SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

Progress of Projects Supported by NSFC

December 2012 Vol. 55 No. 12: 2774–2784
doi: 10.1007/s11432-012-4751-z

An efficient method for detecting concurrency errors in object-oriented programs

HE YanXiang[1,2], WU Wei[2]* & CHEN Yong[2]

[1]*School of Computers, Wuhan University, Wuhan* 430072*, China;*
[2]*State Key Laboratory of Software Engineering, Wuhan University, Wuhan* 430072*, China*

Received October 7, 2012; accepted November 21, 2012

Abstract Multicore and multi-threaded processors have become the norm for modern processors. Accordingly, concurrent programs have become more and more prevalent despite being difficult to write and understand. Although errors are highly likely to appear in concurrent code, conventional error detection methods such as model checking, theorem proving, and code analysis do not scale smoothly to concurrent programs. Testing is an indispensable technique for detecting concurrency errors, but it involves a great deal of manual work and is inefficient. This paper presents an automatic method for detecting concurrency errors in classes in object-oriented languages. The method uses a heuristic algorithm to automatically generate test cases that can effectively trigger errors. Then, each test case is executed automatically and a fast method is adopted to identify the actual concurrency error from anomalous run results. We have implemented a prototype of the method and applied it to some typical Java classes. Evaluation shows that our method is more effective and faster than previous work.

Keywords concurrency error, dynamic test, data race, atomicity violations, test case generation

Citation He Y X, Wu W, Chen Y. An efficient method for detecting concurrency errors in object-oriented programs. Sci China Inf Sci, 2012, 55: 2774–2784, doi: 10.1007/s11432-012-4751-z

1 Introduction

Owing to the increased availability of multi-core machines, concurrent programs, specifically multi-threaded and multi-processor programs on shared memory machines, have become increasingly important and prevalent in the past few years. A study performed at Microsoft reports that over 60% of developers face concurrency issues and most of the respondents handle concurrency errors on a monthly basis [1]. Not only do concurrency errors reduce the productivity of developers, but they can cause serious problems and disasters, such as the Northeastern blackout [2] and the Therac-25 accident [3], which were caused by race conditions. Concurrency errors are difficult to find because they occur in a specific interleaving of memory-access sequences. The non-deterministic behavior of concurrent programs exacerbates reproducing the specific interleaving [4]. Monitoring and investigating all memory accesses is practically impossible because the number of possible interleavings increases exponentially with the number of threads [5]. To detect such concurrency errors, researchers have focused on finding those involving a single shared variable. Early work has detected data race conditions [6] that, without synchronization, can result in problematic interleaving. Recently, researchers have investigated other important classes of

*Corresponding author (email: whuwuwei@126.com)


```
class Queue {
  vector⟨int⟩items;
  int qsize;
  Queue() {
    items =...
    qsize = 0;
  }
  synchronized pop() {
   if (qsize == 0) return null;
   qsize–;
   return items[qsize];
  }
  synchronized push(int item) {
   items.add(item);
   qsize++;
  }
  synchronized size() {
   return size;
  }
  getqueuetop() {
    if (qsize == 0) return null;
    return items[qsize];
  }}
```

Figure 1 Source code of an example class.

concurrency errors, such as order violations [7] and single-variable atomicity violations [8]. Although several existing techniques can successfully find such errors, their ability is quite limited, and most of these are inefficient and tend to produce false alarms.

This paper deals with the problem of finding thread-unsafe classes in object-oriented programs. Many object-oriented programming languages support multiple threads, so it is common for a class instance to be used in multiple threads. However, if a class is thread-unsafe, it may cause concurrency errors such as data races, order violations, and atomicity violations in a multi-threaded environment. Writing a thread-safe class is difficult because the programmer does not know how the class will be used in other programs. For example, considering the class Queue given in Figure 1, although every operation that modifies the member variables is synchronized, the class is still thread-unsafe as explained in Subsection 3.2. In this paper, we present an automatic dynamic approach for detecting class concurrency errors. Like a unit test, our approach requires test cases of the class, which are then executed to detect an error. Traditional testing techniques cannot detect concurrency errors efficiently. First, because real world concurrent programs have a vast interleaving space, a large number of test cases are required to check a large portion of this space. Creating these test cases by hand is both time consuming and requires much manpower. Second, the test cases need to be executed one by one and the results checked to see whether the run has exposed a concurrency error. Unlike a sequential program where the same input results in the same output, a concurrent program's output is different every time it executes owing to the non-deterministic interleaving. So a test case needs to be executed many times to detect errors inevitably skipped in other production runs. To address the above problems, our method first uses an efficient algorithm to generate high quality test cases aimed at concurrency errors. Then, it uses an automatic execution engine to execute the test cases and an algorithm to detect the concurrency errors by analyzing the execution results. In summary, this paper makes the following contributions:

