A Splitting Strategy for Testing Concurrent Programs

Xiaofang Qi, Huayang Zhou
School of Computer Science and Engineering
Southeast University
Nanjing, China
xfqi@seu.edu.cn, hbfxzhy@163.com

***Abstract*—Reachability testing is an important approach to testing concurrent programs. It generates and exercises every partially ordered synchronization sequence automatically and on-the-fly without constructing a static model and saving any test history. However, test sequences generated by existing reachability testing are ineffective in detecting concurrency faults involved in access anomalies, such as accessing shared variables without proper protections or synchronizations, incorrectly shrinking or shifting critical regions, etc. In this paper, we present a splitting strategy as well as a prototype tool called SplitRichTest for revealing such concurrency faults. SplitRichTest adopts the framework of conventional reachability testing while implementing our splitting strategy. By splitting relevant codes, SplitRichTest generates and exercises fine-grained synchronization sequences. The key ingredient of SplitRichTest is an efficient heuristic algorithm to select and sort candidate splitting points. We conducted an empirical study on representative concurrent Java programs and evaluated the effectiveness of SplitRichTest using mutation testing. Experimental results show that our splitting strategy facilitates generating more fine-grained synchronization sequences and significantly improves concurrency fault detection capability of reachability testing.**

***Index Terms*—Software testing, concurrency testing, reachability testing, splitting strategy**

I. Introduction

Concurrent programs are becoming increasingly important and widespread with the advent of multi-core techniques and the support for concurrent programming in modern programming languages [1]. While computational efficiency and resource utilization are improved significantly, concurrent programs often exhibit non-deterministic behaviors. Multiple executions of concurrent programs may select different sequences of synchronization events and produce different outcomes, making them difficult to test [2-3].

One approach to dealing with nondeterministic behaviors is stress testing, which executes a concurrent program with the same input many times in order that different interleavings would be chosen to expose more faults [2]. It is easy to carry out, but it is inefficient. Some interleavings may be executed many times while others may never be exercised at all. An alternative approach is random testing, which increases the probability of running different interleavings by injecting artificial delays during testing concurrent programs [2, 4]. Unfortunately, this approach is still not capable of preventing the same interleavings from being executed again and again.

To perform more diverse interleavings than stress or random testing for concurrent systems, researchers have developed systematic testing, in which program executions are controlled to perform approximately all possible different interleavings by exploring state spaces of concurrent programs [5]. Since an interleaving is a totally-ordered sequence of concurrent events, it is possible that two or more different interleavings have the same partial ordering of events. Exhaustively exploring state spaces of concurrent programs may incur state explosions, which substantially limit the applications of systematic testing.

Reachability testing, which systematically executes every partially ordered sequence of concurrent events exactly once, generates test sequences automatically and on-the-fly without saving any test history [6-7]. As it avoids different interleavings with the same partial ordering of concurrent events to be executed repeatedly, it significantly enhances test efficiency. However, during reachability testing, test sequences are generated and performed on the level of synchronized method, e.g. message passing or synchronized method invoking. Non-synchronized common method invoking is completely ignored when deriving test sequences. If a method or code block containing shared data accesses should be, but is not properly protected or synchronized, access anomalies (e.g. data race, atomicity violation, and atomic-set violations) or other concurrency bugs may arise. As a result, reachability testing is not capable of detecting such concurrency bugs effectively.

Generally, the effectiveness of reachability testing is evaluated by mutation testing, which allows creating mutants (artificial faults) as a substitute for real faults since real faults in a program are always unknown a priori [8-9]. During mutation testing, various mutation operators (syntactic variations) are used to generate mutants. If a test suite distinguishes the program under test from a mutant, it kills the mutant. A test suite with a higher mutation score (the ratio of the number of killed mutants to the number of all non-equivalent mutants) indicates a stronger fault detection capability. Test sequences generated during reachability testing kill almost all sequential mutants, which are created by applying sequential mutation operators [7]. However, recent research shows that test sequences that achieve a high mutation score for sequential mutants do not imply a high mutation score for concurrent mutants, which are generated by applying concurrent mutation operators [8]. Empirical results in this paper (Section 5.2) further demonstrate that reachability testing is ineffective in killing concurrent mutants, e.g. accessing shared variables without proper protections or synchronizations, incorrectly shrinking or shifting critical regions, etc.

978-1-7281-0591-8/19/$31.00 © 2019 IEEE

In this paper, we present a splitting strategy for reachability testing to improve its concurrency fault detection capability. With splitting relevant codes, more fine-grained synchronization sequences are generated and exercised to kill more concurrent mutants. We have developed a prototype tool called SplitRichTest, which adopts the framework of conventional reachability testing and implements our splitting strategy. We conducted an empirical study on representative concurrent Java programs and evaluated the effectiveness and performance of SplitRichTest. Results shows that our SplitRichTest significantly enhances concurrency fault detection capability for reachability testing. Though its performance is decreased due to running more test sequences, the execution time taken by SplitRichTest is still acceptable for most cases. Our study makes the following contributions:

- We present a splitting strategy for reachability testing to make it effective in detecting concurrency faults. Using our splitting strategy, relevant codes are split to generate fine-grained synchronization sequences, which substantially facilitate exposing more concurrency faults.
- We propose an efficient algorithm to select and sort splitting points for our splitting strategy. Experimental results show that SplitRichTest detects 68.4% of mutants that are not detected by reachability testing at the first splitting points, and 94.9% at the first three splitting points.
- We present a prototype tool SplitRichTest, which adopts the framework of conventional reachability testing and implements our splitting strategy.
- We evaluate the effectiveness and the performance of SplitRichTest on concurrent Java programs, as compared with conventional reachability testing tool RichTest and a comprehensive state-of-the-art concurrent bug detection tool Pecan [6, 9].

The remainder of this paper is organized as follows. Section 2 provides an overview of reachability testing. Section 3 describes our splitting strategy in details. Section 4 discusses the implementation of our tool SplitRichTest. Section 5 reports experimental results. Section 6 reviews related work. Section 7 concludes this paper and provides the guidelines of the future work.

II. OVERVIW OF REACHABILITY TESTING

Reachability testing uses a general execution model that allows it to be applied to several commonly used synchronization constructs, e.g. asynchronous and synchronous message passing, semaphores, and monitors [6]. Due to space constraints, we illustrate the reachability testing process with a monitor-based program. In this section, we first review basic concepts, namely SYN-sequence, race, and race variant, then illustrate the reachability testing process [6-7].

