IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received February 2, 2019, accepted February 18, 2019, date of publication February 25, 2019, date of current version March 12, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2901588

Efficient Test Case Generation for Thread-Safe Classes

LILI BO[1,2], SHUJUAN JIANG[1,2], JUNYAN QIAN[3], RONGCUN WANG[1,2], AND XINGYA WANG[4]

[1]School of Computer Science and Technology, China University of Mining and Technology, Xuzhou 221116, China
[2]Engineering Research Center of Mine Digitalization, Ministry of Education, Xuzhou 221116, China
[3]Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology, Guilin 541004, China
[4]State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China

Corresponding author: Shujuan Jiang (shjjiang@cumt.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61673384 and Grant 61562015, in part by the National Science Foundation of Jiangsu Province under Grant BK20181353, in part by the China Postdoctoral Science Foundation under Grant 2015M581887, in part by the Jiangsu Planned Projects for Postdoctoral Research Funds under Grant 2018K028C, and in part by the Innovation Project for State Key Laboratory for Novel Software Technology under Grant ZZKT2018B02.

ABSTRACT Generating test cases automatically for thread-safe classes is an effective approach to validating their correctness. However, the existing concurrent test generation techniques usually consume a large amount of time and efforts before finding concurrency bugs. To alleviate this problem, we present an automatic and efficient approach which combines the advantages of both the bug-driven and coverage-guided techniques to generate test cases for thread-safe classes. First, method pairs that cannot be executed concurrently are removed by the static analysis. Then, a strategy of the bug-driven grouping of method pairs is designed to divide the remaining method pairs into two groups. One group contains the method pairs with a high priority, and another group contains the method pairs with a low priority. Finally, iterative generation of concurrent test cases, which consists of the coverage-guided generation of concurrent tests and concurrency bug detection, is conducted to find concurrency bugs. Our evaluation is on 20 thread-safe classes. Compared with four state-of-the-art approaches, the results show that our approach can obtain a significant improvement in efficiency without impairing bug finding capacities.

INDEX TERMS Software testing, thread-safe class, test case generation, concurrency bug, static analysis.

I. INTRODUCTION

With the shared memory mechanism and the non-deterministic thread scheduling, concurrent software is difficult to program, to test and to debug. To alleviate this problem, programmers usually use thread-safe classes which encapsulate many concurrency-related challenges. Therefore, it is critical to ensure the correctness of thread-safe classes.

Automated test case generation for thread-safe classes is an effective approach to validating their correctness [1]–[3]. Previous studies have proposed many approaches to generating test cases for concurrent classes. They can be classified into three categories: random-based approaches [4]–[6], sequential test-based approaches [7]–[9] and coverage-based approaches [10]–[12]. Random-based approaches are inefficient as it may repeatedly test behaviors that have been exercised and miss behaviors that can expose bugs. Sequential test-based approaches depend on the pre-given test suite of sequential tests and produce an expensive computing cost. Coverage-based approaches lie on the pre-designed coverage requirements which guide test generation.

Recently, *concurrent method pairs* [13], an approximate interleaving coverage metric, is applied to concurrency bug detection and concurrent test generation. A research shows that 96% concurrency bugs involve no more than two threads [14]. Therefore, in the process of test generation, two methods can constitute a method pair. Then they are scheduled by two threads in different orders. Taking concurrent method pairs as coverage requirements can improve the efficiency of concurrent test generation. However, if any two methods of a class under test (CUT) constitute a method pair, there will be a large numbers of concurrent method pairs. In fact, many method pairs are redundant because two methods can neither execute concurrently nor lead to concurrency

The associate editor coordinating the review of this manuscript and approving it for publication was Weizhi Meng.

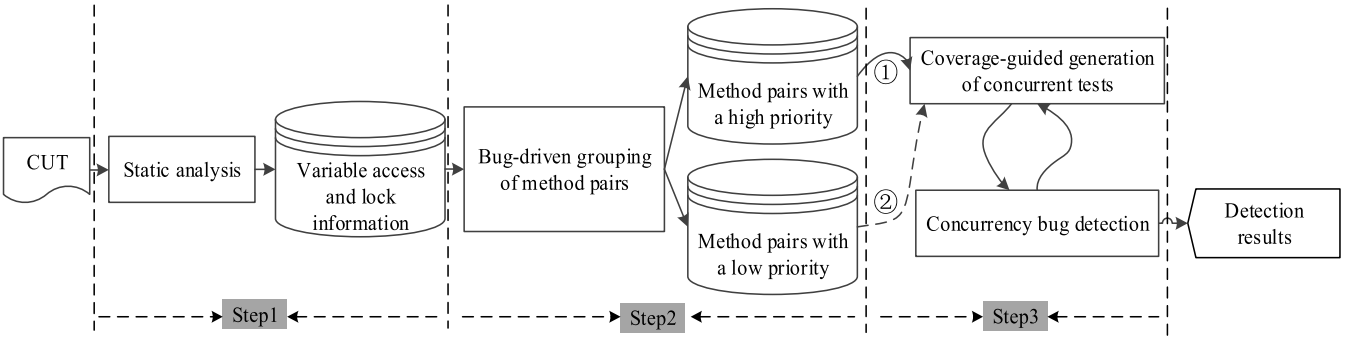


FIGURE 1. Overview of BC-ConTest.

conflicts. It is both time-consuming and useless to generate tests for these method pairs.

In this paper, we present a novel approach named BC-ConTest (Bug-driven and Coverage-guided Concurrent Test generation) to generate concurrent tests. It uses static analysis to improve the efficiency of test generation. First, we statically analyze the bytecode files of CUTs to filter the initial set of method pairs. In the initial set of method pairs, each method pair consists of any two methods. Then, we iteratively generate a concurrent test and execute it to detect concurrency bugs. In each iteration, method pair that is most likely to trigger bugs can be selected so as to find concurrency bugs as soon as possible. Finally, we evaluate the bug finding capacities and efficiency of BC-ConTest on 20 thread-safe classes and empirically compare it with the state-of-the-art approaches. The results show that our approach outperforms compared approaches.

The main contributions of this work are concluded as follows:

- We propose a novel test case generation approach named BC-ConTest for thread-safe classes. It combines bug-driven grouping of method pairs and coverage-guided generation of concurrent tests.
- We evaluate the bug finding capacities and efficiency of BC-ConTest on 20 thread-safe classes, and compare it with the state-of-the-art approaches. The results show that our approach can improve the efficiency significantly without impairing bug finding capacities.

The remainder of this paper is organized as follows. Our approach is described in Section II. Section III presents an empirical study to show its validity. Section IV summarizes the related work. Finally, we draw conclusions of this paper in Section V.

II. OUR APPROACH

A. OVERVIEW

The overview of our approach is shown in Fig. 1. It consists of three steps: static analysis, bug-driven grouping of method pairs and iterative generation of concurrent test cases. The first step is static analysis. We use static analysis to identify variable access and lock information in CUT. The second step is bug-driven grouping of method pairs. The method pairs are divided into two groups. One is the group of method pairs which have more possibility to expose concurrency bugs. Hence, it is given a high priority. The other is the group of the remaining method pairs. It is given a low priority. The final step is iterative generation of concurrent test cases, which consists of coverage-guided generation of concurrent test cases and concurrency bug detection. As this is a process of iteration, we take them as a whole. Concurrent test generation first selects method pairs from the group with a high priority. Only if all the method pairs in it have been selected yet not find concurrency bugs, method pairs from the group with a low priority will be selected. Our approach combines bug-driven grouping of method pairs with coverage-guided generation of concurrent tests, which can improve the efficiency of concurrency bug detection.