1) A heuristic method for automatically generating test cases for concurrency errors. There are some existing methods for generating test cases automatically; however, most of these aim to find sequential

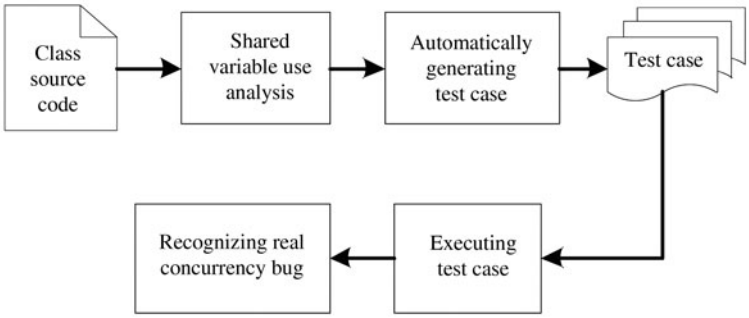

Figure 2 Overview of the proposed method.

errors. Several methods focusing on concurrency errors simply use a random method to generate test cases. Although a random method is an important tool for dynamically detecting errors, it is fairly aimless and inefficient. Usually, there are some concurrency errors that cannot be detected except by executing a specific test case. Thus, we present a heuristic method that can generate a test case set with high interleaving coverage. Our method uses variable access patterns and synchronization information, both of which have not, to the best of our knowledge, been used before to guide test case generation.

2) A novel efficient technique for detecting concurrency errors by analyzing dynamic execution results. Failure of a test case indicates that there is an error in the program. However, not all errors are concurrency errors. The error may be caused by the incorrect use of a class. Because we are focusing on concurrency errors, an efficient method for recognizing real concurrency errors is needed. Previous methods execute all possible linearizations of a test case when analyzing abnormal execution to ascertain whether there is an equivalent sequential execution. Obviously, this takes much time. Instead of running the linearization of a test case every time, we execute it only once before running the test case and record the execution result. If execution of the test case fails, we can decide whether it is a concurrency error by comparing it to the recorded result.

The rest of this paper is organized as follows. Section 2 gives an overview of our method. Section 3 describes the techniques for class code analysis and test case generation, and explains the core algorithm. Section 4 discusses how the test case is executed to detect concurrency errors. Section 5 presents our experiments, which show that our method is precise and efficient compared to existing approaches. Section 6 discusses related work, while Section 7 presents our conclusions and future work.

2 Overview

This section gives a brief description of our method, which employs dynamic analysis to detect concurrency errors in classes of an object-oriented programming language. In general, three steps are used to detect concurrency errors in classes using dynamic analysis. First, we design a concurrency test case using the tested class that may run into a state exposing the error. Next, we execute the test case with different code interleaving to expose the error. Finally, we analyze the execution result and decide whether the error in execution is a real concurrency error. Various approaches for solving the second step have been proposed [9,10]. Until recently, there have been some studies that have addressed the first and last steps [11]; however, these works are inefficient and require a great deal of time to detect real errors. In practical software development, since the code changes frequently, a fast test method is needed; as such, previous methods are not suitable. To improve the efficiency and accuracy, we present a new method for automatically generating test cases, and a new method to recognize real concurrency errors.

The key parts and operations in our method for deciding whether a class is threadsafe are illustrated in Figure 2. The method consists of four main processes. Given the code for a tested class, we analyze the use patterns of the class' member variables by the class' member functions in the first step. In this step, we determine which of the class' member variables are shared variables and which functions read from or write to them. In addition, we determine whether a function is synchronized by the key word

"synchronized". After obtaining the variable use and synchronization information, we use a heuristic method to automatically generate test cases that can expose unsafe thread errors with a high probability. Using the automatically generated test cases instead of relying on existing test cases has many advantages. First, it automates our approach making it easier to use. Second, it can generate many different test cases with high coverage, something that would require considerable time and effort if done manually. Finally, we use a heuristic generation method which can generate test cases that can expose errors not triggered by manually created ones. Next, we execute the generated test cases. Unlike the execution of sequential programs, which is almost solely determined by the input, both the execution and error triggering of concurrent programs are greatly affected by the non-deterministic interleaving of concurrent execution components (threads or processes). As a result, we may need to run a test case many times to trigger an error. Since the class is supposed to be thread-safe, it should synchronize concurrent calls on the shared instance as needed. As a result, the methods should "behave as if they occur in some serial order that is consistent with the order of the method calls made by each of the individual threads involved" [12]. When an abnormal execution of the test case occurs, we need to analyze the exception report and decide whether the concurrent execution shows behavior not possible in any sequential execution. If it does, a concurrent error is reported. Otherwise, we need to run the test case again or generate a new test case. To identify a real concurrency error, we need to explore all possible sequential executions, of which there is a huge number. Thus, we present an improved method that does not run a linearization schedule of test cases, thereby reducing the error reorganization time.