Fig.1 shows a classical concurrent program EvenNumGenerator. The program consists of two threads, which interact with each other by accessing a monitor *m*. When a thread t_1 calls a synchronized method of *m*, i.e., *evenIncrement()*, a monitor *call* event *s* occurs on t_1, and we refer to the calling as a *sending event*. When t_1 finally enters *m*, a monitor *entry* event *r* occurs on *m* and starts to execute inside *m*, and we refer to the entry as a *receiving event*. When the called monitor method is entered into the monitor, we say that the *sending event s* is synchronized with the receiving event *r* and <*s*, *r*> is a synchronization pair. *s* is the sending part of *r,* and *r* is the receiving part of *s*. For any method defined in a monitor, a sending event *s* can be synchronized with a receiving event *r* if the called monitor of *s* is the same as the owning monitor of *r*.

```
1.    class Generator {
2.        private int i = 0;
3.        public synchronized void checkValue() {
3'            public void checkValue() {
4.            if ( i % 2 != 0){
5.                System.err.println("Fail!");
6.                return;
7.            }
8.            System.out.println("Pass!");
9.        }
10.       public synchronized void evenIncrement()
      {
10'           public void evenIncrement() {
11.           i++;
12.           i++;
13.       }
14.   }
15.   public class EvenNumTest{
16.       public static void main(String args[]){
17.           Generator m = new Generator();
18.           Thread t1 = new Thread(){
19.               public void run(){
20.                   m.evenIncrement();
21.               }
22.           };
23.           Thread t2 = new Thread(){
24.               public void run(){
25.                   m.checkValue();
26.               }
27.           };
28.           t1.start();
29.           t2.start();
30.       }
```

Fig.1. An example program EvenNumGenerator

Given an execution of a concurrent program, a *SYN-sequence* (short for synchronization sequence) Q is defined as a tuple (Q_1, Q_2, …, Q_n, Λ), where Q_i is a totally ordered sequence of sending and receiving events that occurred on a thread or a synchronization object, and Λ is the set of synchronization pairs exercised in the execution. A SYN-sequence is usually represented as a space-time diagram. In this diagram, a vertical line represents a thread or monitor, and a horizontal solid arrow from a sending event to a receiving event represents a synchronization pair between them. Fig.2 shows two SYN-sequences, namely Q_0 and Q_1. The two sending events s_1 and s_2 represent calling *m.evenIncrement()* and *m.checkValue()* respectively while the two receiving events r_1 and r_2 represent the first and second entry event into *m* respectively. In Q_0, s_1 is first synchronized with r_1 (thread t_1 calls *m.evenIncrement()*and enter *m*), then s_2 is synchronized with r_2 (thread t_2 calls *m.checkValue()* and enters *m*). Similarly, in Q_1, s_2 is first synchronized with r_1 (thread t_2 first calls *m.checkValue()* and enters *m*), then s_1 is

synchronized with r_2 (thread t_1 calls *m.evenIncrement()*and enters *m*).

Let *s* be a sending event, *r* be a receiving event. *s* is synchronized with *r* in a SYN-sequence *Q*. Let *s′* be another sending event in *Q*. If *s′* could be synchronized with *r* in a different execution *Q′*, we say there exists a *lead race* between *s* and *s′* with respect to *r*. Note that all the events that happen before *s′* or *r* in *Q*, and the synchronizations between these events are the same as in *Q′*[6]. This condition guarantees that *s′* and *r* will be executed in *Q′* since only those events that happen before *s′* or *r* in *Q* may potentially affect the feasibility of *s′* or *r*. If these events and the synchronizations between them are not changed, *s′* and *r* will certainly be executed. The *race set* of *r* in *Q*, denoted as *race_set*(*r*, *Q*), is the set of sending events in *Q* that have a lead race with *s* w.r.t. *r* [6]. Lei et al. have presented a method to compute the race set in [6]. As shown in Fig.2, s_1 is synchronized with r_1 in Q_0, and there exists a race between s_1 and s_2 w.r.t. r_1 because s_2 is synchronized with r_1 in another SYN-sequence Q_1. Therefore, *race_set*(r_1, Q_0) is $\{s_2\}$. *race_set*(r_2, Q_0) is empty because s_1 and r_1 happen before r_2 and the synchronization between them cannot be changed.

A *race variant V* of a SYN-sequence *Q* is a prefix of *Q* by changing the sending part of one or more receiving events in *Q*. Let *r* be a receiving event that is synchronized with a sending event *s* in *Q*. If the sending part of *r* is changed to be *s′* in *V*, the following conditions must be satisfied:

1. *s′* must be in the race set of *r* in *Q*;
2. Let *e* be a sending or receiving event in *Q*. Then *e* is not in *V* if and only if the existence of *e* might be affected by this change.

The first condition is evident. The second condition guarantees the feasibility of race variant *V*. The events which are affected by the sending partner change of *r* are removed to ensure that there exists at least one program execution in which the sequence of events in *V* can be exercised. For a SYN-sequence *Q*, one approach to generating race variants is to build a race table. Details on building a race table can be found in [6].

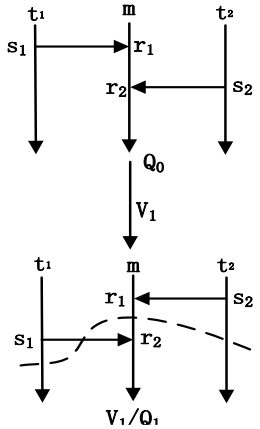


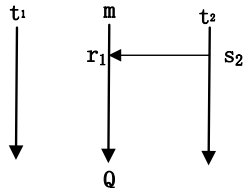
Fig.2. Reachability testing process of the example program

Reachability testing begins by executing a program non-deterministically. For the example program, we assume it initially exercises a SYN-sequence Q_0 in Fig.2. Then, the race set of each receiving event in Q_0 is computed to derive the variant of Q_0. Since *race_set*(r_1, Q_0) is $\{s_2\}$, the sending event of r_1 is changed to be s_2 and V_1 is generated. In V_1, r_2 is removed from Q_0 because the existence of r_2 is affected by r_1. Finally, the events in V_1 are s_1, r_1, and s_2, which are those above the dashed line in Q_1. Since *race_set*(r_2, Q_0) is empty, no variant is generated for r_2. After that, each variant is used to perform a prefix-based test run, in which the events and the synchronizations in the variant are controlled to be replayed. Then the test run continues non-deterministically again without controlling which SYN-sequence is exercised until it ends. In Fig.2, prefix-based testing is applied with only one variant, namely V_1, and then a complete sequence Q_1 is exercised. The sequence Q_1 and the variant V_1 are depicted in the same space-time diagram. Since no new variants can be derived from Q_1, the reachability testing process ends.

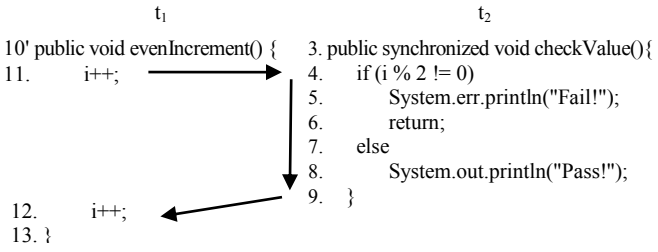
III. Splitting Strategy for Reachability Testing

A. Motivation

To make our technique comprehensible, in this section, we use the example in Fig.1 to illustrate the basic idea of our splitting strategy on how to improve reachability testing. For description purposes, each statement in the program is identified as its line number.