B. STATIC ANALYSIS

Static analysis is used to obtain the class information, including member variable accesses, member methods, method call relationships, lock and unlock operations. The reasons for static analysis firstly are that: 1) our approach requires nothing but classes under test, thus they cannot be executed in the beginning; 2) although classes under test will be executed with the driver of generated concurrent tests in the third step, dynamic analysis is extremely time-consuming; 3) bug-driven grouping of method pairs is based on the data accesses in methods, thus it is necessary to conduct static analysis before grouping.

As our approach is specific to Java classes, our static analysis is based on Soot, a Java bytecode analysis framework. The source code of CUT should firstly be complied into bytecode. If the current CUT extends some classes, its super classes will also be analyzed. Every field access in these classes is instrumented and identified by Soot.[1]

Note that our approach aims to generate concurrent tests for two publicly accessible methods, thus only public methods are instrumented and analyzed. Specifically, lock and unlock operations can be identified by EnterMonitorStmt()

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

and ExitMonitorStmt(). Corresponding locks can also be obtained. Synchronized methods can be identified by function isSynchronized(). Hence, the method pairs can be removed directly if both are synchronized methods. In addition, if method f_1 calls method f_2, field access information in f_2 will be copied into f_1. Finally, the complete class information is collected.

C. BUG-DRIVEN GROUPING OF METHOD PAIRS

A concurrency bug occurs when two threads access a common variable concurrently with at least a write access. Like [12], our approach generates a concurrent test for each method pair, which is constituted with two methods. We divide the method pairs into two groups based on the bug-driven techniques. The key insight of our grouping strategy is that, it is more likely to expose concurrency bugs if two threads access a common variable concurrently with at least a write access, i.e., accesses to common variables are not protected by common locks. In this case, we will assign a high priority to the method pairs. Then, the remaining method pairs are given a low priority.

First, we remove the method pairs, in which both are synchronized methods, since they cannot execute concurrently. Next, we focus on two situations to search for the common locks that protect the accesses of common fields: 1) writing common fields occurs in both methods, and 2) writing common fields occurs only in one method (all accesses to common fields are read in the other method). We call the first and the second situations as *write-write method pair* (*WW*) and *write-read method pair* (*WR*), respectively.

We adopt tree structure to record the *Maximum Protect Lock* (*MPL*) for all accesses of each common field. For the first situation, concurrency conflicts will not occur if all accesses to common fields are protected by the same locks. Therefore, *Maximum Protect Lock* of each common field in each method is defined as the follows.

Definition 1 [Maximum Protect Lock for WW (MPL_{WW})]: In each method m, given a set of all accesses to a common field x, ACC_{mx} and a set of all the lock objects $LOCK_{mx}$, *Maximum Protect Lock* is a subset of $LOCK_{mx}$, in which any of the locks can protect all accesses to the common field x. It can be formalized as the follows:

$$\begin{aligned} MPL = \{&lock_i | lock_i \in LOCK_{mx} \wedge \forall acc_x \in ACC_{mx}, \\ &acc_x \text{ is protected by } lock_i, \\ &i = 1, 2, 3, \ldots; x = 1, 2, 3, \ldots\} \end{aligned} \tag{1}$$

For the second situation, concurrency conflicts will not occur if only the write accesses in the write method and the read accesses in the read method are protected by the same locks. Therefore, *Maximum Protect Lock* of the common field in each method is defined as the follows:

Definition 2 [Maximum Protect Lock for WR (MPL_{WR})]: In method m_1 in which all accesses to the common field are read, given a set of all read accesses to a common field x, ACC_{mrx} and a set of all the lock objects $LOCK_{mx}$, *Maximum Protect Lock* is a subset of $LOCK_{mx}$, in which any of the locks can protect all accesses to the common field x. It can be formalized as the follows:

$$\begin{aligned} MPL_r = \{&lock_i | lock_i \in LOCK_{mx} \wedge \forall acc_x \in ACC_{mrx}, \\ &acc_x \text{ is protected by } lock_i, \\ &i = 1, 2, 3, \ldots; x = 1, 2, 3, \ldots\} \end{aligned} \tag{2}$$

In method $m2$, given a set of all write accesses to a common field x, ACC_{mwx} and a set of all the lock objects $LOCK_{mx}$, *Maximum Protect Lock* is a subset of $LOCK_{mx}$, in which any of the locks can protect all accesses to the common field x. It can be formalized as the follows:

$$\begin{aligned} MPL_w = \{&lock_i | lock_i \in LOCK_{mx} \wedge \forall acc_x \in ACC_{mwx}, \\ &acc_x \text{ is protected by } lock_i, \\ &i = 1, 2, 3, \ldots; x = 1, 2, 3, \ldots\} \end{aligned} \tag{3}$$

For each common field, we construct their access trees for two methods, separately. Then, we traverse the nodes in the trees top-down and calculate the *MPLs* of two methods (MPL_1 and MPL_2) according to the above definitions. If the intersection of MPL_1 and MPL_2 is null, it implies that the accesses to the common field are not protected by common locks. Thus, this method pair is assigned a high priority. Otherwise, it is assigned a low priority.

Next, we give an example to illustrate *MPL* and our static analysis. Fig. 2 presents two access trees for all common fields x and y in method m_1 and m_2, respectively. In each access tree, the root node, represented by a square box, indicates a method. The text in the box shows its method name. The difference is that, a double bordered square box means this is a synchronized method and the lock is *this*. Circles indicate the accesses (read and write accesses) to common fields. The lock and unlock nodes are represented by the corresponding symbols.

Both m_1 and m_2 contain read and write accesses to x and y. Specifically, there is a nested synchronized block in m_1. After entering m_1, lock l_1 is obtained. In this synchronized

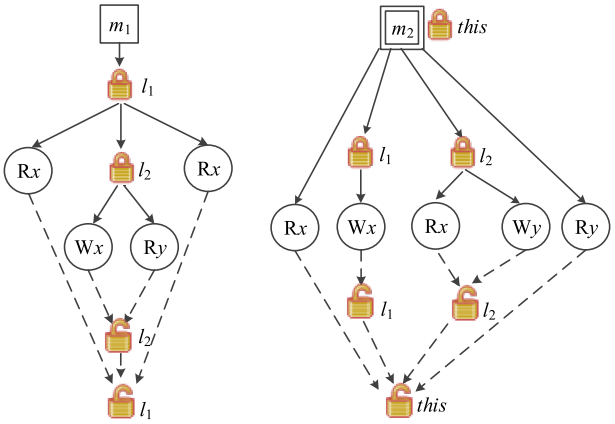


FIGURE 2. Access trees of all common fields (x and y) in method m_1 and m_2.

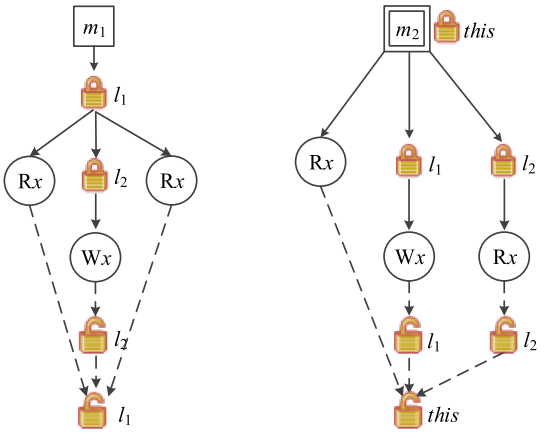


FIGURE 3. Access trees of the common field x in method m_1 and m_2.

