of length 3, e.g., $(s_1, W_x^\nu), (s_4, W_x^\mu), (s_3, R_x^\mu)$, which allows (s_3, R_x^μ) to read the value μ from (s_4, W_x^μ). The value read by s_3 is μ since it is executed after s_4, which writes μ to x (which is different from the original value read by s_3). However, this pattern does not restrict the execution order of $(s_2, W_x^{\nu'})$ and (s_4, W_x^μ). Specifically, if s_4 is executed before s_2, this pattern can not generate a new program state since the value read by s_3 is still the value written by s_2; if s_4 is executed after s_2, a new program state is generated. However, such a situation can be replaced by $(s_2, W_x^{\nu'}), (s_4, W_x^\mu), (s_3, R_x^\mu)$. Therefore, we do not allow interrupted patterns, i.e., the instructions from the same thread must be consecutive in the generated pattern.

Rule 6: Remove redundant exchange. For some patterns of length 4, which involve operations on two shared variables, changing from one pattern to the other does not result in new values of the shared variable, and the changed behavior between two variables do not affect each other. We thus filter such redundant patterns. For instance in Fig. 5(d), suppose we generate a new pattern of length 4, i.e., $(s_2, W_x^\nu), (s_1, R_y^\tau), (s_4, W_y^{\tau'}), (s_3, R_x^\nu)$ as shown in (2), according to the original pattern in (1). The new generated pattern does not lead to a new program state since both x and y read the same value. Therefore it is redundant and can be removed from the generated patterns.

For each of the executed trace, we construct all candidate patterns, and try to match them with the generic patterns shown in Table 1 as many as possible. In this way, we maximize the pattern coverage of the given test case, whilst filtering patterns that cannot introduce a new program state, with the aim of uncovering concurrency bugs efficiently. Finally, the patterns that satisfy all pattern constraints (i.e., *patterns*) are returned.

In the following, we discuss the soundness of PCR. PCR takes a test case of thread-safe class as input, and iteratively identity all potential memory-access patterns that can be formed by executing the test case with different interleaving of the threads. Given an execution trace (of a test case), Algorithm 3 is used to identify all potential patterns. We define the soundness of Algorithm 3 as follows: given a test case, if there exists an interleaving such that the test execution exhibits a memory-access pattern pt and results in a set of program state ST, Algorithm 3 generates at least one execution trace that exhibits pt unless all program states

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3144480, IEEE Transactions on Software Engineering

TABLE 3: 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	Apache Tomcat	7.0	org.apache.catalina.connector	InputBuffer	20	22	575	Data race
V4		7.0	org.apache.catalina.connector	Request	59	143	3736	Data race
V5	JDK	1.1	java.io	BufferedInputStream	7	9	237	Atomicity
V6		1.6.0	java.util	ConcurrentHashMap	15	29	1007	Atomicity
V7		1.6.0	java.util	HashTable	14	31	558	Data race
V8		1.4.1	java.util.logging	Logger	18	44	530	Atomicity
V9		1.6.0	java.lang	StringBuffer	5	52	845	Atomicity
V10		1.1.7	java.util	Vector	3	22	177	Atomicity
V11		1.4.2	java.util	Vector	5	51	660	Atomicity
V12	JFreeChart	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

ID is the unique identity of the tested class. **Project**, **Version** and **Package** are the corresponding project, version and package that the thread-safe classes being tested are originated; **Class Name** is the name of the tested class; **Fields** and **Methods** are the number of variables defined in the class, including those from its super class, and the number of public methods, respectively; **LOC** shows the number of lines of code in the class. The concurrency bugs in these classes mainly include data race and atomicity violation, as shown in the **Bug** column.

in ST have been explored (by execution traces generated by Algorithm 3). Intuitively, Algorithm 3 is sound if it never misses any pattern which results in a new program state.

The soundness of Algorithm 3 depends on whether the filtering rules are sound, i.e., given a test case, applying the rules does not prevent us from exploring a new memory-access pattern which results in a new program state if a different interleaving is adopted. A new program state means that a read instruction reads a different value [13]. Note that a branch that is not covered may become covered only if a new program state is reached. Our soundness discussion follows in spirit that of MCR presented [13]. In fact, MCR may be regarded a special case of our approach, i.e., only Rule 4 are applied. From another point of view, our approach further filters those interleaving 'selected' that exhibit the same memory-access patterns.

In terms of each specific rule, the effect of Rule 1 is that we only focus on the set of 17 memory-access patterns shown in Table 1. We refer the readers to [4], [5] for reasons on why these memory-access patterns are useful. Rule 3 filters out infeasible patterns that do not satisfy happens-before relation. Since our happens-before relation is an under-approximation in general, only infeasible patterns are filtered. The soundness of Rule 4 follows that of MCR [13]. The soundness of Rule 5 and Rule 6 have been argued above, i.e., they only filter out those patterns that do not result in new program states. The only rule that is not sound is Rule 2. That is, Rule 2 may filter out interleavings that result in different program states but have the same memory access patterns. This is a price that we pay to reduce the number of interleavings to be explored. Intuitively, this is justified in practice as it has been shown that different behaviors are often associated with different memory-access patterns, i.e., while in theory Rule 2 is not sound, in practice, the missed behaviors due to Rule 2 may be few. This is partly evidenced through our empirical evaluation.

4.3 Pattern Constraint Solving

Algorithm 4 generates a valid thread schedule based on the given constraint set, or returns if the constraints are unsatisfiable. To further reduce the invocation of the constraint solver, which is often the performance bottleneck, we make use of the unsatisfiable core as a further filter. That is, for each constraint which is proven unsatisfiable, we record its unsatisfiable core, which is then used to filter those pattern constraints which have the same set of constraints in any of the unsatisfiable cores. For instance, our algorithm first checks whether there exists an unsatisfiable core using the SMT solver, which is a subset of the constraints to be solved (line 1). If it is the case, we will return without solving the constraints (lines 2). Otherwise, we will solve the constraints, which returns either a valid schedule s or an unsatisfiable core ϕ_u (line 3). If s is valid (not null), we return s (line 4-5). Otherwise, the unsatisfiable core ϕ_u is added to the set of unsatisfiable core Φ_u and return null (lines 6-8).

5 Implementation and Evaluation

In this section, we first present the implementation details of our proposed approach. Then, we evaluate its effectiveness and efficiency.

5.1 Implementation

We implement our approach as a self-contained toolkit for testing Java programs. PCR is built upon the bytecode analysis and modification tool ASM [14]. The instrumented class dynamically records the execution trace of the test case and controls the thread schedule. To generate a new program schedule, PCR constructs a constraint file by analyzing execution trace and solves the constraints through a popular constraint solver Z3 [12]. MAPTest [5] is adopted to generate the test cases used in the experiment.