(a) Synchronization sequences exercised by reachability



(b) A buggy interleaving

Fig. 3. Reachability testing after removing the keyword *synchronized* and a buggy interleaving that reveals the fault

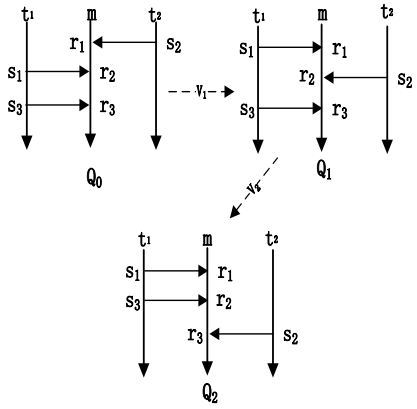
If the keyword *synchronized* in 10 in Fig.1 is incorrectly removed, i.e., 10 is replaced with 10', *evenIncrement()* is no longer a synchronized method. When reachability testing is performed on the faulty program, any event involving *evenIncrement()* call is not monitored and considered. Therefore, as shown in Fig.3(a), only one synchronization sequence *Q* is generated during reachability testing because only the sending and receiving events of synchronized method *checkValue(),* namely s_2 and r_1, are recorded and used to derive

new race variants. *Q* is not capable of detecting the fault in 10, i.e., no keyword *synchronized*. However, as shown in Fig.3(b), there exists a buggy interleaving, in which 11 is first executed, then 4, 5, and 6, finally 12. It can trigger the fault and fails to generate an even number.

To generate and exercise the buggy interleaving in Fig.3(b), the code in *evenIncrement()* could be split into two separate synchronized methods and then perform reachability testing. Fig.4 shows the split code and the corresponding synchronization sequences exercised by reachability testing. After splitting, three fine synchronized methods are used for generating race variants. Accordingly, three fine synchronization sequences, namely Q_0, Q_1, and Q_2, are generated and exercised during reachability testing. Q_1 forces the same execution as the buggy interleaving in Fig.3(b), hence revealing the fault in line 10.

```
10. public void
      evenIncrement() {
11.     synchronized(this){
12.           i++;
13.     }
14.     synchronized(this){
15.           i++;
16.     }
17. }
```

(a) Split code



(b) Reachability Testing

Fig. 4. Reachability testing after splitting

B. Splitting Strategy

1) Concurrent Mutation Operators: As mentioned in Section 1, the effectiveness of a testing technique is often evaluated by mutation testing. Recent empirical research also shows that mutants can be used as a substitute for real faults, and mutant detection is positively correlated with real fault detection [10].

According to Java concurrency defect patterns, researchers have designed twenty-seven concurrent mutation operators to generate concurrent mutants [8]. However, performing mutation testing for concurrent programs is especially expensive since each test on each mutant has to be executed for all possible thread schedules. To speed up mutation testing, we can select a subset of concurrent mutation operators to represent all mutation operators, i.e., a test suite that kills all mutants generated by this subset also kills (almost) all mutants generated by all mutation operators. A prior empirical study provides several such subsets of concurrent mutation operators [8]. In this paper, we select one subset, namely {RSK, RTCX, SKCR, SPCR, WTIS, MSP, SHCR}. Using this subset, one can generate and test only about half of the mutants (reduce 46.37%) while achieving a mutation score as high as 99.67% [8]. The seven concurrent mutation operators are summarized as follows.

- **RSK** (Remove Synchronized Keyword from Method) RSK removes the keyword *synchronized* from a synchronized method, which may cause data races or atomicity violations.
- **MSP** (Modify Synchronized Block Parameter) MSP replaces an object that is used as a parameter for a synchronized block with another object. The MSP operator may result in a wrong lock fault.
- **RTXC** (Remove Thread Method-X Call) RTXC removes calls to the following methods: *wait()*, *notify()*, *notifyAll()*, *join()*, *sleep()*, and *yield()*. Removing a *wait()* method may lead to incorrect interference between threads whereas removing a *notify()* or *notifyAll()* method call may cause an infinite waiting fault.
- **SHCR** (Shift Critical Region) SHCR shifts up or down a critical region, which may also cause data races or atomicity violations by no longer synchronizing access to shared variables.
- **SKCR** (Shrink Critical Region) SKCR move the statements which require synchronizations outside critical sections. SKCR causes similar concurrency faults to SHCR.
- **SPCR** (Split Critical Region) SPCR splits a critical region into two regions, which may cause a set of statements that should be atomic to become non-atomic.
- **WTIS** (Replace While With If Inside Synchronized) WTIS replaces *while* with *if* inside a synchronized method or block. Typically, *a while-wait structure* is required to implement a monitor. If it is replaced with *if*, it may cause an order violation.

Reachability testing, which generates and exercises (almost) all partial-order synchronization sequences, excels in detecting concurrency faults caused by faulty interleavings between synchronization events. It can kill concurrent mutants generated by applying RTCX, SPCR, and WTIS operators since such faults are often exposed in specific synchronization sequences. For RSK, SHCR, MSP and SKCR mutants, reachability testing does not guarantee to generate proper synchronization sequences to detect them because non-synchronized methods or blocks are not monitored and then used to derive race variants. However, such mutants could be killed by fine-grained synchronization sequences using splitting strategy. As discussed in Section 3.1, removing the keyword *synchronized* for *evenIncrement()* is an RSK mutant, which can be killed after splitting the method.

To achieve a tradeoff between effectiveness and cost, in this paper we split a method or a block into two sections for the following reasons. First, if a method or block is split into more than two sections, the number of synchronization sequences

may increase rapidly, even exponentially in many cases. Second, an empirical study on real-world concurrency faults have shown that the manifestation of almost all (nearly 97%) non-deadlock faults involve only two threads [11]. Moreover, most (nearly 90%) concurrency faults can be manifested if certain orders among two, three or four memory accesses are enforced. Suppose that two threads are involved in the manifestation of a non-deadlock concurrency bug. If the order between two memory accesses are scheduled, there exists one memory access in each thread. For three memory accesses, there exists one memory accesses in one thread, and two in the other. Similarly, for four memory accesses, there exist two memory accesses in each thread (The other case is one memory access in one thread, and three in the other. However, this case can be regarded as three memory accesses since the order of three accesses in one thread is certain and only two accesses in the thread and one access in the other thread are required to be scheduled). In summary, at most two accesses in each thread are needed to be considered. As a result, splitting a method or block into two sections can substantially reduces the number of test sequences while keeping a good concurrency fault detection capability.

2) Splitting Point Selection: For description conveniences, unless otherwise specified, a method and a code block are not distinguished in the sequel. Both of them are regarded as methods.

Definition 1 Let *m* be a method in a concurrent program *CP*. *CP* has a concurrency fault *cf*. After *m* is split into two sections at a point *p*, *CP* becomes *CP′*. Let *QS* be the set of synchronization sequences generated by reachability testing of *CP′*. If there exists a synchronization sequence in *QS*, which is capable of detecting the fault *cf*, we say that the point *p* is an *effective splitting point* w.r.t. *cf*.

In Fig.3(b), the point *p* between line 11 and line 12 is an effective splitting point. After the method *evenIncrement()* is split into two sections at *p,* there exists a synchronization sequence, namely Q_1 in Fig.4(b), which is capable of detecting the removing keyword *synchronized* fault.