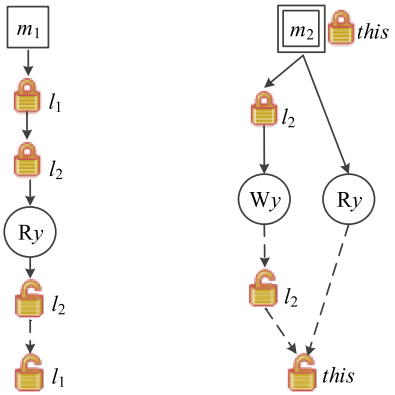


FIGURE 4. Access trees of the common field y in method m_1 and m_2.

block, field x is first read. Then, lock l_2 is obtained. In the synchronized block locked by l_2, the common fields x is written and then y is read. After that, lock l_2 is released followed by reading x. Finally, lock l_1 is released. In method m_2, the situation is different with that in m_1. First of all, object lock *this* is obtained as m_2 is a synchronized method. After entering m_2, the common field x is read. Then, two locks l_1 and l_2 are obtained and then released one by one. In the synchronized block locked by l_1, x is written. In the synchronized block locked by l_2, x is read followed by writing y. After the lock l_2 is released, y is read. At last, lock *this* is released.

Our approach constructs access trees for each common field separately. Fig. 3 and Fig. 4 present the access trees for the common field x and y, respectively. The method pair for x belongs to the first situation. The method pair for y belongs to the second situation. We randomly select a pair of access trees of a certain common field. Assume that the common field x is firstly selected. The *MPLs* can be calculated by (1). In Fig. 3, the *MPLs* of x in method m_1 and m_2 are $\{l_1\}$ and $\{this\}$. According to (4), we can conclude that invocations of method m_1 and m_2 concurrently may lead to concurrent conflict. Then, the method pair (m_1, m_2) is added to the group with a high priority.

$$\{l_1\} \cap \{this\} = \varnothing \tag{4}$$

Since (m_1, m_2) has been confirmed to be added to the group with a high priority, the access trees of other common fields will not be analyzed any more. However, if the common field y is first selected, the analysis is difference. As only read access to y occurs in method m_1, the MPL_r can be calculated by (2). Based on Definition 2 and (3), we can get MPL_w in method m_2. Thus, MPL_r and MPL_w of the common field y in m_1 and m_2 are $\{l_1, l_2\}$ and $\{this, l_2\}$, respectively. According to (5), we can find that the read accesses to y in m_1 and the write accesses to y in m_2 are protected by the common lock l_2.

$$\{l_1, l_2\} \cap \{this, l_2\} = \{l_2\} \tag{5}$$

However, whether a method pair should be added to the group with a low priority does not only depend on one field. In other words, only if all the fields are checked and the related accesses to them are protected by the common locks, the method pair can be added to the group with a low priority. In this example, we have to analyze access trees of the other common field y. As described above, invocations of method m_1 and m_2 concurrently may lead to concurrent conflicts. Thus, the analysis is stopped. We can get a conclusion that invocating m_1 and m_2 concurrently has a high probability to expose concurrent conflicts. Therefore, (m_1, m_2) is added to the group of a high priority.

The process of bug-driven grouping of method pairs is shown in Algorithm 1. Algorithm 1 takes the initial set of method pairs *InitMethPairs* constituted with any two methods of CUT as input, and takes two grouped set of method pairs *HighPrioMethPairs*, *LowPrioMethPairs* as output. For each method pair in *InitMethPairs*, it will be removed from *InitMethPairs* if both of the two methods in this method pair are synchronized methods (lines 2 and 3). Otherwise, bug-driven grouping of method pairs is conducted. First, the common fields are computed (lines 5-7). If there are no common fields, this method pair is given a low prior and the next method pair is considered (lines 8 and 9). Otherwise, for each common field, two access trees are constructed to represent all accesses of the common field in two methods. If all accesses to the common field are read in one method, *MPLs* are computed with equation (2) and equation (3) (lines 19-24). Otherwise, *MPLs* are computed with equation (1) (line 26). After computing *MPLs*, we check the intersection of two *MPLs*. If the intersection is null, this method pair is added to *HighPrioMethPairs*, which implies that it has more possibility to detect potential concurrency bugs if these two methods execute concurrently (lines 27 and 28). Then, other access trees in these two methods will not be analyzed any more. But if the intersection is not null, the access trees of the next common field should be analyzed. Only if the access trees of all common fields are checked and all the intersections are not null, this method pair will be given a low priority and then be added to *LowPrioMethPairs* (line 30).

Algorithm 1 Bug-Driven Grouping of Method Pairs

Input: *InitMethPairs* - the initial set of method pairs.
Output: *HighPrioMethPairs* - the set of method pairs with high priority.
LowPrioMethPairs- the set of method pairs with low priority.

begin
for each *methodPair* ∈ *InitMethPairs* **do**
//remove synchronized method pairs
if both of methods in *methodPair* are synchronized methods **then**
remove *methodPair* from *InitMethPairs*;
else
//group method pairs based on bug-driven idea
*AcceFields*1 = Set of accessed fields in the first method m_1;
*AcceFields*2 = Set of accessed fields in the second method m_2;
SamFields = *AcceFields*1 ∩ *AcceFields*2;
if *SamFields* = Ø **then**
continue;
else
TotalAccTypes = set of all the accessed types of all the same fields in *SamFields*;
if *TotalAccTypes*.size = 1 && *TotalAccTypes*.get(0)="*READ*" **then**
continue;
else
for each *samField* ∈ *SamFields* **do**
construct access trees representing all accesses of *samField*;
*AccTypes*1 = set of all accessed types of *samField* in m_1;
*AccTypes*2 = set of all accessed types of *samField* in m_2;
if (*AccTypes*1.size = 1 && *AccTypes*1.get(0) = "*READ*") **then**
compute MPL_1 with equation (2);
compute MPL_2 with equation (3);
else if (*AccTypes*2.size = 1&&*AccTypes*2.get(0) = "*READ*") **then**
compute MPL_1 with equation (3);
compute MPL_2 with equation (2);
else
compute MPL_1 and MPL_2 with equation (1);
if $MPL_1 \cap MPL_2 = Ø$ **then**
HighPrioMethPairs.add(*methodPair*);
break;
LowPrioMethPairs = *InitMethPairs*.removeall(*HighPrioMethPairs*);
return *HighPrioMethPairs*, *LowPrioMethPairs*;
end

D. ITERATIVE GENERATION OF CONCURRENT TESTS

After bug-driven grouping of method pairs, two sets are obtained, i.e., *HighPrioMethPairs* and *LowPrioMethPairs*.