5.2 Evaluation Settings

Table 3 lists all the programs we used for evaluation. The benchmark contains a set of 20 buggy 'thread-safe' classes derived from real-world applications and have been widely adopted in previous studies [5], [15], [16].

Since there are no available test cases, we adopt MAPTest [5] to generate a set of 1,000 test cases for each class in Table 3. We explicitly mark those test cases which trigger thread-safe violations. We ensure there is at least one such test case in the generated test case set by keep executing MAPTest until a thread-safety violation is triggered which often takes hours. To check whether PCR successfully exposes a concurrency bug, we use a simple oracle which monitors unexpected exceptions and assertion failures.

All our experiments are conducted on a server with two octa-core CPUs Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60GHz and 125G RAM. The operating system is Ubuntu 18.04.2 LTS (64 bit). The time limit is set to be 3,600s for each run. To reduce the effect of randomness, we run each experiment 10 times independently and report the average result.

Our experiment is designed to evaluate two aspects. First, we explore and analyze the ability of MAPTest in covering the specified memory-access patterns (RQ1). Second, we explore whether PCR can improve existing work on achieving high MAP-coverage (RQ2) and generate failure-inducing schedules (RQ3).

5.3 Evaluation Results

RQ1: Are existing random testing based approaches effective in covering the intended pattern?

The main idea of MAPTest is generating a set of test cases that can achieve high MAP-coverage, and then to reveal concurrency bugs[2]. To achieve the goal, each time MAPTest selects a memory-access pattern that is not covered or less frequently covered to construct a test case, and tries to generate a schedule execution that covers the intended pattern. However, it relies on simple heuristics for schedule generation and may miss the intended pattern. To evaluate how effective MAPTest can cover the intended pattern, for each thread-safe class, we randomly select a test case to execute 50 times and record the number of times that MAPTest successfully covers the intended pattern.

The evaluation results are shown in Column "RQ1-CIP(M.)" of Table 4, where CIP is short for Cover Intended Pattern, and M. represents the results of MAPTest. Among the 20 test programs, MAPTest covers the intended pattern sometimes for only 5 test programs. For all the other programs, MAPTest fails to cover the intended pattern for all 50 runs. The reason for the poor performance is that MAPTest matches patterns by statically analyzing the bytecode instructions of the test class, which is inaccurate (e.g., the program path for covering the pattern might be infeasible). Furthermore, MAPTest tries to cover the intended pattern by utilizing random testing with simple heuristics, which very relies on "chance". Therefore, the generated schedule may not be able to cover the intended pattern.

RQ2: Is PCR effective in achieving high MAP-coverage?

We design this experiment to observe whether the proposed PCR approach is effective in achieving high MAP-coverage. In particular, for each thread-safe class under test, we apply PCR and MAPTest to a set of 1,000 test cases. Note that, for fairness, we run MAPTest to randomly generate 1,000 test cases as test subjects, and run each method on the same set of ordered test cases[3] to observe the MAP-coverage changes. Moreover, MAPTest tries to cover the pre-matched pattern through simple scheduling. As the results of RQ1 shows, MAPTest has a relatively limited capability of covering intended patterns due to the randomness in the heuristics. We run 10 times on each test case for MAPTest to improve its chance of covering the intended pattern. The timeout is set to 10 minutes, and the MAP-coverage is recorded every 500 ms. We run 1,000 test cases repeatedly until the timeout. To reduce the effect of randomness, we repeat the experiment 5 times and report the average result. Then we plot the MAP-coverage change with the increasing number of test cases for PCR and MAPTest. The experimental results are shown in Fig. 6, where the horizontal axis represents the running time, and the vertical axis shows the MAP-coverage achieved over time. Since it is nontrivial to know exactly how many patterns there are in the program, we use the highest number of patterns obtained in the two methods as an estimation of the total number of patterns when calculating the MAP-coverage.

From Fig. 6, we can observe that for PCR, the MAP-coverage increases monotonically over time for most programs. For some programs, such as XStream, there are few write operations on the shared variables and most of the write operations are in private and protected methods. Therefore, the total number of patterns is small, which explains why the MAP-coverage jumps rather than increasing gradually. Compared with MAPTest, PCR can obtain higher MAP-coverage faster on 13 projects. PCR also achieves higher or equal MAP-coverage than MAPTest after 10 minutes on 19 programs. The only exception is the StringBuffer class (V9 in Fig. 6), where MAPTest achieves higher MAP-coverage faster. By a close inspection of the class, we find that most of the pattern constraints generated by PCR are due to a large number of write operations to a char array defined in the class, and the pattern constraints do not generate new program states, which are filtered by the Old-state rule by PCR. Those filtered patterns are not considered as covered by PCR, and thus MAPTest achieves higher coverage in this case. From the results, we can observe that PCR is able to achieve higher MAP-coverage faster, as compared with MAPTest.

RQ3: Is PCR effective in generating failure-inducing schedules on a given test case?

To answer this question, we randomly select a test case, which can potentially expose thread-safety violations in the tested class from the test cases that are marked as bug-inducing, and report the time elapsed before the first bug is revealed. For a baseline comparison, we compare PCR with the state-of-the-art approach MAPTest, which has been proved to significantly better than CovCon [15], Contege [16], and AutoConTest [17]. For each test case, we run MAPTest repeatedly until a bug is found or the time limit is reached. We additionally record the number of schedules of MAPTest and the number of schedules of PCR before the first bug is revealed. Schedule reduction is calculated with formula 3, which measures the ratio of schedules reduced by PCR (P.) compared to MAPTest (M.) if $Schedules_{M.}$ is larger than $Schedules_{P.}$; otherwise, formula 4 is used.

$$\frac{Schedules_{M.} - Schedules_{P.}}{Schedules_{M.}} \tag{3}$$

2. Given that MAP-coverage is shown to be positively correlated with the bug revealing capability [5]

3. We sort the test cases randomly and use the same order for both methods for comparison purposes.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3144480, IEEE Transactions on Software Engineering