Given a concurrency fault *cf*, assume that there exists at least one effective splitting point w.r.t. *cf*. It is a difficult task to precisely identify such effective splitting points using static or dynamic program analysis techniques, due to the inherent complexity of concurrent programs. With a large number of experimental studies, we empirically find two heuristic strategies to identify effective splitting points.

Strategy 1 A point *p* may be an *effective splitting point* if the intersection set between the two blocks after splitting at *p* contains the most shared variables at all splitting points.

As we know, many concurrency faults are caused by non-atomic executions of two or more consecutive accesses to the same shared variables. If the corresponding code is split exactly at the point such that the atomicity is violated, fine-grained synchronization sequences will be generated to expose the fault. Motivation in Section 2 shows a good example for this strategy. The rationale behind Strategy 1 is as follows. If the intersection set between the two blocks after splitting contains more shared variables, this splitting point might cause more atomicity violations, hence having a greater probability to reveal such concurrency faults.

Strategy 2 If a method *m* has a *while-wait* structure, the point between the *while-wait* structure and the succeeding code in *m* may be an effective splitting point. Similarly, if *m* has an *if-notify* or *if-notifyAll* structure, the point between the *if-condition* and the succeeding code may be also an effective splitting point.

Strategy 2 considers another typical concurrent fault source, namely conditional synchronization. To implement conditional synchronizations, a *while-wait* structure and a corresponding *if-notify* or *if-notifyAll* structure are often used. Fig.5 shows the code. If a *while-wait* and the succeeding code are not properly protected within a lock, there may exist a time interval between them, which allows other threads to change the condition. Consequently, conditional synchronization would fail, which may lead to an incorrect program behavior, even a crash. If we split the code at the point between a *while-wait* structure and the succeeding code, i.e., the point between 4 and 5 in Fig.5(a), fine-grained synchronization sequences would be generated to expose the fault. Similarly, if a condition and the succeeding code in an *if-notify* structure are not protected within a lock, there also exists a time interval between them, hence yielding a non-conditional synchronization. Then the point between a *if-condition* and the succeeding code, i.e., the point between 2 and 3 in Fig.5(b), could also be an effective splitting point.

```
1.  while(condition)
2.  {
3.      wait();
4.  }
5.  ......
```
(a) while-wait structure

```
1.  if(condition)
2.  {
3.      notify();/notifyAll();
4.  }
5.  ......
```
(b) if-notify structure

Fig. 5. Code for illustrating Strategy 2

3) Splitting Point Prioritization: More than one candidate effective splitting points may be achieved using the above two heuristic strategies. To detect faults efficiently, these splitting points are prioritized based on some criterion. Specifically, we set two attributes for each splitting point, i.e., *numSV* and *condSyn*. The attribute *numSV* represents the size of a shared variable intersection set. The attribute *condSyn* indicates whether a splitting point is related to a *while-wait* or *if-notify/notifyAll* or not. If it is related to a conditional synchronization structure, *condSyn* is 1; otherwise, it is 0. We define the priority between two splitting points p_i and p_j as follows.

Definition 2 Let p_i, p_j be two candidate effective splitting points. p_i has a higher priority than p_j, denoted as $p_i \triangleright p_j$, if one of the two following conditions is satisfied:

(1) $p_i.condSyn > p_j.condSyn$;

(2)$p_i.numSV > p_j.numSV$ if $p_i.condSyn$ equals to $p_j.condSyn$.

Given the control flow graph *g* of a method *m*, Algorithm 1 is used to sort candidate splitting points for *m*. Each splitting point is represented as an edge in the control flow graph. The algorithm first computes the intersection set of shared variables for each edge (lines 1-4). For each splitting point between n and n′, we compute shared variable sets in the first section and

second section respectively using *computeSV()*, which is realized by a classical fixed-point algorithm [12]. C*omputeSV(entry, n)* computes the shared variable set contained in all the paths from the *entry* of *g* to *n* whereas *computeSV(n′, exit)* computes the shared variable set contained in all the paths from *n′* to the *exit* of *g*. Thereafter, the algorithm searches a *while-wait* or *if-notify/notifyAll* structure and computes the attribute *condSyn* for each splitting point by calling *computeCondAttrib()* (line 5). Finally, Algorithm 1 computes the priority for each candidate splitting point and then sort them.

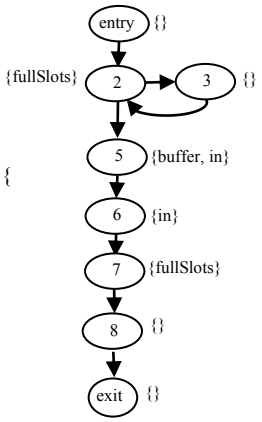
Input: the CFG of a method m
Output: a sorted list of splitting points spList

```
SortSplitPoint(cfg g)
      // Compute the intersection set of shared variables for each edge
1.   for each edge e=(n, n′) in g do
2.          FirstSV[e]=computeSV(entry, n);
3.          SecondSV[e]=computeSV(n′, exit);
4.          intersectionSV[e]= FirstSV[e] ∩ SecondSV[e];
     end for;
    //Search a while-wait or if-notify/ntifyAll structure and compute the
     attribute condSyn for each splitting point
5. condSyns = computeCondAttrib(g);
6. spList=quickSort(intersectionSV, conSyns);
```

Algorithm 1 SortSplitPoint

```
1.  public void deposit(int value) {
2.      while(fullSlots == capacity) {
3.          wait();
4.      }
5.      buffer[in] = value;
6.      in = (in + 1) % capacity;
7.      ++fullSlots;
8.      notifyAll();
9.  }
```

(a)



(b)

Fig. 6. BoundedBuffer program and its CFG graph

Fig.6 provides an example for illustrating Algorithm 1. First, Algorithm 1 computes the intersection set of shared variables for each edge. For (*entry*, 2), the set is empty since the path from *entry* to *entry* contains only the *entry* node, in which no shared variable appears. For (2, 3), the set of shared variables appearing in all the paths from *entry* to 2 is {*fullSlots*} whereas the set of shared variables contained in all the paths from 3 to *exit* is {*fullSlots, buffer, in*}. Then the intersection set is {fullSlots}. Note that *capacity* is not considered because it is a constant. Similarly, the intersection set for (3, 2) is {*fullSlots*}, (2, 5) is {*fullSlots*}, (5, 6) is{*in, fullSlots*}, (6, 7) is {*fullSlots*}, both (7, 8) and (8, *exit*) are empty. After that, it computes the attribute *condSyn*, which is 1 only for (2, 5). For each other edge, it is 0. Finally, according to definition 2, it combines the values of two attributes to compute the priority of each edge, and sort them. The sorting result is as follows: (2, 5), (5, 6), (2, 3), (3, 2), (6, 7), (7, 8), (*entry*, 2), and (8, *exit*).