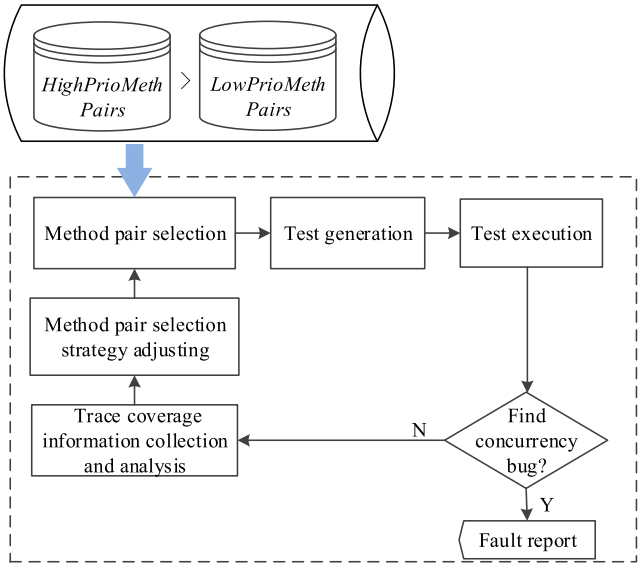


FIGURE 5. Overview of iterative generation of concurrent tests.

The method pairs in *HighPrioMethPairs* are more likely to expose concurrency bugs when two methods of each method pair execute concurrently. Therefore, we give preference to the method pairs in *HighPrioMethPairs* to generate concurrent test cases and then to detect concurrency bugs.

The generation of concurrent tests is a process of iterative learning. It is based on the coverage-guided techniques. The key insight is to guide test generation toward not yet covered interleavings. That can avoid repeatedly generating tests to explore the interleavings which cannot expose concurrency bugs.

In order to select two methods that are more likely to expose concurrency bugs when they execute concurrently, we designed a method pair prioritization strategy. This is inspired by the priority mechanism in [15]. Next, we present a formal definition of several symbols:

- p is a concurrent method pair, that is, two methods which may execute concurrently;
- r is the number of tests that tried to cover p;
- c is the number of tests that has indeed covered p in program execution;
- $S(p)$ is the prioritization score of p.

Then, $S(p)$ can be computed by the following formula (6):

$$S(p) = max(|r - c|, 1) \cdot r \tag{6}$$

where r is initialized to 1. Then, the number of r will plus one if the corresponding method pair is selected. As every iteration proceeds, r and c are updated. Hence, the prioritization scores of every method pairs are updated after every iteration.

The overview of iterative generation of concurrent tests is presented in Fig. 5. As shown in Fig. 5, method pairs in *HighPrioMethPairs* receive the high priority, that is, we firstly select method pairs in *HighPrioMethPairs*. After a method pair is selected, concurrent test cases will be generated for it.

Then, each concurrent test is executed to expose concurrency bugs based on the thread safety oracle. If concurrency bugs are not exposed, the trace coverage information is collected and analyzed. Based on the trace coverage information, prioritization scores for the method pairs are updated so as to select the method pair with the highest priority in the next iteration. If all the method pairs in *HighPrioMethPairs* are selected yet no concurrency bugs are found, our approach turns to select method pairs from *LowPrioMethPairs*. The procedure will be repeated until concurrency bugs in CUTs are found or timeout.

III. EXPERIMENTS

In this section, we conducted some experiments to evaluate the validity of our approach. All the experiments are conducted on the same platform, i.e., a virtual machine running the Ubuntu-16.04, hosted on a Dell server with 32GB of memory and two 2.40GHz XEON X5675 CPUs.

A. EXPERIMENTAL DESIGN

Our static analysis is implemented based on Soot, a Java program analysis framework. In addition, we utilize the existing test generator to generate concurrent test cases. In test execution, random scheduling of JVM scheduler is adopted to explore the scheduling space. All the data results are averaged over 10 times for each CUT. The time-out of bug detection in each time is one hour.

To comprehensively show the validity of BC-ConTest, we compare our approach with four existing concurrent test generation techniques, i.e., CovCon [12], ConTeGe [6], Nainom [7]–[9] and AutoConTest [11]. CovCon detects thread-safety violations via coverage-guided generation of concurrent tests. ConTeGe is a random-based approach. It generates concurrent tests by randomly selecting methods to schedule under multiple concurrent threads. Nainom is a combination of Narada, Intruder and Omen, which are sequential test-based approaches. The sequential tests are generated by feedback-directed random test generation (e.g., Randoop [16]). AutoConTest is also a coverage-based approach yet is specialized in atomicity violation detection.

B. EXPERIMENTAL SUBJECTS

We select 20 thread-safe classes as our experimental subjects. Table 1 presents their detailed information, i.e., class id (column 1), class name (column 2), lines of code (column 3), number of methods (column 4), number of method pairs (column 5) and bug type (column 6). The max size in lines of code is 3080. The number of methods is from 8 in HashSet to 217 in XYPlot.

The initial number of method pairs is calculated by (7),

$$f(n) = C_n^2 + n \tag{7}$$

where n indicates the number of methods. In this paper, we calculate the combination of any two methods rather than the permutation, because the concurrent test cases generated for (m_1, m_2) and (m_2, m_1) are equivalent for bug detection. Moreover, (m_i, m_i) $(i = 1, 2, 3, \ldots, n)$ can also construct a concurrent method pair.

TABLE 1. Experimental subjects.

ID	Class	LoC	#Meth.	#CMPs	Bug
1	LinkedList(JDK1.1)	78	10	55	Race
2	SynchronizedMap	79	15	120	Deadl.
3	XYSeries	200	25	325	Race
4	HashSet	230	8	36	Atom.
5	BufferedInputStream	239	9	45	Atom.
6	Day	267	26	351	Race
7	TimeSeries	359	41	861	Race
8	ArrayList	498	19	171	Race
9	Logger	531	44	990	Atom.
10	SharedPoolDataSource	546	51	1,326	Race
11	LinkedList(JDK1.5)	656	39	780	Race
12	PerUserPoolDataSource	719	65	2,145	Race
13	StringBuffer(JDK1.6)	789	52	1,378	Atom.
14	Vector(JDK1.5)	889	41	861	Atom.
15	XStream	926	66	2,211	Race
16	ConcurrentHashMap	972	22	253	Atom.
17	StringBuffer(JDK1.0)	1075	32	528	Atom.
18	NumberAxis	1,662	110	6,105	Atom.
19	PeriodAxis	1,975	125	7,875	Race
20	XYPlot	3,080	217	23,653	Race

TABLE 2. The time required to find bugs with BC-ConTest and the existing approaches.

Class	Time to bugs (seconds, avg.)				
	BC-ConTest	CovCon	ConTeGe	Nainom	AutoConTest
LinkedList(JDK1.1)	**665**	**1815**	**2680**	**3600**	**3600**
SynchronizedMap	296	861	3600	3600	-
XYSeries	66	30.88	132	603	3600
HashSet	8	26	50	286	167
BufferedInputStream	**2**	**22**	**123**	**103**	**690**
Day	683	805	1568	2601	3600
TimeSeries	141	185	223	605	3600
ArrayList	**3**	**9**	**29**	**151**	**3600**
Logger	**32**	**327**	**2769**	**3600**	**3600**
SharedPoolDataSource	**26**	**71**	**1494**	**3600**	**-**
LinkedList(JDK1.5)	**2**	**9**	**27**	**200**	**3600**
PerUserPoolDataSource	**16**	**80**	**2394**	**3600**	**-**
StringBuffer(JDK1.6)	1026	891	1215	598	-
Vector(JDK1.5)	262	1180	654	589	492
XStream	665	1674	389	3600	-
ConcurrentHashMap	**714**	**2084**	**2200**	**1004**	**3600**
StringBuffer(JDK1.0)	20	64	196	338	137
NumberAxis	85	91	602	3600	186
PeriodAxis	115	259	225	3600	-
XYPlot	**331**	**1239**	**2356**	**3600**	**-**

C. EXPERIMENTAL RESULTS AND ANALYSIS

1) COMPARISON WITH THE STATE-OF-THE-ART APPROACHES ON BUG FINDING CAPABILITY

Bug finding capabilities reflect that whether the approaches can find the bugs in CUTs. Table 2 presents, for each class, the class name (Column 1) and the time required to find the

bugs in them with five approaches (Column 2-6). ''3600'' and ''-'' represent time-out and tool crash, respectively.