TABLE 4: Results on comparison of PCR with MAPTest

ID	RQ1-CIP(M.)	PCR (10 times avg.)			MAPTest (10 times avg.)			RQ3-Comparison (P. over M.)			
	Freq. (runs)	Schedules	Success Rate	Time(ms)	Schedules	Success Rate	Time(ms)	Schedules Reduction(%)	p-value	Speedup	p-value
V1	**0.68**	5.0	100%	482.5	15.2	100%	1180.8	**67.1**	**0.006**	**2.4**	**0.006**
V2	**1.00**	6.4	100%	684.0	411.6	100%	7769.2	**98.4**	**0.005**	**11.4**	**0.032**
V3	0.00	5.0	100%	2685.9	35.1	100%	6230.2	**85.8**	**0.010**	**2.3**	**0.014**
V4	0.00	9.5	100%	3607.8	10	100%	3114.3	**5.0**	0.858	-1.2	0.067
V5	0.00	2.0	100%	325.7	4.8	100%	418.0	**58.3**	**0.021**	**2.3**	**0.011**
V6	0.00	2.8	100%	5953.3	1.0	100%	411.9	-64.3	0.003	-14.5	**0.006**
V7	0.00	2.0	100%	1174.0	31.4	100%	3202.1	**93.6**	**0.006**	**2.7**	**0.006**
V8	0.00	5.0	100%	3744.6	9.0	100%	3763.7	**44.4**	**0.051**	**1.0**	1.000
V9	0.00	1.0	100%	168.5	6.3	100%	645.5	**84.1**	**0.014**	**3.8**	**0.006**
V10	0.00	2.0	100%	241.2	14.0	100%	2083.1	**85.7**	**0.006**	**8.6**	**0.006**
V11	0.00	3.0	100%	1421.7	44.9	100%	4944.5	**93.3**	**0.004**	**3.5**	**0.006**
V12	0.00	11.0	100%	1060.4	149.0	100%	6997.6	**92.6**	**0.004**	**6.6**	**0.006**
V13	**0.26**	6.8	100%	2583.7	22.0	100%	3624.5	**69.1**	**0.007**	**1.4**	**0.008**
V14	0.00	2.0	100%	267.8	1.0	100%	348.7	-50.0	0.001	**1.3**	**0.006**
V15	**0.50**	3.0	100%	776.1	24.6	100%	2355.5	**87.8**	**0.004**	**3.0**	**0.006**
V16	0.00	4.9	100%	344.7	17.0	100%	1038.7	**71.2**	**0.006**	**3.0**	**0.006**
V17	0.00	2.0	100%	1235.2	86.2	100%	12328.7	**97.7**	**0.004**	**10.0**	**0.006**
V18	**0.72**	3.0	100%	418.5	21.7	100%	1458.2	**86.2**	**0.005**	**3.5**	**0.006**
V19	0.00	2.0	100%	211.1	20.4	100%	1056.0	**90.2**	**0.004**	**5.0**	**0.006**
V20	0.00	3.0	100%	5981.5	670.3	100%	67461.8	**99.6**	**0.004**	**11.3**	**0.006**

RQ1-CIP (M.) reports the number of times that MAPTest successfully matches the intended pattern among 50 independent runs; **Schedules** is the average number of schedules used before the first bug is revealed; **Success Rate** reports the number of times a bug is successfully revealed among 10 independent runs; **Time** is the average time (ms) spent before the first bug is revealed; **RQ3-Comparison(P. over M.)** shows the results of PCR compared to MAPTest on the percentage of **Schedule Reduction**, **Speedup** and statistical testing results of **p-value**. A negative number indicates that MAPTest performs better than PCR on the corresponding class and the corresponding metric.

TABLE 5: Bugs detection effectiveness comparison

ID	Bugs (10 times avg.)		Comparison (P. over M.)	
	PCR	MAPTest	Difference	p-value
V1	4.0	2.3	**1.7**	**0.008**
V2	5.0	0.5	**4.5**	**0.005**
V3	3.0	1.4	**1.6**	**0.005**
V4	13.0	20.1	-7.1	0.003
V5	6.6	4.4	**2.2**	**0.005**
V6	2.6	11.5	-8.9	0.005
V7	11.6	5.8	**5.8**	**0.006**
V8	4.2	1.4	**2.8**	**0.005**
V9	34.4	2.1	**32.3**	**0.005**
V10	4.0	1.3	**2.7**	**0.004**
V11	5.0	0.4	**4.6**	**0.005**
V12	3.0	0.5	**2.5**	**0.005**
V13	1.6	0.0	**1.6**	**0.005**
V14	3.0	2.0	**1.0**	**0.002**
V15	7.0	2.9	**4.1**	**0.005**
V16	2.0	1.7	**0.3**	0.233
V17	20.7	0.0	**20.7**	**0.005**
V18	11.0	3.3	**7.7**	**0.005**
V19	4.8	2.0	**2.8**	**0.005**
V20	1.0	0.1	**0.9**	**0.003**

$$-\frac{Schedules_{P.} - Schedules_{M.}}{Schedules_{P.}} \tag{4}$$

A negative number indicates that the schedules of PCR are larger than that of MAPTest. The Speedup is calculated with formula 5 if $Time_{M.}$ is larger than $Time_{P.}$; otherwise formula 6 is used.

$$\frac{Time_{M.}}{Time_{P.}} \tag{5}$$

$$-\frac{Time_{P.}}{Time_{M.}} \tag{6}$$

The Speedup measures the improvement in execution time by PCR (compared to MAPTest). A negative number indicates that MAPTest is faster than PCR in triggering the first bug.

To examine whether there is a statistically significant difference between the two methods, we use the Wilcoxon signed-rank test [18]. We set the significance level to be 0.05, which means if p-value is smaller than 0.05, we reject the null hypothesis, i.e., the difference between the two methods is statistically significant; otherwise, we accept the null hypothesis, which means the difference is not statistically significant.

The results are shown in Table 4. We can observe from the results that: (1) MAPTest needs more schedules (79.78 times on average) to find the first bug compare to PCR (4.07 times on average). PCR shows a schedule reduction on 18 out of 20 programs and a maximum reduction of 99.6% over MAPTest. (2) PCR is better than MAPTest on 18 programs in terms of execution time used to reveal the first bug and achieves the maximum speedup of 11.4 times. (3) Both MAPTest and PCR find concurrent bugs in all run. However, the results of PCR are consistently and significantly better than MAPTest, as indicated by the p-value.