3 Class code analysis and test case generation

Given a class to be tested, we first analyze the class code to gain some useful information, which can guide the test case generation. In fact, concurrency errors are typically caused by abnormal access to shared variables. When a class instance is used by multiple threads, the public member variables can be considered to be shared variables. Hence, the trigger for a concurrency error is strongly related to which variables the class member functions can access and how the class' public member functions are called in multiple threads. The underlying theory of our test case generation algorithm is based on the above observations. This section presents the class code analysis and test case generation techniques in detail.

3.1 Class code analysis

In object-oriented languages, a representative class consists of various member variables and functions. A class is instantiated by the constructor function and then the member functions can be called by the instance. When a class instance is used in a multi-threaded program, the class member variables can be accessed by different threads at the same time. This is likely to cause a concurrency error if the class does not have any synchronization mechanisms. Consequently, many classes are declared threadunsafe in Java, for example. In essence, the reason for concurrency errors such as order violation, atomicity violation, and data race conditions is abnormal access to shared variables. So we can take advantage of the variable access pattern and synchronization information of the class code to generate better test cases.

Let $\mathcal{C}$ represent a class, then $\mathcal{V}_\mathcal{C}$ and $F_\mathcal{C}$ denote sets of class member variables and functions, respectively, which are defined as follows:

Definition 1 (Variable set). $\mathcal{V}_\mathcal{C} = \{v | v \text{ is a member variable of } \mathcal{C}\}$.

Definition 2 (Public function set). $F_\mathcal{C} = \{f | f \text{ is a public member function of } \mathcal{C}\}$.

For the sake of simplicity, in the remainder of this paper, $\mathcal{V}_\mathcal{C}$ is abbreviated as $\mathcal{V}$ and $F_\mathcal{C}$ as F for a specific class.

Analysis of the class code consists of two steps. First, we analyze every function $f \in F$. If function f is not declared with the key word "synchronized", it is not executed atomically. We obtain the set of non-atomic functions denoted as $\mathcal{S}_f$. Because the execution of a non-atomic function can be interrupted by another thread, the existence of a non-atomic function is a necessary condition for the occurrence of

a non-deadlock concurrency error. Thus, if $\mathcal{S}_f = \emptyset$ can be established, the class is thread-safe. For the example class given in Figure 1, the member function set F= {Queue, pop, push, size, getqueuetop}, and the non-atomic function set $\mathcal{S}_f$= {queuetop}.

Second, we analyze how the class member functions access the member variables. For every variable v in $\mathcal{V}$, we need to find the functions that read v and functions that can change the value of v. To record the results of the analysis, we use two maps $\mathcal{M}_r : \mathcal{V} \to 2^F$ and $\mathcal{M}_w : \mathcal{V} \to 2^F$ to save each variable's read and write related functions, respectively. $\mathcal{M}_r(v)$ denotes all the class member functions that read v, while $\mathcal{M}_w(v)$ denotes all the class member functions that write to v. For the example class given in Figure 1, there are two member variables: items and qsize. It is easy to see that $\mathcal{M}_r$(qsize) = {pop, size, getqueuetop} and $\mathcal{M}_w$(qsize) = {pop, push}. For a complex class, the member functions may call each other and recursive function calls may also exist. So computing $\mathcal{M}_r$ and $\mathcal{M}_w$ is an iterative process. Let f_r represent the set of variables read by function f. Then $\mathcal{M}_r(v)$= $\{f|v \in f_r\}$, and therefore, we can compute f_r to get $\mathcal{M}_r$. The iterative process for computing f_r is shown below.

$$\begin{cases} f_r^0 = \emptyset, \quad f \in F, \\ f_r^{i+1} = \{v|v \text{ appears as right value in } f\} \cup f_{rf' \in f_F}^{'i}. \end{cases}$$

In the above formula, f_F denotes the set of functions called by f. The iterative computational process terminates when f_r^{i+1} is the same as f_r^i. We can obtain $\mathcal{M}_w$ in a similar way to $\mathcal{M}_r$.

Analysis of the class source code is necessary to generate high quality test cases in the next step. The analysis process can be implemented effectively and easily using the compiler front end. This is feasible even if analysis of the class code is done manually.