The results in Table 2 show that only BC-ConTest and CovCon can successfully find all concurrency bugs in 20 CUTs within one hour. ConTeGe cannot find the concurrency bugs in SynchronizedMap because of time-out. Its random selection makes it consume too much time to repeatedly test behaviors that have been exercised and miss the chance to test behaviors that can expose concurrency bugs. Nainom cannot find the concurrency bugs in nine CUTs due to time-out. There are two reasons for this: 1) its static inference, combination and validation operations reduce the efficiency of finding bugs, 2) the pre-given sequential tests limit its effectiveness of bug detection. AutoConTest is inferior to other approaches in bug finding capacities since it is just designed for detecting atomicity violations.

2) COMPARISON WITH THE STATE-OF-THE-ART APPROACHES ON EFFICIENCY

Efficiency is represented by the time required to find the concurrency bugs in thread-safety classes. In Table 2, the significant efficiency improvements of our approach compared with other approaches are marked by printing the time in bold. From the data in Table 2, we can conclude that BC-ConTest can generate tests and find concurrency bugs in 257.9s on average, which is at least 2x faster than CovCon (586.14s). AutoConTest occurs the situations of both time-out and the tool crash, which implies that it is not general enough for multiple kinds of concurrency bugs.

BC-ConTest combines bug-driven and coverage-guided techniques to generate concurrent test cases for thread-safe classes. The method pair with the highest priority is selected in each iteration, which implies that we always try to generate concurrent test cases to find the concurrency bugs with the maximum possibility. Thus, it can find the concurrency bugs in all CUTs efficiently. For example, for PerUserPoolDataSource and SharedPoolDataSource, BC-ConTest is 149x and 57x faster than ConTeGe.

For XYSeries and StringBuffer(JDK1.6), the time consumed by BC-ConTest is slightly more than that with CovCon. The reason is that, the concurrency bugs are found by the concurrent test cases generated for the method pairs in the group of low priority. Specifically, in StringBuffer, there is no method pairs in which common variables are accessed with at least a write access and are protected by common locks. Similar to CovCon, the prioritization strategy is less successful for XStream in steering testing toward higher interleaving coverage. That is why BC-ConTest is slower than ConTeGe for it.

Next, we focus on using statistical tests to explore the difference between BC-ConTest and CovCon on efficiency since they have the same bug finding capacities. We ran each approach 10 times for each program. Fig. 6 shows the time distribution required to trigger concurrency bugs in 20 CUTs in 200 runs. From Fig. 6 we can find the red boxes are much shorter than the green boxes, which implies that the

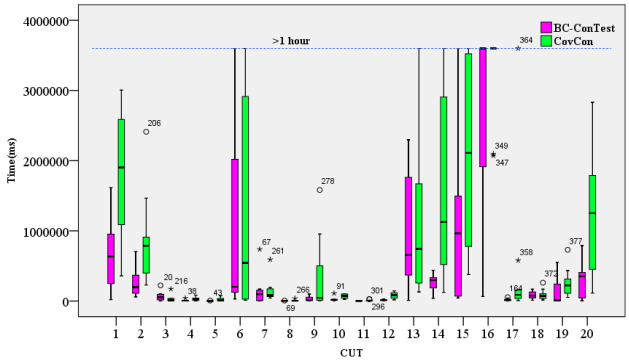


FIGURE 6. Time required to trigger the concurrency bugs in 20 CUTs in 200 runs.

time distribution with BC-ConTest is much concentrated. Moreover, Fig. 6 can clearly show that the time required to find the concurrency bugs by BC-ConTest is much less than that by CovCon in total 200 runs.

In order to prove the significance of the difference between BC-ConTest and CovCon on efficiency, we calculated the p-Value (a number of p-Value below 0.05 is considered statistically significant) using two tailed Mann-Whitney U test and Wilcoxon W test. The test result shows that the p-Value of two approaches is 0.00088, which is much less than 0.05. This indicates that there is a significant difference between BC-ConTest and CovCon.

In addition, Cohen's d was also applied to measure the effect size between two approaches. The calculation of Cohen's d is presented in (8):

$$Cohen's\ d = \frac{M_1 - M_2}{\sqrt{(\sigma_1^2 + \sigma_2^2)/2}} \tag{8}$$

M_1 and M_2 represent the means of the experimental and control groups. σ_1 and σ_2 represent the standard deviations of the experimental and control groups.

Table 3 presents the statistical test results of BC-ConTest and CovCon, including the mean values, variances, standard deviations and Cohen's d. The Cohen's d result of BC-ConTest and CovCon is -0.338.

To sum up, we can conclude that BC-ConTest significantly performs better than the state-of-the-art CovCon on efficiency.

3) COMPARISON ON BUG-DRIVEN GROUPING OF METHOD PAIRS

Both CovCon and BC-ConTest adopt the coverage-guided technique. CovCon selects a method pair from all the method pairs which are constituted by any two methods. Unlike CovCon, BC-ConTest uses bug-driven grouping of method pairs to divide the initial method pairs into two groups with two priorities. To improve the efficiency of concurrency bug detection, method pairs from the group with a high priority are firstly selected. Only if all the method pairs in it have been

TABLE 3. Statistical test results of BC-ConTest and CovCon.

Approach	Mean	Variance	StdDev	Cohen's *d*
BC-ConTest	3.943E+5	7.558E+11	8.694E+5	
CovCon	7.374E+5	1.298E+12	1.139E+6	
BC-ConTest vs. CovCon				-0.338

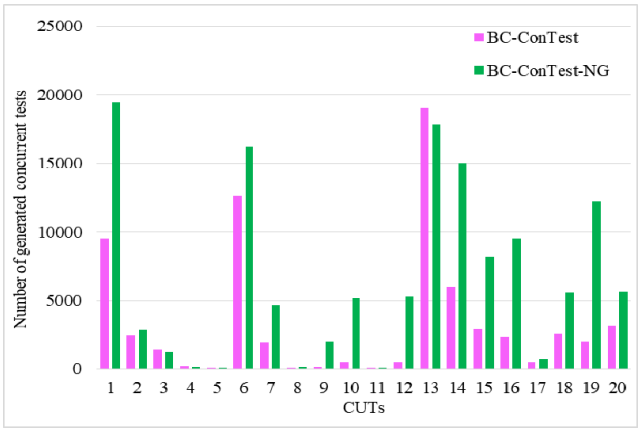


FIGURE 7. Number of generated concurrent test cases for 20 CUTs with BC-ConTest and BC-ConTest-NG.

selected yet not find concurrency bugs, method pairs from the group with a low priority are selected.

To validate the effectiveness and efficiency of our bug-driven grouping of method pairs, we compared BC-ConTest with BC-ConTest-NG on the number of generated concurrent tests and the time required to find concurrency bugs. Here, BC-ConTest-NG means to conduct BC-ConTest without grouping of method pairs.