As shown in Table 4, due to the small number of bugs in these programs, both PCR and MAPTest can achieve a 100% success rate. To further evaluate the effectiveness of PCR, we design a new experiment to see whether PCR can find more buggy traces than MAPTest given the same amount of time. As shown in Table 5, **Bugs (10 times avg.)** shows the average number of buggy traces found in 10 independent runs. Note that each run is set to be 10s. **Difference** shows the difference between the average number of buggy traces found by PCR and MAPTest. A negative number indicates that MAPTest can find more bugs than PCR. Based on this result, we observe that PCR can find more buggy traces than MAPTest in 18 programs, which is consistent with the results of Table 4. Moreover, the significant analysis results show that PCR is significantly better than MAPTest on 17 programs.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3144480, IEEE Transactions on Software Engineering

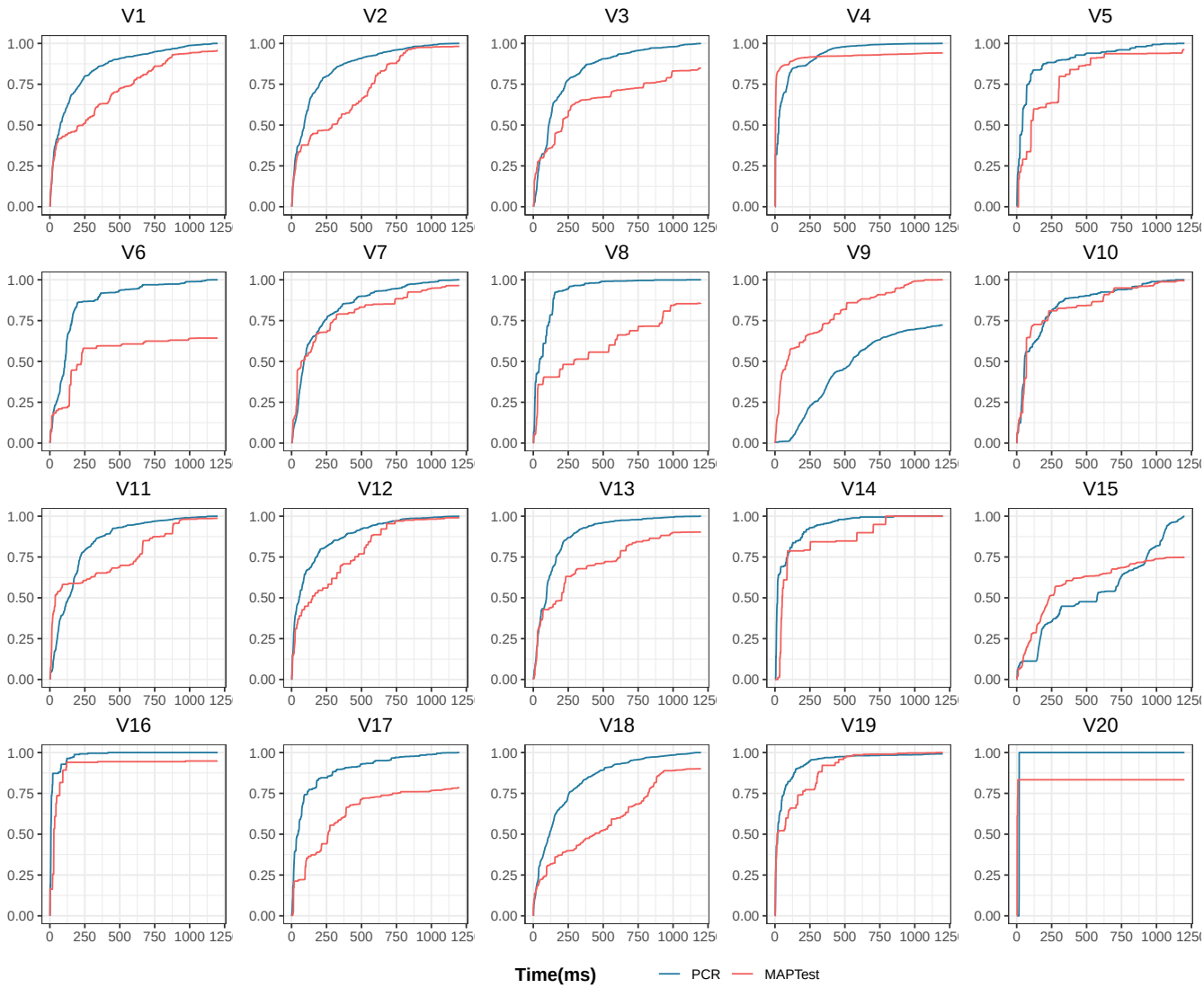


Fig. 6: MAP-coverage achieved over time

RQ4: Can PCR effectively reduce the efforts of constraint solving?

MCR [13] relies on applying constraint solving to generate failure-inducing schedules for concurrent programs to reveal concurrency bugs. The main idea is to require a read operation (on a shared variable) to obtain a different value compared to the original schedule. This rule can be represented by p_2 in Table 1. As MAP-coverage [5] has been shown to be positively correlated with the bug revealing capability (of a set of schedules), PCR aims at focusing on a much smaller set of thread interleavings (than those explored using MCR) that achieve high MAP-coverage. Meanwhile, PCR utilizes six heuristic rules to reduce redundant constraints to reduce constraint solving efforts. In this research question, we report the number of schedules, the number of constraints generated, and the time taken to expose the first bug by PCR and MCR, respectively. The aim is to show that PCR reduces the effort of constraining solving without sacrificing the capability of exposing bugs. We also report the schedule reduction, constraint reduction, and speedup of PCR over MCR. The calculation formulas are the same as those used in RQ3.

The evaluation results are shown in Table 6. From the experimental results, we can observe that the results on the number of schedules, constraints, and the execution time of PCR and MCR are relatively close to each other. In terms of the generated schedules, PCR shows a reduction in nine programs compared to MCR and a maximum reduction ratio of 90.6%. MCR shows a reduction in seven programs compared to PCR, with a maximum reduction ratio of 53.1%. In terms of the generation constraints, PCR shows a reduction in ten programs compared to MCR and a maximum reduction ratio of 94.9%. MCR shows the reduction in eight programs compared to PCR, with a maximum reduction ratio of 73.2%. In general, PCR performs better than MCR on 10 programs and achieved a maximum speedup of 3.4 times. For V17, i.e., the AppenderAttachableImpl class, PCR achieves high schedule reduction and constraint reduction as compared to MCR. We further analyze the experimental results. In all ten independent runs, PCR successfully generates a schedule from the constraints synthesized based on p_4 in Table 1 with the initial execution trace, and triggers the bug. However, MCR was unable to synthesize such constraints and does not generate the schedule that triggers the bug in the initial execution trace, which leads to a large number of constraints to be generated.