3.2 Test case generation

After analyzing the class code, we use an efficient method to generate concurrent test cases that use the class in multiple threads. Input to the method is the class member variable set $\mathcal{V}$, two variable-function maps $\mathcal{M}_r$ and $\mathcal{M}_w$, and the function set $\mathcal{S}_f$. The important output of the method is class call sequences. A class call sequence $(c_1, c_2, c_3, \ldots, c_n)$ is a sequence of the invocations of the functions in the class. Each c_i in a class call sequence consists of a function signature, a list of input variables, and an output variable. Because a class call sequence is a component of the test case, it must be correctly formatted. We say that a class call sequence has good format if each input variable of c_i is a numerical value or the output variable c_j with $i < j$, that is, every c_i has the correct input parameters.

Similar to [11], a test case consists of a sequential part and a number of parallel parts. The sequential part first instantiates the class and calls methods on the class instance to "grow" the object, that is, to bring it into a state that may allow the parallel part to trigger an error. Each parallel part consists of a call sequence supposed to be executed concurrently with other parallel parts after execution of the sequential part. All parallel parts share the output variables of the sequential part and can use them as input variables for the parallel calls. In particular, all parallel parts share the class instance created in the sequential part. While our method can create any number of parallel parts, we only create two parallel parts here, that is, each of our generated test cases consists of two threads. Thus, each test case can be represented by the triple (p, c_1, c_2) where p is the sequential part and c_1 and c_2 are parallel parts. The underlying reason for this choice is that most real-world concurrency errors involve no more than two threads [13]. An example test case for the example class given in Figure 1 is shown in Figure 3. The sequential part of the test case first instantiates the class by calling the constructor. Each of the two suffixes calls a single method using the shared class instance "tc" as the receiver. In practice, effective concurrent test cases are not always that simple. Calling a method often requires providing parameters of a particular type, which in turn may require calling other methods. Furthermore, triggering an error often requires bringing the class instance into a particular state, for example, by calling setter methods.

Figure 4 describes the algorithm for our test case generation method. Input for the algorithm is the analysis results from the previous step. Every time the algorithm is run, a test case instance is returned. Next we explain each step of the algorithm.

```
1 Queue tc = queue();
2 tc.push(1);
3 Thread t1 = new Thread {
4           public void run() { tc.pop();}};
5 Thread t2 = new Thread {
6           public void run(){tc.getqueuetop();}};
7 t1.start(); t2.start();
8 t1.join(); t2.join();
```

Figure 3 A simple example test case generated by our method.

Algorithm 1 Generate concurrent test case

Name: gettest

Input: $\mathcal{V}$: class member variable sets

$\mathcal{M}_r$: map of variable to functions reading it

$\mathcal{M}_w$: map of variable to functions writing it

$\mathcal{S}_f$: set of functions that are synchronized

Output: test case

```
1  while true do
2     v ←randselect(V)
3     if |(M_r(v) ∪ M_w(v)) ∩ S_f| = 0 then continue endif
4     f ←randselect((M_r(v) ∪ M_w(v)) ∩ S_f)
5     p ←instanceclass()
6     for j ←MaxpLength do
7        p ← p⊕randselclasscall()
8     c1 ← f
9     c2 ←randselect(M_r(v) ∪ M_w(v))
10    for j ←MaxcLength do
11       c2 ← c2⊕randselect(M_w(v))
12       c2 ← c2⊕randselect(M_r(v))
13 return (p, c1, c2)
```

Figure 4 Test case generation algorithm.

Initially, the algorithm randomly selects a variable from the class member variable set $\mathcal{V}$, and checks whether all of the functions that access the selected variable are synchronized. If there are no non-atomic functions that access the selected variable, the algorithm randomly selects a variable again until it reaches the condition (lines 1 to 3). After selecting a variable, the algorithm obtains the set of non-atomic functions that access the selected variable from $\mathcal{M}_r, \mathcal{M}_w$, and $\mathcal{S}_f$, and then randomly selects a function from the set (line 4).

The next step in the algorithm involves creating the sequential part of the test case (lines 5 to 7). The operator $\oplus$ in our algorithm is used to concatenate sequences. It first creates a statement to instantiate the class, and then creates a statement by randomly selecting a function. The statement is appended to the created part. The algorithm repeats this process until it reaches the limit for the maximum length of the sequential part. Next, the algorithm generates the parallel parts (lines 8 to 12). During this step, the two parallel parts are created differently. The function selected in line 4 serves as one of the parallel parts. The other parallel part is created by an iterative process. The algorithm first randomly selects a function from the set of functions that access the previously selected variable. Then it appends two functions, which are randomly selected from the set of functions that read the selected variable and the set of functions that write to the selected variable, respectively. After the number of iterations has reached the limit for the maximum length of the parallel part, this step terminates. Finally, the algorithm returns the newly generated test case. The test cases generated by Algorithm 1 are used as input for

testing a class without any manual effort, and hence, increase the usefulness of existing dynamic analysis methods.