4)Reachability Testing Based on Splitting Strategy: We propose Algorithm 2 to improve reachability testing with our splitting strategy. Given a concurrent program cp, the algorithm ReachTestWithSplit first invokes reachability testing function RichTest() to generate synchronization sequences sq (line 1). If sq detects the fault, it will return 1 (lines 2-3). Otherwise, the SortSplitPoint algorithm is invoked to generate a ranked candidate splitting point list spList for m (line 4). After that, m is split using the first candidate splitting point, and reachability testing is performed. If the fault is detected, it will return; otherwise, the process continued using the next candidate splitting point until the fault is detected or each candidate splitting point in spList has been tried (lines 5-9).

Input: a concurrent program cp and a mtethod m containing a concurrency fault
Output: a test sequence set sq that detects the fault

```
ReachTestWithSplit(cp: a concurrent program, m: a method)
1.   sq=RichTest(cp);
2.   if the fault is detected then
3.       return 1;
     end if;
4.   spList=SortSplitPoint(m);
5.   for each point p in spList do
6.       split the method m at the point p;
7.       sq=RichTest(cp′);        // cp′ is the program after splitting
8.       if the fault is detected then
9.           return 1;
         end if;
     end for;
10.  return 0;
```

Algorithm 2 ReachTestWithSplit

Empirical study (Section 5.2) shows that in most cases splitting the faulty method *m* itself can facilitate generating test sequences to expose the fault. Yet it is not always useful. In this case, other *relevant* methods can be split to generate effective test sequences using a similar splitting strategy in Algorithm 1. Relevant methods are identified using program slicing technique [13]. First we select some important or related shared variables in a method *m* as the slicing criteria, then perform backward and forward slicing. If the generated slice contains some statement in a method *n*, then *n* is a relevant method. For space limitations, details on splitting relevant methods are not described.

IV. IMPLEMENTATION

We have developed a prototype tool called SplitRichTest, which adopts the framework of conventional reachability testing tool RichTest while implementing our splitting strategy described in Section 3 [6-7]. As shown in Fig.7, SplitRichTest consists of three major components: Controller, Splitting Point Analyzer, and RichTest.

Controller collects execution results during reachability testing and checks whether a fault is detected or not. If it is detected, it saves the synchronization sequence as a test case. Splitting Point Analyzer is responsible for splitting a mutated or relevant method. RichTest is responsible for performing

reachability testing using. RichTest, which provides various supports for testing and debugging Java multithreaded programs, is responsible for performing reachability testing [6-7].

As for static program analysis, SplitRichTest uses *ThreadLocalObjectAnalysis* in Soot to perform a static thread-escape analysis and identify shared variables [14]. It distinguishes *read* and *write* accesses to remove shared immutable variables, which are never changed after initialization. It also leverages Spark to perform a pointer analysis.

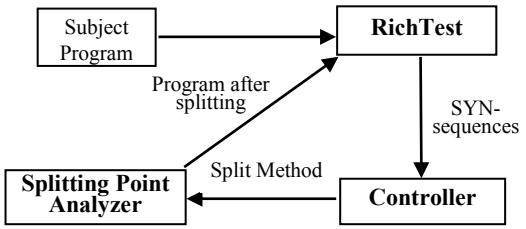


Fig.7. Architecture of SplitRichTest

V. EVALUATION

To assess the efficacy of SplitRichTest, we have conducted an empirical study on concurrent Java programs. The goal of this experiment is to evaluate concurrency fault detection capability of SplitRichTest, the effectiveness of our splitting point selection and sorting strategy, and the performance of Split RichTest, as compared with conventional reachability testing tool RichTest, and a state-of-the-art comprehensive concurrency fault prediction tool Pecan [9]. Pecan，which leverages various faulty access patterns to search faulty interleavings, can detect many types of concurrency faults, such as data races, atomicity violations, atomicity-set violations, etc. Typically, it first collects some trace of the program under test, then statically creates schedules that could expose faults, finally performs deterministic thread schedules to verify their feasibility. Our research questions are summarized as follows:

RQ1 What is concurrency fault detection capability of SplitRichTest?

RQ2 How effective is our splitting point selection and sorting strategy?

RQ3 How many test sequences are generated by SplitRichTest? What is the performance of SplitRichTest?

A. Evaluation Methodology

We evaluated SplitRichTest, RichTest and Pecan on a set of popular concurrent Java programs, mostly used in prior studies [1, 6, 7, 9, 15]. The synchronization or communication patterns in these multi-threaded programs are representative and widely used in many practical concurrent systems. Table 1 shows their properties including the number of lines of codes (LOC), the number of threads (Thread), and the number of shared variables (SharVar) in the programs. *Account* and *BuggyPrg* (BugProg) are selected from IBM contest benchmark suite [15]. *Bridge* and *ReadWrite* are solutions to the classical single-lane bridge and reader/writer problems respectively [16]. *BoundBuff* and *DisMutEx* are subjects in conventional reachability testing [6, 7， 17]. *BoundBuff* implements a bounded-buffer for producer-consumer problem [6]. *DisMutEx* solves the distributed mutual exclusion problem with three nodes in a computer network [7, 17]. *MineDrain*, a solution to mine drainage, represents industrial control concurrent systems [1]. The last three subjects, namely *StringBuff*(StringBuffer), *HashTab*(HashTable) and *Vector*, are open libraries in Suns JDK 1.4.2, from which many similar subjects are selected to evaluate Pecan [9].

We selected seven concurrent mutation operators mentioned in Section 3.2.1, i.e., {RSK, RTCX, SKCR, SPCR, WTIS, MSP SHCR}, to generate concurrent mutants. Since there is no available mutation tool especially designed for concurrent mutants, we manually generated mutants for each subject program, and removed all equivalent mutants, which are syntactically different but functionally equivalent to original programs. Table 2 shows the number of non-equivalent mutants for each program and each concurrent mutation operator. There are totally 283 non-equivalent concurrent mutants in this experiment.

TABLE 1 SUBJECT PROGRAMS

Program	LOC	Thread	SharVar
Account	143	4	4
BugProg	152	5	4
Bridge	118	5	5
ReadWrite	105	5	2
DisMutEx	255	7	6
BoundBuff	120	7	4
MineDrain	547	11	16
StringBuff	1,354	3	2
HashTab	1,245	3	29
Vector	1,560	3	15

Our experiments were performed on Windows 7, running on a 3.2GHz, Intel Core i5-3470 CPU with 32 GB memory. The version of Java virtual machine is Java HotSpot Server.

TABLE 2 THE NUMBER OF NON-EQUIVALENT MUTANTS

Program	RSK	RTXC	SKCR	SPCR	WTIS	MSP	SHCR	Total
Account	2	2	7	4	1	0	0	16
BugProg	1	2	9	7	1	0	0	20
Bridge	4	4	26	18	2	0	0	54
ReadWrite	4	4	8	4	2	0	0	22
DisMutEx	3	0	5	3	0	0	0	11
BoundBuff	2	4	27	28	1	0	0	62
MineDrain	11	2	21	8	1	8	2	53
Stringuff	2	0	2	1	0	1	2	8
HashTab	2	0	10	6	0	1	0	19
Vector	5	0	8	3	0	2	0	18
Total	36	18	123	82	8	12	4	283

B. Experimental Results

1) Concurrency Fault Detection Capability (RQ1): Table 3 shows the number of killed mutants by RichTest, SplitRichTest, and Pecan respectively. The column "RSK", "RTXC", "SKCR", "SPCR", "WTIS", "MSP", and "SHCR" report the number of killed mutants generated by applying corresponding mutation operators respectively. The column "Total" reports the total number of killed mutants for each program. The row "Killed Mutants" and "Mutation Score" reports the number of killed mutants and the mutation score respectively. The experimental results show that SplitRichTest achieves the highest mutation score (97.2%), then Pecan (66.8%), finally RichTest (62.5%). SplitRichTest kills almost all mutants, which implies that our splitting strategy significantly enhances detecting concurrency fault capability for reachability testing.