TABLE 6: Results on comparison of PCR with MCR

ID	PCR (10 times avg.)			MCR (10 time avg.)			Comparison (PCR over MCR)		
	Schedule	Constraint	Time(ms)	Schedule	Constraint	Time(ms)	Schedule Reduction(%)	Constraint Reduction(%)	Speedup
V1	5.0	15.2	482.5	3.0	5.0	758.0	-40.0	-67.1	**1.6**
V2	6.4	29.9	684.0	3.0	8.0	126.0	-53.1	-73.2	-5.4
V3	5.0	16.1	1543.8	11.6	70.5	2321.5	**56.9**	**77.2**	**1.5**
V4	9.5	20	3607.8	4.3	14.5	854.1	-54.7	-27.5	-4.2
V5	2.0	5.2	325.7	5.0	12.9	705.2	**60.0**	**59.7**	**2.2**
V6	2.8	36.9	5953.3	6.8	24.8	3863.7	**58.8**	-32.8	-1.5
V7	2.0	50.5	1174.0	21.3	192.1	3967.3	**90.6**	**73.7**	**3.4**
V8	5.0	24.0	3744.6	4.7	39.6	2078.8	-6.0	**39.4**	-1.8
V9	1.0	1.0	168.5	1.0	1.1	528.8	0.0	0.0	**3.1**
V10	2.0	3.0	241.2	3.0	14.0	398.7	**33.3**	**78.6**	**1.7**
V11	3.0	9.1	1421.7	4.0	5.0	559.6	**25.0**	-45.1	-2.5
V12	11.0	41.9	1060.4	7	12	520.3	-36.4	-71.4	-2.0
V13	6.8	26.0	2583.7	5.8	23.0	269.8	**14.7**	-11.5	-9.6
V14	2.0	2.0	267.8	2.0	2.0	682.3	0.0	0.0	**2.5**
V15	3.0	8.2	776.1	3.0	17.9	196.0	0.0	**54.2**	-4.0
V16	4.9	14.3	344.7	3.1	5.1	239.7	-36.7	-64.3	-1.4
V17	2.0	9.0	1235.2	8.2	176.0	2634.5	**75.6**	**94.9**	**2.1**
V18	3.0	8.0	418.5	2.8	19.4	523	-6.7	**58.8**	**1.2**
V19	2.0	5.5	211.1	3.3	6.4	257.1	**39.4**	**14.1**	**1.2**
V20	3.0	8.0	5981.5	3.0	11.0	5018.8	0.0	**27.3**	-1.2

Schedule is the average number of schedules generated for a given test case to trigger the first bug. **Constraint** shows the average number of constraint solving processes required. **Time** shows the average time required before the first bug is found. The last three columns show how much PCR has reduced on Schedules and Constraints compared to MCR, and how much speedup PCR achieves.

Note that the constraints generated by MCR require the read operation to read a different value, which always happens following a write operation. Therefore, MCR can be considered to synthesize constraints based on pattern p_2 and partially p_7 in Table 1 and the results of MCR are very close to PCR. In general, PCR performs better than MCR.

5.4 Threats to Validity

First, the experiment is conducted with a test suite of 20 Java thread-safe classes, which are widely adopted by existing related approaches [5], [15], [16]. The evaluation results show the effectiveness of our approach. The results may be a bit conservative due to the test suite adopted for evaluation. However, we follow existing related approaches [5] for the test suite selection. More thread-safe classes are desirable to further evaluate our approach.

Second, the test case used for each thread-safe class may also affect the evaluation results. Due to the fact that there are no available test cases for the thread-safe classes adopted for our evaluation, we adopt MAPTest to generate 1,000 test cases for each thread-safe class and use those test cases for both methods for fairness. The automatically generated test cases may be different from manually designed test cases. High-quality, manual crafted test cases could help better evaluate the methods on different experiment settings.

6 Related Work

Our study is closely related to studies on concurrent bug detection, in particular, on concurrent program test case generation and testing schedule exploration.

6.1 Test Case Generation

Since writing test case manually is time-consuming and laborious, there have been various methods for test case generation. One way is generating new test cases by randomly generating parameters and calling methods through random seeds, such as Contege [16]. However, random test case generation approaches do not provide termination criteria. A widely studied approach is the coverage-guided generation approach. Those approaches rely on coverage criteria, which are used to measure the adequacy of testing and provide guidance for generating test cases [19], [20].

For instance, method-pair coverage [15] matches the combination of all method pairs in the test class and extracts method pairs that have not been covered or rarely covered by analyzing the execution trace of test cases. AutoConTest [17] collects context information, dynamically and iteratively calculates coverage requirements, and generates new test cases based on sequential coverage. Maple [21] uses thread interleaving idioms to define coverage metrics. Heuristics are proposed to predict untested interleavings based on the recorded interleavings. HaPSet [22] utilizes ordering constraints from good test runs as a metric to cover important concurrency scenarios, and guide the testing of the program by gathering and analyzing the ordering constraints in the program. ConSuite [23] uses thread interleaving of test cases and generates test cases to cover the specific thread interleaving that is not covered. Similar studies include Narada [24], Intruder [25] and Omen [26], etc. Def-use pair coverage [27] was built based on alldu-path coverage. Kena *et al.* [28] design a method to generate new coverage criteria based on existing dynamic and static analysis methods such as Eraser [29], and GoldiLocks [30], and expand multiple existing concurrent test coverage criteria, e.g., ConcurPairs, definition-use coverage, and synchronization pair coverage. Recently, researchers have proposed some coverage criteria for concurrency programs and corresponding test case generation methods. MAP-coverage [5] is the latest development. MAP-coverage uses a memory-access pattern to abstract the thread interleaving in execution trace, and design MAPTest to preferentially select patterns that have not been covered or rarely covered (guided by MAP-coverage) to generate new test cases.

Different from the previous studies, MAPTest tries to achieve

higher MAP-coverage by generating new test cases. PCR generates new schedules to increase the coverage of a single test case. Note that PCR can also be combined with existing test case generation tools.

6.2 Schedule Space Exploration

Our work is related to the work of detecting concurrency bugs [31], [32], [33]. Research on the detection of concurrency bugs usually focuses on improving the efficiency and effectiveness of exploring the state space. Commonly used exploration approaches include completely random, heuristic-guided or thorough exploration. In concurrency bug detection, the most common way is random testing. However, random testing is limited by redundant exploration, which increases the overhead of testing. Some previous studies [34], [35], [36], [37] try to optimize random scheduler to detect concurrency bug deterministically. For instance, PCT [38] uses a disciplined schedule-randomization technique to force the program along with buggy schedules. CHESS [39] controls thread scheduling and uses effective search technology to drive programs through possible thread interleavings. Heuristic-guided methods usually use heuristic algorithms to guide the execution of test cases or reduce the state space of the test case. Commonly used approaches include Happens-before analysis [40], [41] and lockset algorithms [42], [43], [44]. RaceChecker [40] uses happens-before relation to prune infeasible races that are reported by imprecise detectors. Eraser [29] uses a lockset algorithm to detect concurrency bugs by monitoring every shared memory reference and verifying locking behaviors. There are some studies [42], [44], [43], [45] that try to further improve the lock set algorithm to reduce overhead, and others [46], [32], [30], [47], [48] try to combine lockset algorithm with happens-before analysis. However, these methods often suffer from false positives, since the static analysis cannot capture runtime information to determine the happens-before and alias information correctly.