4 Test execution and concurrency error detection

After generating a test case, we need to execute the test case to detect any concurrency errors in the class. If there are no concurrency errors in the class, we say that the class is thread-safe. In this section we explain the process of detecting a concurrency error by dynamically running the test case. In general, sufficient test cases are automatically created by Algorithm 1, which when executed, can expose a concurrency error. However, because not all the errors are caused by thread interleaving, we need to check the exception execution results to decide whether it is a real concurrency error. How to determine effectively whether abnormal execution is a real error is a key problem that is solved by our method. A fast decision method can speed up the error detection process and improve the detection accuracy. Before introducing our algorithm, we present some basic concepts and definitions.

A class is said to be thread-safe if multiple threads can use it without synchronization and if the behavior observed by each thread is equivalent to a linearization of all calls that maintains the order of calls in each thread [12]. If all the member functions of a class are thread-safe or have no concurrency errors, the class is thread-safe. The test case in Figure 3 is an example of a thread-unsafe program. Since function "getqueuetop" can be interrupted by thread 1, this test case is an example of the thread-unsafe property of class Queue. A property related to thread safety is atomicity, that is, the guarantee that a sequence of operations performed by a thread appears to execute without interleaved operations by other threads. One way to make a class thread-safe is to guarantee that each call to one of its methods appears to be atomic for the calling thread. However, a thread-safe class does not guarantee that multiple calls to a shared instance of the class are executed atomically [14]. Even so, the concept of atomicity is a good foundation for deciding whether an abnormal execution exposes a concurrency error. When the execution of a test case causes an exception, we compare it to executions of the linearization of the test case. If the same exception has not been caused by executions of the linearization of the test case, we say that this test case exposes a concurrency error in the class. Like [11], we use the following definition to explain our algorithm.

Definition 3 (Interleaving)**.** For a pair of code segments (c_1, c_2), $I(c_1, c_2)$ is the interleaving space of c_1 and c_2, defined as follows:

$$I(c_1, c_2) = \{c_{12} | \forall c.c \in c_{12} \leftrightarrow c \in c_1 \cup c_2 \wedge \forall c, c', (c \prec_{c1} c' \rightarrow c \prec_{c12} c') \wedge (c \prec_{c2} c' \rightarrow c \prec_{c12} c')\}.$$

We use $c \prec_{c1} c'$ to indicate that statement c executes before statement c' in code sequence $c1$. In fact, the interleaving of sequence c_1 and c_2 is a combination of c_1 and c_2 in a way that preserves the related order of statements in c_1 and c_2

Definition 4 (Linearization)**.** For a test case (p, c_1, c_2), we use $L(p, c_1, c_2)$ to denote the set of linearizations of the test case.

$$L(p, c_1, c_2) = \{p \oplus c | c \in I(c_1, c_2)\}.$$

Definition 5 (Thread safety)**.** If all test cases of a class execute concurrently like the linearization of the test, we say the class is thread-safe. This can be expressed by the following formula:

$$\text{thread-safe}(C) \Leftrightarrow \forall t \in T_C \quad \forall e_t \in E(t) \quad \exists l \in L(t) \rightarrow e_t \cong e_l.$$

In the above formula, predicate thread-safe is true implies that a class is thread-safe; T_C indicates all the possible test cases for class C; $E(t)$ denotes the set of all distinguishable executions of test case t; e_l gives the execution of sequence l; and $e_t \cong e_l$ indicates that execution e_t is equivalent to execution e_l. If two executions e_1 and e_2 both terminate without exceptions or deadlocks, or with the same exception, we say that e_1 and e_2 are equivalent, which is expressed as $e_1 \cong e_2$. According to Definition 3, if there is

Algorithm 2 Execute test case and detect concurrency error

Input: empty

Output: bug report

```
1  for i ←Maxtime do
2      t ←gettest(V, M_r, M_w, S_f)
3      S ←serialize(t)
4      E ←exceptionrecord(S)
5      for j ←MaxRun do
6          e ←execute(t)
7          if failed(e) ∧ e ∈ E then
8              return bug(e)
9          else
10             continue
```

Figure 5 Dynamic concurrency error detection algorithm.

a test cast that exposes behavior not possible with any linearization of the test case, we can conclude that the class being tested is thread-unsafe. To decide whether two executions are equivalent, we only need to compare the exceptions and deadlocks caused by the executions. Although this abstraction ignores many potential differences between executions, such as different return values of method calls, it is crucial to ensure that the analysis only reports a class as being thread-unsafe if this is definitely the case.