From the results in Table 3, we also find that both RichTest and SplitRichTest are capable of killing all mutants generated by applying RTXC, SPCR and WTIS while Pecan are not. The reason is that RTXC, SPCR and WTIS mutants are created due to improper synchronizations. Reachability testing can generate every partial-order synchronization sequence and thus reveal such faults while Pecan cannot.

The main difference between SplitRichTest and RichTest is the capability of killing concurrent mutants generated by applying RSK, SKCR and SHCR mutation operators. A common feature of these concurrent mutation operators is to modify critical regions, i.e., to make accesses to shared variables lose proper protections or synchronizations. Experimental results show that SplitRichTest kills many more such concurrent mutants than RichTest. It is caused by the fact that is that SplitRichTest split relevant codes into several critical regions and then generate fine-grain synchronization sequences to reveal more faults while RichTest does not consider non-synchronized methods and cannot derive interleavings between non-synchronization events.

Note that the mutation score of our SplitRichTest is only 33.3% for MSP mutants. The reason is that the original code may be no longer protected if a synchronization object is replaced by another. Even though splitting relevant codes, SplitRichTest does not ensure to generate interleavings between synchronization events with different monitor objects.

Pecan detects concurrency faults based on specific bug patterns, especially data races, atomicity violations, and atomic-set violations. As a result, it is not capable of detecting other concurrency faults, such as order violations and deadlocks. For instance, there is an RSK mutant of DisMutEx that yields a deadlock in a specific schedule. SplitRichTest generates all schedules and then kills them whereas Pecan generates a very limited number of schedules that cannot detect it. Moreover, Pecan is sensitive to the initial trace. If a faulty statement is not executed in the initial trace, the fault will not be detected by Pecan. In contrast, our SplitRichTest is not restricted by these limitations, hence killing more concurrent mutants than Pecan.

2) Splitting Point Selection and Sorting (RQ2): To evaluate the effectiveness of our splitting point selection and sorting algorithm (Algorithm 1), we analyzed the 98(275-177) non-equivalent concurrent mutants, which are not killed by RichTest but killed by SplitRichTest.

Table 4 reports the statistical number of killed mutants using different splitting points. The column "FP", "SP", and "TP" and "RP" show the number of killed mutants using the first, the second, the third and succeeding effective splitting points, and splitting relevant methods, respectively. The row "Mutation Score" reports mutant scores for each type of splitting points. The results in Table 4 show that our splitting point selection and sorting algorithm is very effective in detecting concurrency faults. Almost all concurrent mutants (94.9%) are killed after splitting the faulty method itself. The left 5.1% of concurrent mutants can be detected by splitting relevant methods. Most concurrent mutants (68.4%) are killed using the first splitting points, and 26.5% mutants are killed using the second, third and succeeding splitting points. For DisMutEx, due to the empty shared variable intersection set, our splitting point sorting strategy is ineffective.

TABLE 3 THE NUMBER OF KILLED MUTANTS

Program	RSK			RTXC			SKCR			SPCR			WTIS			MSP			SHCR			Total		
	RichTest	Pecan	SplitRichTest	RichTest	Pecan	SplitRichTest	RichTest	Pecan	SplitRichTest	RichTest	Pecan	SplitRichTest	RichTest	Pecan	SplitRichTest	RichTest	Pecan	SplitRichTest	RichTest	Pecan	SplitRichTest	RichTest	Pecan	SplitRichTest
Account	1	1	2	2	1	2	3	6	7	4	4	4	1	1	1	0	0	0	0	0	0	11	13	16
BugProg	1	1	1	2	0	2	4	7	9	7	5	7	1	0	1	0	0	0	0	0	0	15	13	20
Bridge	0	4	4	4	0	4	12	24	26	18	16	18	2	2	2	0	0	0	0	0	0	36	46	54
ReadWrite	0	4	4	4	1	4	4	8	8	4	3	4	2	2	2	0	0	0	0	0	0	14	18	22
DisMutEx	0	0	3	0	0	0	2	0	5	3	0	3	0	0	0	0	0	0	0	0	0	5	0	11
BoundBuff	2	2	2	4	0	4	20	22	27	28	22	28	1	1	1	0	0	0	0	0	0	55	47	62
MineDrain	1	7	11	2	0	2	8	16	21	8	5	8	1	0	1	1	4	2	0	1	2	21	33	47
StringBuff	1	2	2	0	0	0	2	1	2	1	0	1	0	0	0	0	1	1	1	1	2	5	5	8
HashTab	0	0	2	0	0	0	4	6	10	6	1	6	0	0	0	0	0	1	0	0	0	10	7	19
Vector	0	2	5	0	0	0	2	4	8	3	0	3	0	0	0	0	1	0	0	0	0	5	7	16
Killed mutants	6	23	36	18	2	18	61	94	123	82	56	82	8	6	8	1	6	4	1	2	4	177	189	275
Mutation Scores	16.6%	63.9%	100%	100%	11.1%	100%	49.6%	76.4%	100%	100%	68.3%	100%	100%	75.0%	100%	8.30%	50.0%	33.3%	25.0%	50.0%	100%	62.5%	66.8%	97.2%

Table 5 reports average number of test sequences exercised during RichTest, SplitRichTest, and Pecan for all non-equivalent mutants. SplitRichTest generates the most test sequences, then RichTest, and finally Pecan. After splitting, more synchronized methods are used to derive race variants, therefore generating and exercising more synchronization sequences when using SplitRichTest. For *BugProg* and *BoundBuff*, a large number of test sequences are unexpectedly generated when performing RichTest and SplitRichTest. It is due to the fact that the key factor influencing the number of sequences is not the size of the program but the complexity of synchronization behaviors between threads.