The comparison results of BC-ConTest and BC-ConTest-NG for 20 CUTs are presented in Fig. 7 and Fig. 8. Fig. 7 shows the number of generated concurrent test cases with BC-ConTest and BC-ConTest-NG. Overall, BC-ConTest can find the concurrency bugs with much less concurrent tests than BC-ConTest-NG. Fig. 8 shows the time required to find the concurrency bugs in 20 CUTs. The bars in Fig. 8 are basically coincided with those in Fig. 7. The more concurrent test cases it generated, the more time it required to find the concurrency bugs. For BufferedInputStream (ID = 5) and Logger (ID = 9), BC-ConTest can reduced the time cost by 91.24% and 90.03%, respectively.

According to the data we collected, we know that BC-ConTest reduced the number of generated concurrent test cases and time cost by 47.94% and 61.04%, respectively. This reduction further validates the efficiency of our approach and the contribution of bug-driven grouping of method pairs.

Contrary to the most CUTs, we find that BC-ConTest generates more concurrent test cases and consumes more time than BC-ConTest-NG for StringBuffer (ID = 13). As we explained in the last section, the concurrency bug in StringBuffer(JDK1.6) was found by the concurrent test generated for the method pair in *LowPrioMethPairs*. Therefore,

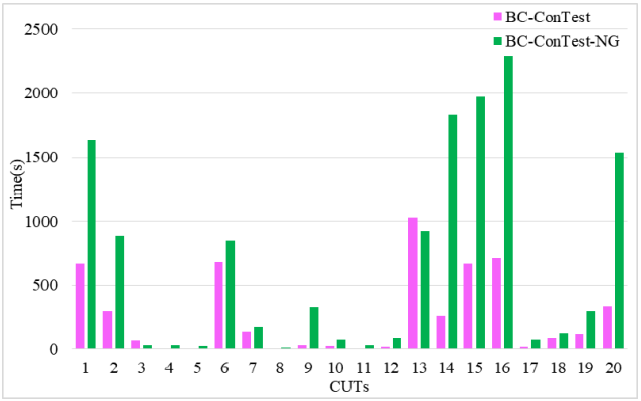


FIGURE 8. Time required to find concurrency bugs in 20 CUTs with BC-ConTest and BC-ConTest-NG.

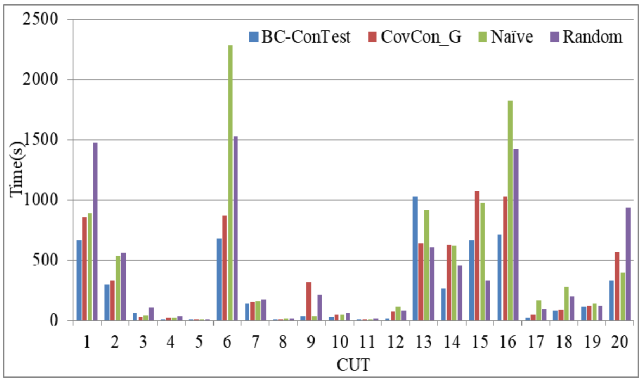


FIGURE 9. Comparison results of time cost with four prioritization strategies.

it consumed an amount of time to search for method pairs to generate concurrent test cases and to detect concurrency bugs.

4) COMPARISON WITH OTHER PRIORITIZATION STRATEGIES ON COVERAGE-GUIDED GENERATION OF CONCURRENT TEST CASES

The goal of this study is to validate the efficiency of different prioritization strategies on method pair selection. To complete this goal, we compared our coverage-guided prioritization strategy with that used in CovCon, Naïve prioritization and random selection. Naïve prioritization means to select the method pair that has been tried the least number of times. To avoid biasing, four prioritization strategies are applied to the grouped method pairs.

Fig. 9 shows the comparison results of time cost with four prioritization strategies, i.e., BC-ConTest, CovCon_G, naïve prioritization and random selection. CovCon_G means to conduct the prioritization strategy of CovCon in two grouped method pairs.

The results in Fig. 9 show that, 1) our prioritization strategy performs much better than naïve prioritization and random selection, 2) it is better or comparable to CovCon. In addition, we can see that, there are 18 CUTs whose time cost with BC-ConTest is less than that with CovCon_G. This implies

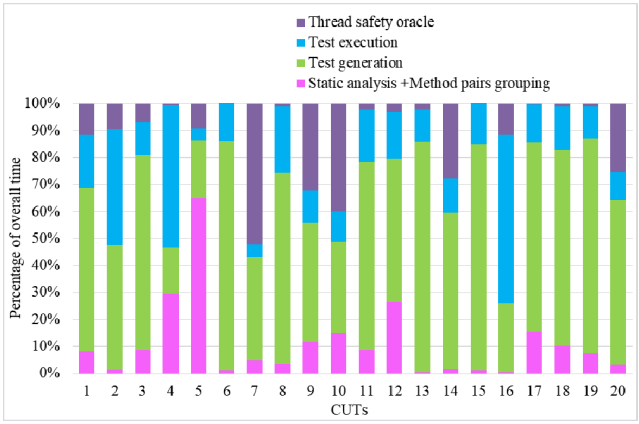


FIGURE 10. Breakdown of overall execution time of BC-ConTest.

that our prioritization strategy has good guidance for method pairs selection for most thread-safe classes. For example, for XStream (ID = 15), BC-ConTest reduced the time by 38.08% than CovCon_G.

5) BREAKDOWN OF OVERALL EXECUTION TIME

To explore how much time BC-ConTest spends on static analysis and grouping of method pairs, we recorded the time spent in different parts of BC-ConTest. Fig. 10 shows the breakdown of overall execution times for 20 thread-safe classes. As the method pairs are grouped along with static analysis, we viewed them as a whole and computed the time.

From Fig. 10, we find that most of the time is used for test generation. This indicates that it is very important to adopt some measures to speed up test generation, such as prioritization strategy. In addition, although we conduct static analysis and method pairs grouping, they occupy only a small proportion of the overall time, i.e., 11.34% on average.

Different from other classes, for BufferedInputStream (ID = 15), static analysis and method pairs grouping occupy the most part of overall time (65%). Two main reasons are for this difference: 1) There are only nine methods in BufferedInputStream, which is smaller than that in other classes. What's more, there are only six method pairs in the group with high priority. The concurrency bug is just found when execute the concurrent test generated for the method pair in the group with high priority. 2) The overall execution time for BufferedInputStream is only 2.2s, most of which is used to write the analysis results to files.

D. THREATS TO VALIDITY

Although the experimental results show the superior efficiency of our approach over the state-of-the-art approaches, there are several threats to the validity of our experiments. They can be summarized into two aspects.

1) INTERNAL VALIDITY

Static analysis may be the threat to the internal validity. There are many challenges in static analysis, such as alias analysis and flow/context sensitivity. For efficiency, our static analysis is based on the flow-insensitive and context-insensitive points-to analysis, which may affect the precision. To avoid false positives from static analysis, we just remove the method pairs in which both are synchronized methods. Then only part of the remaining method pairs are firstly selected. If the concurrency bugs are not found, the second group will be considered. The AutoRT proposed by Schimmel *et al.* [2] also used a single static analysis to identify methods that cannot be executed in parallel for improving concurrent test generation. It reduced method pairs following four requirements. In contrast to BC-ConTest, AutoRT only focuses on the method pairs that are relevant for data race detection. BC-ConTest is more generalized, which can detect many kinds of concurrency bugs. Next, we will reference to the requirements in AutoRT and use a more precise points-to analysis with flow-sensitivity [17] as well as context-sensitivity [18], [19] to enhance the effectiveness of our approach.