Algorithm 2 in Figure 5 gives the test case execution process and how to decide whether the execution exposes a real error. The algorithm repeatedly obtains a test case by Algorithm 1 and executes it until the maximum number of test iterations is reached, or until an error is found. To decide quickly whether there is a linearization that behaves in the same way as the execution of the test case, we keep the execution results of all executions of the distinguishable linearizations of the test case before executing the test case (lines 3 to 4). Method "serialize" returns all the linearizations of the test case while method "exceptionrecord" runs each linearization and records abnormal execution results. Because concurrency error triggering is greatly affected by non-deterministic interleaving, the algorithm executes the test case a sufficient number of times to detect the error (lines 5 to 8). If a test case execution fails, we compare the exception thrown at runtime with the record of execution results of linearization. If the same exception exists in the record, the algorithm continues to run the test case until it reaches the terminating condition. Otherwise, there is a real error, that is, the class is thread-unsafe, and the algorithm returns an error and exits.

Our method relies on two assumptions, which are true for most real-world classes. The first is that uncaught exceptions and deadlocks that occur in the concurrent use of a class but not in the sequential use thereof, are considered to be a problem. This assumption is in line with the commonly accepted definition of thread safety [12]. The second assumption is that sequential executions of a call sequence behave deterministically. As a result, despite our threadsafe decision method being sound, it is incomplete.

5 Experiments and evaluation

This section describes the experimental process to validate our method and presents a comparison of the results with the latest similar work, BALLERINA [15]. Since Java is used widely in many companies, we implemented a prototype of our method in Java. To realize automation, we took advantage of the dynamic characteristics of Java. We used a syntax directed method to generate test cases, and then adopted a reflection-based approach to execute the test cases. To achieve dynamic execution, we use the APIs in the library package tools.jar supplied in version 7 of the JDK. After generating the test case source code, we used method "compile" to translate the Java file to a Java class byte code file. We then used method "invoke" to dynamically execute the byte code file and method "debug" to obtain the execution results. Based on empirical estimation, we set parameters MaxpLength, Maxtime, MaxRun, and MaxcLength to 5, 500, 100, and 3, respectively. The experiment was executed on a dual-core machine

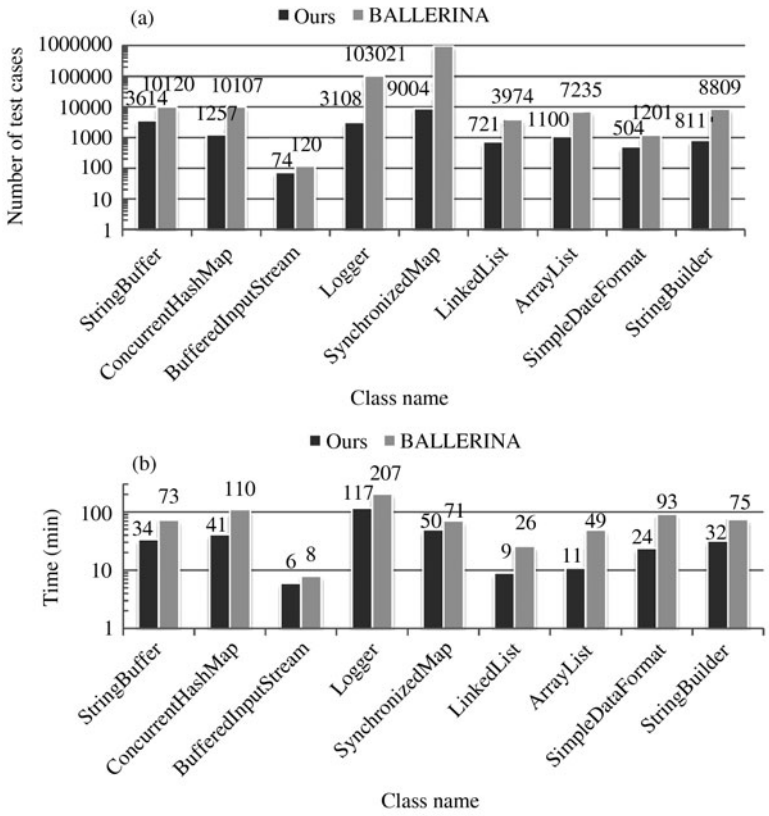

Figure 6 Experimental results. (a) Number of test cases generated before detecting an error; (b) time to trigger an error.

with 2 GHz Intel Duo processors and 2 GB memory running 32-bit Ubuntu Linux. We used the latest version of the JDK version 7.