TABLE 4 THE NUMBER OF KILLED MUTANTS USING DIFFERENT SPLITTING POINTS

Program	FP	SP	TP	RP	Total
Account	0	5	0	0	5
BugProg	5	0	0	0	5
Bridge	18	0	0	0	18
ReadWrite	8	0	0	0	8
DisMutEx	0	0	5	1	6
BoundBuff	7	0	0	0	7
MineDrain	12	5	9	0	26
StringBuff	2	0	0	1	3
HashTable	7	2	0	0	9
Vector	8	0	0	3	11
Killed Mutants	67	12	14	5	98
Mutation Score	68.4%	12.2%	14.3%	5.1%	100%

TABLE 5 THE NUMBER OF TEST SEQUENCES

Program	RichTest	SplitRichTest	Pecan
Account	6	30	21
BugProg	630	16,800	9
Bridge	2,056	13,511	60
ReadWrite	2,888	19,043	27
DisMutEx	4,032	12,999	4
BoundBuff	12,096	2,481,408	23
MineDrain	2,590	7,960	12
StringBuff	3	6	4
HashTab	2	13	2
Vector	2	3	8

TABLE 6 THE EXECUTION TIME (SEC)

Program	RichTest	SplitRichTest	Pecan
Account	0.09	0.17	69.8
BugProg	1.35	28.8	67.6
Bridge	3.61	22.2	73.2
ReadWrite	5.42	31.5	48.6
DisMutEx	13.98	61.8	45.7
BoundBuff	16.8	8,449	56.2
MineDrain	456	930	42.2
StringBuff	0.07	0.09	42
HashTab	0.077	0.18	53.8
Vector	0.067	0.07	47.8

Table 6 reports average execution time for all non-equivalent mutants. Since the execution time for reachability testing mainly depends on the number of synchronization sequences, the execution time in Table 6 approximately coincides with the number of sequences in Table 5. Even though the execution time taken by SplitRichTest is increased significantly, it is still in a reasonable amount of time for most subjects.

Completely different from reachability testing, Pecan uses fault patterns to generate test sequences. Therefore, it generates much less test sequences than reachability testing. However, the execution time taken by Pecan is not accordingly reduced because it has to take a long time to collect traces, generate new schedules and re-execute test sequences to verify their feasibility. Unlike RichTest and SplitRichTest, the number of test sequences is not the dominating factor of the execution time for Pecan. Thus, sometimes SplitRichTest takes less time than Pecan whereas sometimes it takes more time.

C. Threats to Validity

The internal threat of our experiment is SplitRichTest may not correctly implement our splitting strategy. To reduce this threat, we checked the execution results of SplitRichTest on all small programs whose lines of code are less than 1,000. The external threat of our experiment is our ability to generalize the results to other concurrent programs. While ten representative concurrent programs are examined, the trends we observed may not necessarily represent all concurrent programs, especially industrial applications. To reduce this threat, we need to conduct more experiments on industrial systems.

VI. RELATED WORK

There has been much research on concurrent program testing techniques [2-7, 9, 18-34]. In this section, we classify related work into the following categories: coverage driven testing, stress or random testing, systematic testing, concurrency bug-based testing, and other concurrent testing techniques.

Coverage Driven Testing. Several coverage criteria have been proposed for testing concurrent programs [3, 18-19]. R. N. Talyor et al. present several control flow coverage criteria based on concurrency graphs (CG), in which each node represents a concurrency state and each edge represents a synchronization transition [3]. Definition-use coverage is also extended to concurrent programs based on extended control flow graph (ECFG), which consists of a number of control flow graphs (one for each thread) connected by synchronization edges [18]. S. Lu et al. propose a family of coverage criteria based on interleaving spaces of concurrent programs [19]. However, these above coverage criteria focus on theoretical investigations. No corresponding tool has been developed to verify the effectiveness and facilitate detecting concurrency bugs. Moreover, as these test models are statically constructed, they are not capable of capturing dynamic behaviors of concurrent programs and then avoiding infeasible test sequences. In contrast, reachability testing overcomes these problems and derives test sequences on-the-fly without building static models. The corresponding tool RichTest has been implemented. Our SplitRichTest further improves its capability of detecting concurrency bugs.

Stress or Random Testing. Stress testing is easy to perform, but its inefficiency limits its applications due to

unnecessarily executing the same thread interleavings. To test more diverse interleavings of concurrent programs, researchers have presented random testing that perturbs normal thread schedules and then yields different interleavings [2, 4]. Contest inserts random delays at synchronization points [2]. PCT appoints threads random priorities and modifies them at random points during test runs [4]. The weakness of random testing is that it may still produce duplicate interleavings, and may not explore a specific interleaving that can reveal a concurrency bug. Comparing to these testing techniques, reachability testing avoids repeatable interleavings and systematically runs every partial order interleaving exactly once.

Systematic Testing. Systematic testing tries to test all possible interleavings by systematically exploring state spaces of concurrent programs. This technique is ensured to test a distinct interleaving in each different run, and then detect various concurrency bugs. Yet it may lead to a significant overhead as the interleaving space is exponential theoretically. To alleviate the problem, static and dynamic partial order reduction techniques are used to reduce the number of interleavings that have the same partial order events to some extent [20]. Also, heuristic methods are presented to reduce test costs at the expense of missing interleavings that may trigger other concurrency bugs[5, 21, 22]. For example, Chess decreases the number of thread schedules by setting the bound of preemption points less than 2 [5]. C. Wang et al. propose a selective search strategy which guides exploration toward covering high-risk statement pairs that have not been covered previously [21]. S. Hong et al. present an interleaving exploration approach, which aims to cover only synchronization pairs [22]. S. Lu et al. propose an approach to testing thread safety class based on method pair coverage [23].

In contrast, reachability testing performs an efficient search to systematically run each partial order synchronization sequence exactly once without any loss of concurrency bug detection capability [6]. To reduce the execution time, distributed reachability testing is proposed and allows different SYN-sequences to be exercised concurrently by multiple different workstations without any synchronization [24]. Additionally, combinatorial strategies are used to reduce the number of synchronization sequences that need to be exercised [7, 25]. The coverage criteria are also leveraged to select race variants which should contain uncovered synchronizations while reachability testing is used to select feasible synchronization events to be exercised [26]. This paper first presents an approach that improves reachability testing by splitting non-synchronized methods. Our splitting strategy facilitates generating fine synchronization sequences to effectively detect concurrency bugs involved in access anomalies, such as accessing shared variables without proper protections or synchronizations, incorrectly shrinking or shifting critical regions, etc.

Concurrency Bug-based Testing. Most concurrency bug-based testing techniques first predict certain types of potential concurrency bugs using static or dynamic analysis, and then generates suspicious buggy thread schedules to verify whether they are real bugs or not [9, 27-29, 34-35, 38]. K. Sen et al. present a family of randomized active thread schedule strategy to test data races, atomicity violations, and deadlocks [27-29]. Maple presents a generic set of interleaving idioms that are based on dependencies between two threads [30]. The main limitation of these techniques is that test efficiency is rather low due to imprecise prediction on potential concurrency bugs. J. Huang presents another data race testing technique that does not need generate schedules for verification [31]. This technique first constructs constraints that buggy interleavings should satisfy, then solve them. The solution corresponds to an interleaving that can trigger the bug. Yet this technique may miss many real data race bugs because of excessive strict constraints. As mentioned in Section V, J. Huang et al. also propose a comprehensive tool called Pecan to detect general access anomalies including data races, atomicity violation, and atomicity set serializability violations [9]. The main disadvantage of concurrency bug-based testing techniques is that they are tailored to expose some specific types of concurrency bugs, and may not expose other types. In contrast, our SplitRichTest, which does not suffer from this limitation, is capable of detecting almost all kinds of concurrency bugs.