In addition, as our static analysis is implemented based on Soot, a Java program analysis framework, it only works for Java classes. We plan to extend our static analysis to other kinds of classes with LLVM [20].

2) EXTERNAL VALIDITY

The scale and quality of classes under test may be the threat to the external validity. We conduct experiments on 20 thread-safe classes. Although our results are satisfying, we cannot ensure that our conclusions could hold for all classes. Therefore, we would search for more open source thread-safe classes for future experiments.

IV. RELATED WORK

In recent years, test case generation for concurrent programs has drawn much attention from researchers. According to the methodology of concurrent test generation, we classify the existing approaches into three categories, i.e., random-based approaches, sequential test-based approaches and coverage-based approaches.

Random-based approaches randomly select some methods of CUTs and then schedule them by random combined threads. Burckhardt *et al.* [4] randomly selected constructors to create shared objects and then randomly selected their methods to be scheduled by multiple concurrent threads. To reduce the time of exploring false thread interleaving, Nistor *et al.* [5] minimized concurrent degree to two threads without sacrificing bug finding capacities and proposed Ballerina. In addition, to overcome the high false positives of existing approaches, Pradel and Gross [6] presented ConTeGe. ConTeGe generated concurrent tests based on random selection and gave thread safety oracle to detect thread-safe bugs. Random-based approaches have the minimum test requirements. However, they are inefficient as they may repeatedly test the behaviors that have been exercised and miss those that can expose bugs. Our approach significantly reduced the randomness by using bug-driven grouping of

method pairs and coverage-guided generation of concurrent tests. Both techniques can steer automatic test generation toward tests that are likely to trigger concurrency bugs and to cover not-yet-covered interleavings. Therefore, our approach is efficient in detecting concurrency bugs.

To alleviate the inefficient problem from random selection, Ma *et al.* [21] focused on generating a series of concurrent test cases in an adaptive manner based on the idea of diversity [22], [23]. First, two diversity metrics were presented from a static metric and a dynamic perspective, respectively. Then, three adaptive test generation approaches for C/C++ concurrent data structures were implemented to expose more interleaving instances with less computation time and fewer tests. But one limitation is that they payed less attention to its test oracle problem. Our approach checks whether executing a concurrent test can expose a concurrency bug based on the thread safety oracle [6]. It reports concurrency bugs if the exception occurs only in a concurrent execution but not in any of linearization of methods.

Sequential test-based approaches analyze the execution traces of the sequential tests, identify concurrency bugs that may occur when combining multiple sequential tests into concurrent tests and finally synthesize such tests. Based on this idea, Samak and Ramanathan [7], [8] and Samak *et al.* [9], [24] proposed a series of concurrent test generation approaches to detect different kinds of concurrency bugs. For example, Omen [7] and Omen+ [25] were designed to detect deadlocks. Narada [9] and Intruder [8] were developed to detect data races and atomicity violations, respectively. Minion [24] was created to detect assertion violations. Sequential test-based approaches do not generate the redundant concurrent tests that are irrelevant to the considered type of concurrency bugs. However, the effectiveness of sequential test-based approaches depends on the pre-given suite of sequential tests. Moreover, this kind of approach may produce an expensive computing cost. In contrast, our approach does not need any extra tests and generates concurrent tests for any two feasible methods in CUTs. Therefore, it is both conceptually and computationally simple, yet effective in finding concurrency bugs.

Coverage-based approaches compute a set of coverage requirements and automatically generate tests to coverage as many requirements as possible. Steenbuck and Fraser [10] defined parameterized coverage criterion, estimated all the feasible coverage requirements and guided test generation to cover these requirements. However, for concurrent programs, it is impractical to statically collect the whole coverage requirements. Terragni and Cheung [11] computed coverage requirements dynamically in the process of exploring method invocations to drive concurrent test generation. They implemented AutoConTest, which significantly improved the performance of test generation. However, it can only detect atomicity violations. Recently, Choudhary *et al.* [12] employed concurrent method pairs to be the approximate interleaving coverage metric. The metric is both efficient and practice to use. Bianchi *et al.* [26] exploited a set of pruning strategies to steer the test code generation towards tests that are likely to reproduce failures. Although our approach also adopts concurrent method pairs as the coverage metric in the process of concurrent test generation, there are two differences. First, our approach leverages bug-driven techniques to divide the feasible method pairs into two groups. One group is given a high priority, and the other group is given a low priority. Method pairs in the high priority group are first selected for generating concurrent tests. Second, a new method pairs prioritization strategy is designed for selecting a method pair in each iteration.

In addition, there is a large body of research involved in exploring the thread interleaving space to find concurrency bugs. Commonly, they are based on a certain interleaving coverage criterion [27]–[30]. Lu *et al.* [28] summarized a concurrent program interleaving coverage criteria hierarchy, in which all the criteria are designed based on different concurrency fault models. Hong *et al.* [29] presented a novel thread-scheduling technique in terms of synchronization-pair coverage criterion for achieving high test coverage. Later on, they conducted a comprehensive empirical investigation of the concurrency coverage metrics effective for testing [30]. From another perspective, Yu *et al.* [31] defined a coverage metric based on a set of interleaving idioms. An interleaving idiom represented a pattern of inter-thread dependencies and indicated a *happens-before* relation between conflicting memory accesses by different threads.

Different from all the above coverage criteria, our approach generates unit test cases for thread-safe classes based on an approximate interleaving coverage metric, i.e., concurrent method pairs. This coverage criterion is first proposed by Deng *et al.* [13] for reducing the dynamic bug-detection cost for predefined inputs. In this paper, we use it to reduce the computation and time cost of concurrent test generation.

V. CONCLUSION

In this paper, we present an efficient test generation approach for thread-safe classes. We first adopt static analysis to remove the method pairs that cannot execute concurrently. Then, we group the remaining method pairs based on the bug-driven techniques. Finally, we iteratively generate concurrent test cases with the coverage-guided techniques and execute them with thread safety oracle to find the concurrency bugs. Our experimental results show that, BC-ConTest can obtain a significant improvement on efficiency without impairing bug finding capacities. In the future, we will combine active testing [32]–[35] to further improve the efficiency of concurrency bug detection. In addition, we plan to extend the proposed approach to the massage-passing parallel programs [36], [37] to improve its generalization.

ACKNOWLEDGMENT

The authors would like to thank editors and anonymous reviewers for their valuable comments and useful suggestions. They would also like to thank all the individuals who

participated and contributed to improve the quality and readability of this paper.