To evaluate our method's effectiveness, we selected several typical thread-unsafe classes from the JDK and ran our analysis on these. The selected classes represent many important data structures such as maps, lists, and trees and are widely used in actual code. At the same time, we also ran BALLERINA on the selected classes and compared its efficiency in detecting errors with that of our method. When an error was detected, we recorded the number of test cases generated from the beginning and also the elapsed time. Because the numerical value of the result was different each time, we ran the analysis several times and took the average value as the final result. The experimental results are shown in Figure 6(a), where the vertical and horizontal axes show the number of test cases and the class name, respectively, shows the average number of test cases generated before an error was detected. In Figure 6(b), where the vertical axis shows the time duration, we compared the average time elapsed before an error was triggered. We can see from these results that our method successfully identifies all the classes as being thread-unsafe. In addition, our method generates fewer test cases and requires less time than BALLERINA.

6 Related work

There are several works dealing with detecting concurrency errors. These studies can be divided into two types, that is, static and dynamic testing methods. Model checking and code analysis are two typical static methods. Because static methods do not scale well when analyzing concurrent programs owing to the large interleaving space they need to explore, dynamic testing plays an important role in detecting concurrency errors. Since our method is a dynamic method, it can detect larger scale programs and is faster than static methods. In addition, as the proposed method integrates static analysis to generate test cases, it dramatically improves the efficiency and precision

Dynamic methods detect errors mainly by executing the code. There are many dynamic data race detectors, which search for unsynchronized conflicting accesses to shared data by analyzing happens-

before relations [6,16], checking whether the program follows a locking discipline [17], or through a combination of these techniques [18]. Refs. [19,20] present techniques for finding atomicity violations that rely on specifications of atomic blocks or sets of atomic variable accesses, respectively, provided manually or inferred heuristically. Naik et al. [21] searched for deadlocks statically but with the aid of tests. Joshi et al. [22] applied model checking to a reduced program with the aid of annotations of condition variables. All the above mentioned approaches can only detect one type of concurrency error, such as a data race, atomicity violation, or deadlock. Our method is a general approach suitable for detecting many common types of concurrency errors. By using an efficient method to decide whether a runtime exception is a real error, our method offers a further advantage in that it does not report false positives.

For concurrency program correctness criteria, Herlihy and Wing introduced linearizability as a correctness criterion for concurrent objects [23]. Line-Up [24] checks the linearizability of calls but requires manually specified method parameters. Whereas Line-Up executes all linearizations each time before running a concurrent test, our method only executes these once. Various studies have been conducted and are ongoing with respect to test case generation. Many works such as [25–27] focus on generating sequential test cases. In contrast, our method creates test cases for concurrent programs. Ballerina [15] generates efficient multi-threaded tests, showing that two threads, each with a single call, can trigger various concurrency errors. Pradel et al. [11] presented a method to automatically generate test cases for detecting thread safely violations and used a thread safety oracle to recognize real errors. Contrary to our work, the work by Pradel et al. does not use any heuristic strategy to guide the test case generation and it needs to execute all linearizations of the test cases several times.

7 Conclusions and future work

At present, many different CPUs have multi-core systems. Moreover, multi-core processors are used in most computer systems to improve system performance. It is difficult to program concurrent software for several reasons, including avoiding deadlocks, race conditions, and so on. Similar reasons make testing concurrent programs difficult. Existing testing methods for sequential programs are not adequate for testing concurrent programs. Therefore, in this paper, we presented an efficient method for detecting concurrency errors in object-oriented programs. In particular, we focused on the problem of guaranteeing the threadsafety of a class. Our method makes use of a heuristic algorithm to generate test cases that use the class in multiple threads in an intelligent way. Then it executes the test case and uses an efficient method to determine whether the execution exposes a real concurrency error. Our method involves very little human effort and produces only true positive error reports. We have validated our method by applying it to several Java classes and have obtained favorable experimental results.

Although our method is effective in detecting many concurrency errors, it can be improved in several aspects and this is what we intend doing in the future. Since most concurrency errors are caused by single-variable accesses, our method focuses on single-variable errors. However, the method can easily be modified to detect multi-variable errors. The test case generation process could use a more power heuristic strategy such as using feedback and profiling information to trigger errors faster. In our method, we obtain high interleaving coverage by executing the test case repeatedly. The number of different interleaving combinations during repeated executions of the same test can be increased either by using a deterministic scheduler or by injecting so-called noise into test executions. Thus, we can use deterministic schedulers or noise injection tools to systematically explore the interleaving space to a certain extent.