Based on trace theory [49], many dynamic partial-order reduction methods have been proposed to mitigate the state explosion program in multi-threaded programs [50], [51], [52], [53], [54]. Optimal-DPOR [51] defines a novel class of sets, called source sets, to make each exploration is the minimal number of executions. Nguyen *et al.* [53] proved that an optimal DPOR exploration is NP-complete, and proposed a hybrid approach Quasi-Optimal POR (QPOR) to turn a non-optimal DPOR into an optimal one. Aronis *et al.* [54] enhance DPOR by introducing the concept of observability to refine which operations are considered as interference. In addition, symbolic execution tests concurrent programs by simultaneously exploring multiple paths that a program could take under different inputs [55]. For instance, KLEE [56] utilizes various of constraints to design search heuristics to obtain high code coverage to test complex system programs. Based on KLEE, Schemmel *et al.* [57] combines symbolic execution and QPOR [53] to systematic test multi-threaded programs. Moreover, symbolic execution can be combined with concrete execution to form the concept of concolic testing [58]. Sen *et al.* [59] proposed two automated concolic testing tools, namely CUTE for C and jCUTE for Java. Farzan *et al.* [50] proposed a systematic testing method for concurrent programs called con2colic testing, which derives test inputs and schedules to systematically explore the execution space of concurrent programs.

The false positives of static analysis approaches can be alleviated by introducing dynamic execution and constraint solving. In general, constraints are constructed based on dynamic executions, and then constraint solving is used to generate new schedules to explore the state space. ConcBugAssist [60] proposes a constraint-based method for diagnosing concurrency bugs and presents a constraint-based method to compute repairs. Mahmoud Said *et al.* [61] propose a symbolic method based on SMT to provide guidance for the debugging of a data race. Zheng *et al.* [62] propose to use race manifestation to characterize the behavior of concurrency bugs, thereby constructing constraints to solve new schedules, and finally triggering data race in the program. Jeff *et al.* [13] propose MCR, which uses the existing execution trace to construct new read and write constraints, and tries to make read operations read different values from write operations to explore the state space of concurrent programs. Similar work include [63], [64], [65], [66], [67], [68], which are also based on constraint solving. Our approach proposes new constraints based on memory-access patterns, adopts constraint solving to generate new schedules, with the purpose of achieving higher MAP-coverage.

In addition, our work also related to the work of software system testing, such as compiler testing [69], [70], [71], [72], deep learning library testing [73], [74], etc.

7 Conclusion

Testing concurrency bugs is notoriously challenging due to its huge state space and non-determinism, therefore how to effectively test them with improved test coverage is an important research problem. In this work, we propose a novel approach PCR to maximize MAP-coverage of a thread-safe class based on constraint solving. We construct constraints based on uncovered memory-access patterns in an execution trace, and use SMT-solver to compute new schedules to explore the state space of concurrent programs. To avoid the repeated solution of infeasible constraints, we use interpolation to get the unsat-core constraints, and filter out the solutions containing these constraints in the next generation. Moreover, our approach can also be combined with existing test case generation tools to further optimize concurrent testing. We have implemented our method and evaluated it on a set of Java programs. Experimental results show that our method can obtain higher MAP-coverage and expose concurrency bugs faster than the state-of-the-art method MAPTest.

Acknowledgment

This work is partially funded by projects 61872263, U1836214, 61802275, 62002256 from National Natural Science Foundation of China, and the Singapore Ministry of Education Research Found, grant number: MOE2016-T2-2-123.

References

[1] F. A. Bianchi, A. Margara, and M. Pezzè, "A survey of recent trends in testing concurrent software systems," *IEEE Trans. Software Eng.*, vol. 44, no. 8, pp. 747–783, 2018. [Online]. Available: https://doi.org/10.1109/TSE.2017.2707089

[2] J. Huang and C. Zhang, "Debugging concurrent software: Advances and challenges," *J. Comput. Sci. Technol.*, vol. 31, no. 5, pp. 861–868, 2016. [Online]. Available: https://doi.org/10.1007/s11390-016-1669-8

[3] Y. Cai, B. Zhu, R. Meng, H. Yun, L. He, P. Su, and B. Liang, "Detecting concurrency memory corruption vulnerabilities," ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 706–717. [Online]. Available: https://doi.org/10.1145/3338906.3338927

[4] S. Park, R. W. Vuduc, and M. J. Harrold, "UNICORN: a unified approach for localizing non-deadlock concurrency bugs," *Softw. Test., Verif. Reliab.*, vol. 25, no. 3, pp. 167–190, 2015. [Online]. Available: https://doi.org/10.1002/stvr.1523

[5] Z. Wang, Y. Zhao, S. Liu, J. Sun, X. Chen, and H. Lin, "Map-coverage: A novel coverage criterion for testing thread-safe classes," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 722–734.

[6] H. Lin, Z. Wang, S. Liu, J. Sun, D. Zhang, and G. Wei, "Pfix: Fixing concurrency bugs based on memory access patterns," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: ACM, 2018, pp. 589–600. [Online]. Available: http://doi.acm.org/10.1145/3238147.3238198

[7] E. Noonan, E. Mercer, and N. Rungta, "Vector-clock based partial order reduction for jpf," *ACM SIGSOFT Software Engineering Notes*, vol. 39, no. 1, pp. 1–5, 2014.

[8] J. Huang, "Stateless model checking concurrent programs with maximal causality reduction," in *ACM SIGPLAN Notices*, vol. 50, no. 6. ACM, 2015, pp. 165–174.

[9] B. Goetz, T. Peierls, J. J. Bloch, J. Bowbeer, D. Holmes, and D. Lea, *Java Concurrency in Practice*. Addison-Wesley, 2006.

[10] "Log4j1." http://logging.apache.org/log4j/1.2/index.html.

[11] M. Vaziri, F. Tip, and J. 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*, 2006, pp. 334–345.

[12] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, p. 337–340.