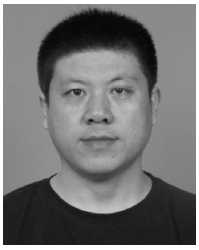
REFERENCES

[1] V. Terragni and M. Pezzè, "Effectiveness and challenges in generating concurrent tests for thread-safe classes," in *Proc. ASE*, Montpellier, France, Sep. 2018, pp. 64–75.
[2] J. Schimmel, K. Molitorisz, A. Jannesari, and W. F. Tichy, "Automatic generation of parallel unit tests," in *Proc. AST*, San Francisco, CA, USA, 2013, pp. 40–46.
[3] A. Jannesari and F. Wolf, "Automatic generation of unit tests for correlated variables in parallel programs," *Int. J. Parallel Program.*, vol. 44, no. 3, pp. 644–662, Mar. 2015.
[4] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan, "Line-up: A complete and automatic linearizability checker," in *Proc. PLDI*, Toronto, ON, Canada, 2010, pp. 330–340.
[5] A. Nistor, Q. Z. Luo, M. Pradel, T. R. Gross, and D. Marinov, "Ballerina: Automatic generation and clustering of efficient random unit tests for multithreaded code," in *Proc. ICSE*, Zurich, Switzerland, 2012, pp. 727–737.
[6] M. Pradel and T. R. Gross, "Fully automatic and precise detection of thread safety violations," in *Proc. PLDI*, Beijing, China, 2012, pp. 521–530.
[7] M. Samak and M. K. Ramanathan, "Multithreaded test synthesis for deadlock detection," in *Proc. OOPSLA*, Portland, OR, USA, 2014, pp. 473–489.
[8] M. Samak and M. K. Ramanathan, "Synthesizing tests for detecting atomicity violations," in *Proc. FSE*, Bergamo, Italy, 2015, pp. 131–142.
[9] M. Samak, M. K. Ramanathan, and S. Jagannathan, "Synthesizing racy tests," in *Proc. PLDI*, Portland, OR, USA, 2015, pp. 175–185.
[10] S. Steenbuck and G. Fraser, "Generating unit tests for concurrent classes," in *Proc. ICST*, Luxembourg, 2013, pp. 144–153.
[11] V. Terragni and S.-C. Cheung, "Coverage-driven test code generation for concurrent classes," in *Proc. ICSE*, Austin, TX, USA, 2016, pp. 1121–1132.
[12] A. Choudhary, S. Lu, and M. Pradel, "Efficient detection of thread safety violations via coverage-guided generation of concurrent tests," in *Proc. ICSE*, Buenos Aires, Argentina, 2017, pp. 266–277.
[13] D. Deng, W. Zhang, and S. Lu, "Efficient concurrency-bug detection across inputs," in *Proc. OOPSLA*, Indianapolis, IN, USA, 2013, pp. 785–802.
[14] S. Lu, S. Park, E. Seo, and Y. Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *Proc. ASPLOS*, Mar. 2008, pp. 329–339.
[15] L. Qi, W. Dou, W. Wang, G. Li, H. Yu, and S. Wan, "Dynamic mobile crowdsourcing selection for electricity load forecasting," *IEEE Access*, vol. 6, pp. 46926–46937, 2018.
[16] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proc. ICSE*, Minneapolis, MN, USA, 2007, pp. 75–84.
[17] B. Hardekopf and C. Lin, "Flow-sensitive pointer analysis for millions of lines of code," in *Proc. CGO*, Chamonix, France, 2011, pp. 289–298.
[18] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras, "Introspective analysis: Context-sensitivity, across the board," in *Proc. PLDI*, Edinburgh, U.K., 2014, pp. 485–495.
[19] C. Vassallo *et al.*, "Context is king: The developer perspective on the usage of static analysis tools," in *Proc. SANER*, Beijing, China, Mar. 2018, pp. 38–49.
[20] *The LLVM Compiler Infrastructure*. Accessed: Dec. 21, 2018. [Online]. Available: http://llvm.org/
[21] L. Ma, P. Wu, and T. Y. Chen, "Diversity driven adaptive test generation for concurrent data structures," *Inf. Softw. Technol.*, vol. 103, pp. 162–173, Jul. 2018.
[22] T. Y. Chen, F.-C. Kuo, D. Towey, and Z. Zhou, "A revisit of three studies related to random testing," *Sci. China Inf. Sci.*, vol. 58, no. 5, pp. 1–9, May 2015.
[23] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. H. Tse, "Adaptive random testing: The ART of test case diversity," *J. Syst. Softw.*, vol. 83, no. 1, pp. 60–66, Mar. 2009.
[24] M. Samak, O. Tripp, and M. K. Ramanathan, "Directed synthesis of failing concurrent executions," in *Proc. OOPSLA*, Amsterdam, The Netherlands, 2016, pp. 430–446.
[25] M. Samak and M. K. Ramanathan, "Omen+: A precise dynamic deadlock detector for multithreaded Java libraries," in *Proc. FSE*, Hong Kong, 2014, pp. 735–738.
[26] F. A. Bianchi, M. Pezzè, and V. Terragni, "Reproducing concurrency failures from crash stacks," in *Proc. ESEC/FSE*, Paderborn, Germany, 2017, pp. 705–716.
[27] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur, "Applications of synchronization coverage," in *Proc. PPoPP*, Chicago, IL, USA, 2005, pp. 206–212.
[28] S. Lu, W. Jiang, and Y. Zhou, "A study of interleaving coverage criteria," in *Proc. FSE*, Dubrovnik, Croatia, 2007, pp. 533–536.
[29] S. Hong, J. Ahn, S. Park, M. Kim, and M. J. Harrold, "Testing concurrent programs to achieve high synchronization coverage," in *Proc. ISSTA*, Minneapolis, MN, USA, 2012, pp. 210–220.
[30] S. Hong, M. Staats, J. Ahn, M. Kim, and G. Rothermel, "Are concurrency coverage metrics effective for testing: A comprehensive empirical investigation," *Softw. Test., Verification Reliabil.*, vol. 35, pp. 334–370, Jun. 2014.
[31] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, "Maple: A coverage-driven testing tool for multithreaded programs," in *Proc. OOPLSA*, Tucson, AZ, USA, 2012, pp. 485–502.
[32] P. Joshi, M. Naik, C.-S. Park, and K. Sen, "CalFuzzer: An extensible active testing framework for concurrent programs," in *Proc. CAV*, Grenoble, France, 2009, pp. 675–681.
[33] C. S. Park and K. Sen, "Randomized active atomicity violation detection in concurrent programs," in *Proc. FSE*, Atlanta, GA, USA, 2008, pp. 135–145.
[34] K. Sen, "Race directed random testing of concurrent programs," in *Proc. PLDI*, Tucson, AZ, USA, 2008, pp. 11–21.
[35] H. Yue, P. Wu, T. Y. Chen, and Y. Lv, "Input-driven active testing of multi-threaded programs," in *Proc. APSEC*, New Delhi, India, 2015, pp. 246–253.
[36] T. Tian and D. Gong, "Test data generation for path coverage of message-passing parallel programs based on co-evolutionary genetic algorithms," *Auto. Softw. Eng.*, vol. 23, no. 3, pp. 469–500, Nov. 2014.
[37] D. Gong, C. Zhang, T. Tian, and Z. Li, "Reducing scheduling sequences of message-passing parallel programs," *Inf. Softw. Technol.*, vol. 80, pp. 217–230, Sep. 2016.

LILI BO is currently pursuing the Ph.D. degree with the School of Computer Science and Technology, China University of Mining and Technology. Her research interests include software analysis and testing.

SHUJUAN JIANG received the Ph.D. degree from Southeast University, in 2006. She is currently a Professor and a Ph.D. Supervisor with the School of Computer Science and Technology, China University of Mining and Technology. Her research interests include compilation techniques and software engineering.

JUNYAN QIAN is currently a Professor and a Ph.D. Supervisor with the Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology. His research interests include software engineering, model checking, and program verification.



RONGCUN WANG received the Ph.D. degree from the Huazhong University of Science and Technology, in 2015. He is currently an Assistant Professor with the School of Computer Science and Technology, China University of Mining and Technology. His research interests include software testing, software maintenance, and machine learning.

XINGYA WANG received the Ph.D. degree from the China University of Mining and Technology, in 2017. He is currently holding a Postdoctoral position with the State Key Laboratory for Novel Software Technology, Nanjing University. His research interests include software testing and program debugging.