Acknowledgements

The work presented in this paper was supported in part by National Natural Science Foundation of China (Grant Nos. 90818018, 9111800).

References

1 Godefroid P, Nagappan N. Concurrency at Microsoft: an exploratory survey. In: Workshop on Exploiting Concurrency Efficiently and Correctly, Princeton, 2008
2 Poulsen K. Tracking the blackout bug Security Focus. 2004-04-07. http://www.securityfocus.com/news/8412
3 Leveson N G. SafeWare: System Safety and Computers. Boston: Addison-Wesley Professional, 1995
4 McDowell C E, Helmbold D P. Debugging concurrent programs. ACM Comput Surv, 1989, 4: 593–622
5 Musuvathi M, Qadeer S. Iterative context bounding for systematic testing of multithreaded programs. ACM SIGPLAN Not, 2007, 6: 446–455
6 Flanagan C, Freund S. FastTrack: efficient and precise dynamic race detection. ACM SIGPLAN Not, 2009, 44: 121–133
7 Park S, Vuduc R W, Harrold M J. Falcon: fault localization in concurrent programs. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, Cape Town, 2010. 245–254
8 Park S, Lu S, Zhou Y. CTrigger: exposing atomicity violation bugs from their hiding places. ACM SIGPLAN Not, 2009, 44: 25–36
9 Burckhardt S, Kothari P, Musuvathi M, et al. A randomized scheduler with probabilistic guarantees of finding bugs. ACM SIGPLAN Not, 2010, 45: 167–178
10 Coons K E, Burckhardt S, Musuvathi M. GAMBIT: effective unit testing for concurrency libraries. ACM SIGPLAN Not, 2010, 45: 15–24
11 Pradel M, Gross T R. Fully automatic and precise detection of thread safety violations. ACM SIGPLAN Not, 2012, 47: 521–530
12 Schildt H. Java 7 the Complete Reference. 8th Ed. New York: Mc-Graw Hill, 2011
13 Lu S, Park S, Seo E, et al. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. ACM SIGPLAN Not, 2008, 43: 329–339
14 Shacham O, Bronson N, Aiken A, et al. Testing atomicity of composed concurrent operations. ACM SIGPLAN Not, 2011, 46: 51–64
15 Nistor A, Luo Q, Pradel M, et al. Ballerina: automatic generation and clustering of efficient random unit tests for multithreaded code. In: ICSE 2012 Proceedings of the 2012 International Conference on Software Engineering, Piscataway, 2012. 727–737
16 Marino D, Musuvathi M, Narayanasamy S. LiteRace: effective sampling for lightweight data-race detection. ACM SIGPLAN Not, 2009, 44: 134–143
17 Praun C V, Gross T R. Object race detection. ACM SIGPLAN Not, 2001, 36: 70–82
18 Callahan R O, Choi J D. Hybrid dynamic data race detection. ACM SIGPLAN Not, 2003, 38: 167–178
19 Flanagan C, Freund S N, Yi J. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. ACM SIGPLAN Not, 2008, 43: 293–303
20 Lu S. Finding atomicity-violation bugs through unserializable interleaving testing. IEEE Trans Softw Eng, 2012, 38: 844–860
21 Naik M, Park C S, Sen K, et al. Effective static deadlock detection. In: ICSE '09 Proceedings of the 31st International Conference on Software Engineering, Vancouver, 2009. 386–396
22 Joshi S, Lahiri S K, Lal A. Underspecified harnesses and interleaved bugs. ACM SIGPLAN Not, 2012, 47: 19–30
23 Herlihy M, Wing J M. Linearizability: a correctness condition for concurrent objects. ACM Trans Program Lang Syst, 1990, 12: 463–492
24 Burckhardt S, Dern C, Musuvathi M, et al. Line-Up: a complete and automatic linearizability checker. ACM SIGPLAN Not, 2010, 45: 330–340
25 Godefroid P, Klarlund N, Sen K. DART: directed automated random testing. ACM SIGPLAN Not, 2005, 40: 213–223
26 Pacheco C, Lahiri S K, Ernst M D, et al. Feedback-directed random test generation. In: ICSE '07 Proceedings of the 29th international conference on Software Engineering, Minneapolis, 2007. 75–84
27 Krishnamoorthy S, Hsiao M S, Lingappan L. Strategies for scalable symbolic execution-driven test generation for programs. Sci China Inf Sci, 2011, 54: 1797–1812