[13] J. Huang, "Stateless model checking concurrent programs with maximal causality reduction," ser. PLDI '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 165–174. [Online]. Available: https://doi.org/10.1145/2737924.2737975

[14] "ASM bytecode manipulation and analysis framework." https://asm.ow2.io.

[15] A. Choudhary, S. Lu, and M. 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*, 2017, pp. 266–277.

[16] M. Pradel and T. 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*, 2012, pp. 521–530. [Online]. Available: https://doi.org/10.1145/2254064.2254126

[17] V. Terragni and S. 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*, 2016, pp. 1121–1132. [Online]. Available: https://doi.org/10.1145/2884781.2884876

[18] F. Wilcoxon, *Individual Comparisons by Ranking Methods*. New York, NY: Springer New York, 1992, pp. 196–202. [Online]. Available: https://doi.org/10.1007/978-1-4612-4380-9_16

[19] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov, "Guidelines for coverage-based comparisons of non-adequate test suites," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 4, pp. 22:1–22:33, 2015. [Online]. Available: https://doi.org/10.1145/2660767

[20] M. Staats, M. W. Whalen, A. Rajan, and M. P. E. 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*, 2010, pp. 161–170.

[21] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, "Maple: A coverage-driven testing tool for multithreaded programs," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 485–502. [Online]. Available: https://doi.org/10.1145/2384616.2384651

[22] C. Wang, M. Said, and A. Gupta, "Coverage guided systematic concurrency testing," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, 2011, pp. 221–230. [Online]. Available: https://doi.org/10.1145/1985793.1985824

[23] S. Steenbuck and G. 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*, 2013, pp. 144–153. [Online]. Available: https://doi.org/10.1109/ICST.2013.33

[24] M. Samak, M. K. Ramanathan, and S. Jagannathan, "Synthesizing racy tests," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 175–185. [Online]. Available: https://doi.org/10.1145/2737924.2737998

[25] M. Samak and R. M. Krishna, "Synthesizing tests for detecting atomicity violations," ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 131–142. [Online]. Available: https://doi.org/10.1145/2786805.2786874

[26] M. Samak and M. K. Ramanathan, "Multithreaded test synthesis for deadlock detection," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 473–489. [Online]. Available: https://doi.org/10.1145/2660193.2660238

[27] C. D. Yang, A. L. Souter, and L. 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*, 1998, pp. 153–162. [Online]. Available: https://doi.org/10.1145/271771.271804

[28] B. Krena, Z. Letko, and T. 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*, 2011, pp. 177–192. [Online]. Available: https://doi.org/10.1007/978-3-642-29860-8_14

[29] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Transactions on Computer Systems (TOCS)*, vol. 15, no. 4, pp. 391–411, 1997.

[30] T. Elmas, S. Qadeer, and S. Tasiran, "Goldilocks: a race and transaction-aware Java runtime," in *ACM SIGPLAN Notices*, vol. 42, no. 6. ACM, 2007, pp. 245–255.

[31] C. Flanagan and S. N. Freund, "FastTrack: efficient and precise dynamic race detection," in *ACM Sigplan Notices*, vol. 44, no. 6. ACM, 2009, pp. 121–133.

[32] C. von Praun and T. R. Gross, "Static conflict analysis for multi-threaded object-oriented programs," in *ACM Sigplan Notices*, vol. 38, no. 5. ACM, 2003, pp. 115–128.

[33] S. Liu, G. Bai, J. Sun, and J. Song Dong, "Towards using concurrent java api correctly," in *2016 21st International Conference on Engineering of Complex Computer Systems (ICECCS)*, 11 2016, pp. 219–222.

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

[35] C.-S. Park and K. Sen, "Randomized active atomicity violation detection in concurrent programs," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 2008, pp. 135–145.

[36] K. Sen, "Race directed random testing of concurrent programs," *ACM Sigplan Notices*, vol. 43, no. 6, pp. 11–21, 2008.

[37] P. Joshi, C.-S. Park, K. Sen, and M. Naik, "A randomized dynamic program analysis technique for detecting real deadlocks," in *ACM Sigplan Notices*, vol. 44, no. 6. ACM, 2009, pp. 110–120.

[38] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "A randomized scheduler with probabilistic guarantees of finding bugs," in *ACM Sigplan Notices*, vol. 45, no. 3. ACM, 2010, pp. 167–178.

[39] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs." in *OSDI*, vol. 8, 2008, pp. 267–280.

[40] K. Lu, Z. Wu, X. Wang, C. Chen, and X. Zhou, "Racechecker: efficient identification of harmful data races," in *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*. IEEE, 2015, pp. 78–85.

[41] D. Perkovic and P. J. Keleher, "Online data-race detection via coherency guarantees," in *OSDI*, vol. 96, 1996, pp. 47–57.

[42] D. Engler and K. Ashcraft, "RacerX: effective, static detection of race conditions and deadlocks," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 237–252.

[43] H. Nishiyama, "Detecting data races using dynamic escape analysis based on read barrier." in *Virtual Machine Research and Technology Symposium*, 2004, pp. 127–138.

[44] T. Elmas, S. Qadeer, and S. Tasiran, "Precise race detection and efficient model checking using locksets," Microsoft Tech Report, Tech. Rep., 2006.

[45] C. von Praun and T. R. Gross, "Object race detection," in *Acm Sigplan Notices*, vol. 36, no. 11. ACM, 2001, pp. 70–82.

[46] Y. Yu, T. Rodeheffer, and W. Chen, "Racetrack: efficient detection of data race conditions via adaptive tracking," in *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5. ACM, 2005, pp. 221–234.

[47] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan, "Efficient and precise datarace detection for multithreaded object-oriented programs," *ACM Sigplan Notices*, vol. 37, no. 5, pp. 258–269, 2002.

[48] E. Pozniansky and A. Schuster, "Multirace: efficient on-the-fly data race detection in multithreaded C++ programs," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 3, pp. 327–340, 2007.