Besides, researchers have presented some other concurrent testing techniques [32-34]. C. Flanagan develops a tool called RedCard, which enables dynamic race detectors to reduce the number of run-time checks notably without loss of precision [32]. Redundant race checks are identified by reasoning about memory accesses within release-free spans. Y. Cai improves the technique to detect data races based on happen-before relation [33]. He proposes a tool called DrFinder to detect hidden races by reversing possible happen-before edges in the previous execution. S. Park et al. implement a fault localization tool called Falcon to rank data races or atomicity violations based on a large scale of collections of traces [34].

VII. CONCLUSIONS

In this paper, we have presented a splitting strategy for improving concurrency fault detection capability of reachability testing. Using our splitting strategy, relevant codes are split and fine-grained synchronization sequences are generated and exercised to detect more concurrency faults. Adopting the framework of reachability testing, we have developed a prototype tool SplitRichTest to implement our splitting strategy. Experimental results show that SplitRichTest significantly enhances the capability of concurrency fault detection for reachability testing in a reasonable amount of time.

In the future work, we plan to conduct more case studies to verify the effectiveness of SplitRichTest. In particular, we will apply SplitRichTest to test industrial concurrent applications. In addition, we plan to further evaluate and improve SplitRichTest using concurrent mutants generated by applying other concurrent mutation operators.

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China under Grant No. 61472076 and No. 61472077, and the National key R&D Program of China under Grant No. 2018YFB1003902.

RERERRENCE

[1] A. Burns and A. Wellings. Real-Time systems and programming languages. Addison Wesley Longman, 2001.

[2] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithread Java program test generation. Journal of IBM Systems, 41(1): 111-125, 2002.

[3] R. N. Taylor, D. L. Levine, and C. D. Kelly. Structural testing of concurrent programs. IEEE Transactions on Software Engineering, 18(3): 206-214, 1992.

[4] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). ACM, pp. 167–178, 2010.

[5] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In Proc. of the Usenix Symposium on Operating Systems Design & implementation(OSDI). Usenix Association, pp.267-280, 2008.

[6] Y. Lei and R.H. Carver. Reachability testing of concurrent programs. IEEE Transactions on Software Engineering, 32(6): 382-403, 2006.

[7] Y. Lei, R.H. Carver, R. Kacker, R. Kacker, and D. C Kung. A combinatorial testing strategy for concurrent programs. Software Testing, Verification and Reliability, 17(4): 207-225, 2007.

[8] M. Gligoric, L.Zhang, C. Pereira, and G. Pokam. Selective mutation testing for concurrent code. In Proc. of International Symposium on Software Testing and Analysis(ISSTA). ACM, pp.224-234, 2013.

[9] J. Huang and C. Zhang. Persuasive prediction of concurrency access anomalies. In Proc. of International Symposium on Software Testing and Analysis (ISSTA). ACM, pp.144-154, 2011.

[10] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In Proc. of ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE). ACM, pp.654-665, 2014.

[11] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes — A comprehensive study on real world concurrency bug characteristics. In Proc. of International Conference on Architectural support for programming languages and operating systems (ASPLOS). ACM, pp.329-339, 2008.

[12] F. Nielson, H. Nielson, and C. Hankin. Principles of Program Analysis. Berlin: Springer, pp.63-80, 1999.

[13] V. P. Ranganath and J. Hatcliff. Slicing concurrent Java programs using Indus and Kaveri. International Journal on Software Tools for Technology Transfer, 9(5): 489-504, 2007.

[14] http://www.sable.mcgill.ca/soot/

[15] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In Proc. of International Parallel and Distributed Processing Symposium (IPDPS). IEEE, pp.286-292, 2003.

[16] J. Magee and J. Kramer. Concurrency-State Models & Java Programs. John Wiley & Sons Ltd, 2006.

[17] G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. Communications of the ACM, 24(1): 9-17, 1981.

[18] Yang, A. L.Souter, and L. L. Pollock. All-du-Path coverage for parallel programs. In Proc. of International Symposium on Software Testing and Analysis(ISSTA). ACM, pp.153-162, 1998.

[19] S. Lu, W. Jiang, and Y. Zhou. A Study of Interleaving Coverage Criteria. In Proc. of joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering(ESEC/FSE). ACM, pp.533-536, 2007.

[20] C. Flanagan and P. Godefroid. Dynamic partial order reduction for model checking software. In Proc. of ACM SIGPLAN SIG-ACT Symposium on Principles of Programming Languages (POPL). ACM, pp.110-121, 2005.

[21] C. Wang, M. Said, and A. Gupta. Coverage guided systematic concurrency testing. In Proc. of International Conference on Software Engineering (ICSE). ACM, pp. 221-230, 2011.

[22] S. Hong, J. Ahn, S. Park, M. Kim, and M. J. Harrold. Testing Concurrent Programs to Achieve High Synchronization Coverage. In Proc. of International Symposium on Software Testing and Analysis(ISSTA). ACM, pp. 210-220, 2012.

[23] A. Choudhary, S. Lu, and M. Pradel. Efficient Detection of Thread Safety Violations via Coverage-Guided Generation of Concurrent Tests. In Proc. of ACM/IEEE International Conference on Software Engineering(ICSE). ACM/IEEE, pp. 266-277, 2017.

[24] R. Carver and Y.Lei. Distributed reachability testing. Concurrency and Computation: Practice and Experience, 22(18):2445-2466, 2010.

[25] X. Qi, J. He, P. Wang and H. Zhou. Variable Strength Combinatorial Testing of Concurrent Programs, Frontiers of Computer Science, 2016, 10(4): 631-643.

[26] S. R. S. Souza, P. S. L. Souza, M. C. C. Machado, M. S. Camillo, A. S. Simão, and E. Zaluska. Using coverage and reachability testing to improve concurrent program testing quality. In Proc. of International Conference on Software Engineering and Knowledge Engineering (SEKE). KSI, pp.207-212, 2011. _:

[27] K. Sen. Race Directed Random Testing of Concurrent Programs. In Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI). ACM, pp.11-21, 2008.

[28] C. S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In Proc. of ACM SIGSOFT International Symposium on Foundations of Software Engineering(FSE). ACM, pp.135–145, 2008.

[29] P. Joshi, C.S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI). ACM, pp. 110–120, 2009.

[30] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: A Coverage-Driven Testing Tool for Multithreaded Programs. In Proc. of ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). ACM, pp. 485-502, 2012.

[31] J. Huang, Meredith, P.O., and G. Rosu. Maximal sound predictive race detection with control flow abstraction. In Proc. of ACM SIGPLAN Symposium on Programming Language Design and Implementation (PLDI). ACM, pp.337-348, 2014.

[32] C. Flanagan and S. N. Freund. RedCard: redundant check elimination for dynamic race detectors. In Proc. of European Conference on Object-Oriented Programming (ECOOP). Springer, pp. 255-279, 2013.

[33] Y. Cai and L. Cao. Effective and precise dynamic detection of hidden races for Java programs. In Proc. of Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE). ACM, pp.450-460, 2015.

[34] S. Park, R. W. Vuduc, and M. J. Harrold. Falcon: fault localization in concurrent programs. In Proc. of ACM/IEEE International Conference on Software Engineering(ICSE). ACM/IEEE, pp.245-254, 2010.