[49] A. W. Mazurkiewicz, "Trace theory," in *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, Germany, 8-19 September 1986*, ser. Lecture Notes in Computer Science, W. Brauer, W. Reisig, and G. Rozenberg, Eds., vol. 255. Springer, 1986, pp. 279–324. [Online]. Available: https://doi.org/10.1007/3-540-17906-2_30

[50] A. Farzan, A. Holzer, N. Razavi, and H. Veith, "Con2colic testing," in *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, B. Meyer, L. Baresi, and M. Mezini, Eds. ACM, 2013, pp. 37–47. [Online]. Available: https://doi.org/10.1145/2491411.2491453

[51] P. A. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas, "Optimal dynamic partial order reduction," in *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, S. Jagannathan and P. Sewell, Eds. ACM, 2014, pp. 373–384. [Online]. Available: https://doi.org/10.1145/2535838.2535845

[52] E. Albert, P. Arenas, M. G. de la Banda, M. Gómez-Zamalloa, and P. J. Stuckey, "Context-sensitive dynamic partial order reduction," in *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, ser. Lecture Notes in Computer Science, R. Majumdar and V. Kuncak, Eds., vol. 10426. Springer, 2017, pp. 526–543. [Online]. Available: https://doi.org/10.1007/978-3-319-63387-9_26

[53] H. T. T. Nguyen, C. Rodríguez, M. Sousa, C. Coti, and L. Petrucci, "Quasi-optimal partial order reduction," in *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, ser. Lecture Notes in Computer Science, H. Chockler and G. Weissenbacher, Eds., vol. 10982. Springer, 2018, pp. 354–371.

[54] S. Aronis, B. Jonsson, M. Lång, and K. Sagonas, "Optimal dynamic partial order reduction with observers," in *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II*, ser. Lecture Notes in Computer Science, D. Beyer and M. Huisman, Eds., vol. 10806. Springer, 2018, pp. 229–248.

[55] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 50:1–50:39, 2018. [Online]. Available: https://doi.org/10.1145/3182657

[56] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, R. Draves and R. van Renesse, Eds. USENIX Association, 2008, pp. 209–224. [Online]. Available: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[57] D. Schemmel, J. Büning, C. Rodríguez, D. Laprell, and K. Wehrle, "Symbolic partial-order execution for testing multi-threaded programs," in *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, ser. Lecture Notes in Computer Science, S. K. Lahiri and C. Wang, Eds., vol. 12224. Springer, 2020, pp. 376–400. [Online]. Available: https://doi.org/10.1007/978-3-030-53288-8_18

[58] A. Farzan, A. Holzer, N. Razavi, and H. Veith, "Concolic testing of concurrent programs," in *Software Engineering 2014, Fachtagung des GI-Fachbereichs Softwaretechnik, 25. Februar - 28. Februar 2014, Kiel, Germany*, ser. LNI, W. Hasselbring and N. C. Ehmke, Eds., vol. P-227. GI, 2014, pp. 101–102. [Online]. Available: https://dl.gi.de/20.500.12116/30982

[59] K. Sen and G. Agha, "Cute and jcute: Concolic unit testing and explicit path model-checking tools," in *CAV*, T. Ball and R. B. Jones, Eds., 2006, pp. 419–423.

[60] S. Khoshnood, M. Kusano, and C. Wang, "Concbugassist: Constraint solving for diagnosis and repair of concurrency bugs," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 165–176. [Online]. Available: https://doi.org/10.1145/2771783.2771798

[61] M. Said, C. Wang, Z. Yang, and K. Sakallah, "Generating data race witnesses by an smt-based analysis," ser. NFM'11. Berlin, Heidelberg: Springer-Verlag, 2011, p. 313–327.

[62] L. Zheng, X. Liao, H. Jin, B. He, J. Xue, and H. Liu, "Towards concurrency race debugging: An integrated approach for constraint solving and dynamic slicing," in *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3243176.3243206

[63] J. Huang, C. Zhang, and J. Dolby, "Clap: Recording local executions to reproduce concurrency failures," vol. 48, no. 6, p. 141–152, Jun. 2013. [Online]. Available: https://doi.org/10.1145/2499370.2462167

[64] N. Machado, D. Quinta, B. Lucia, and L. Rodrigues, "Concurrency debugging with differential schedule projections," *ACM Trans. Softw. Eng. Methodol.*, vol. 25, no. 2, Apr. 2016. [Online]. Available: https://doi.org/10.1145/2885495

[65] J. Huang, "Ufo: Predictive concurrency use-after-free detection," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 609–619. [Online]. Available: https://doi.org/10.1145/3180155.3180225

[66] C. Castellini, E. Giunchiglia, and A. Tacchella, "Sat-based planning in complex domains: Concurrency, constraints and nondeterminism," *Artificial Intelligence*, vol. 147, pp. 85–117, 07 2003.

[67] Z. Wang, H. Wang, S. Liu, J. Sun, H. Wang, and J. Chen, "IFIX: fixing concurrency bugs while they are introduced," in *25th International Conference on Engineering of Complex Computer Systems, ICECCS 2020, Singapore, October 28-31, 2020*, Y. Li and A. W. Liew, Eds. IEEE, 2020, pp. 155–164. [Online]. Available: https://doi.org/10.1109/ICECCS51672.2020.00025

[68] H. Wang, Z. Wang, J. Sun, S. Liu, A. Sadiq, and Y. Li, "Towards generating thread-safe classes automatically," in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 2020, pp. 943–955. [Online]. Available: https://doi.org/10.1145/3324884.3416625

[69] Y. Zhao, Z. Wang, J. Chen, M. Liu, M. Wu, Y. Zhang, and L. Zhang, "History-driven test program synthesis for JVM testing," in *44th IEEE/ACM International Conference on Software Engineering, ICSE 2022*, to appear.

[70] J. Chen, J. Patra, M. Pradel, Y. Xiong, H. Zhang, D. Hao, and L. Zhang, "A survey of compiler testing," *ACM Comput. Surv.*, vol. 53, no. 1, pp. 4:1–4:36, 2020. [Online]. Available: https://doi.org/10.1145/3363562

[71] J. Chen, G. Wang, D. Hao, Y. Xiong, H. Zhang, and L. Zhang, "History-guided configuration diversification for compiler test-program generation," in *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 2019, pp. 305–316. [Online]. Available: https://doi.org/10.1109/ASE.2019.00037

[72] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, and B. Xie, "Learning to prioritize test programs for compiler testing," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, S. Uchitel, A. Orso, and M. P. Robillard, Eds. IEEE / ACM, 2017, pp. 700–711. [Online]. Available: https://doi.org/10.1109/ICSE.2017.70

[73] Z. Wang, M. Yan, J. Chen, S. Liu, and D. Zhang, "Deep learning library testing via effective model generation," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, P. Devanbu, M. B. Cohen, and T. Zimmermann, Eds. ACM, 2020, pp. 788–799. [Online]. Available: https://doi.org/10.1145/3368089.3409761

[74] Z. Wang, H. You, J. Chen, Y. Zhang, X. Dong, and W. Zhang, "Prioritizing test inputs for deep neural networks via mutation analysis," in *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 2021, pp. 397–409. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00046