Proposal of a Testing Method Using Similarity of Interleaving for Java Multi-threaded Programs

Shoichiro Kitano*, Tetsuro Katayama*, Yoshihiro Kita†,
Hisaaki Yamaba*, Kentaro Aburada‡ and Naonobu Okazaki*
**University of Miyazaki, 1-1 Gakuen-kibanadai nishi, Miyazaki, 889-2192 Japan*
†Kanagawa Institute of Technology, 1030 Shimo-ogino, Kanagawa, 243-0292 Japan
‡Oita National College of Technology, 1666 Maki, Oita, 870-0152 Japan
E-mail: kitano@earth.cs.miyazaki-u.ac.jp, kat@cs.miyazaki-u.ac.jp, y.kita@ccy.kanagawa-it.ac.jp, yamaba@cs.miyazaki-u.ac.jp, aburada@oita-ct.ac.jp, oka@cs.miyazaki-u.ac.jp

Abstract

In order to improve the efficiency of testing Java multi-threaded programs, this research proposes a testing method to detect order violation in them using similarity of interleaving. The proposed method improves the efficiency of testing by executing interleaving which can test the places where lead the order violation easily in source codes and by reducing interleaving which is similar to executed one already. The efficiency of the method is shown by experiments for confirmation.

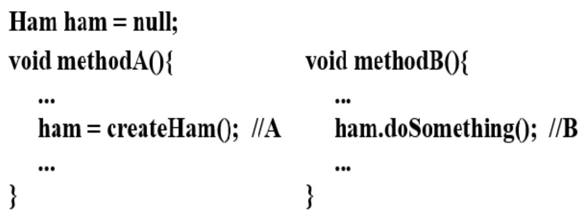
Keywords: multi-threaded program, testing, similarity, Java

1. Introduction

In recent years, many computers are adapted multi-core CPUs. In order to use such resources effectively, the demand of multi-threaded programs increases.

It is difficult for even expert programmer to implement multi-threaded programs. And it is easier to embed bugs than single-threaded programs[1]. There are some distinctive bugs in a concurrent program which are different from the bugs in a single-threaded program. Those bugs often appear in the latter of development process or when the program is used by the users. In this case, it is difficult to fix the detected bugs. In order to prevent this problem, we need to detect the bugs and to fix them in unit testing. However, normal unit testing cannot test multi-threaded programs enough. Unit testing tests often only single interleaving because executed tasks are too small in unit testing. Therefore, the bugs appear when the modules are integrated. One of the testing methods for multi-threaded programs in unit testing is to execute programs in the plural interleaving by staggering the timing of execution between each thread. That method can detect the existing bugs in multi-threaded programs. However, there is too large number of interleaving that we must execute and many interleaving cannot detect the distinctive bugs in a concurrent program. It takes the same result to test by the interleaving that is similar to tested one already and it is excessive testing. Moreover, each distinctive bug in a concurrent program has the own causes each other. It is not effective to detect all such bugs by an only testing method.

In order to improve the efficiency of testing for Java multi-threaded programs, this research proposes a testing method to detect order violation in them using similarity of interleaving. Our proposed method improves the efficiency of testing by executing interleaving which can test all places which lead the



```
Ham ham = null;
void methodA(){                 void methodB(){
    ...                             ...
    ham = createHam();  //A         ham.doSomething();  //B
    ...                             ...
}                               }
```

Fig. 1. An example of a source code which includes order violation.

Table 1. The number of threads
which involved to detect concurrency bugs.

Application	Total	> 2 threads	2 threads	1 threads
MySQL	22	1	17	4
Apache	17	0	17	0
Mozilla	56	1	54	1
OpenOffice	8	0	6	2
Overall	103	2	94	7

order violation easily in source codes and by reducing interleaving which is similar to executed one already.

2. The Kinds of Bugs That Our Proposal Method Can Detect

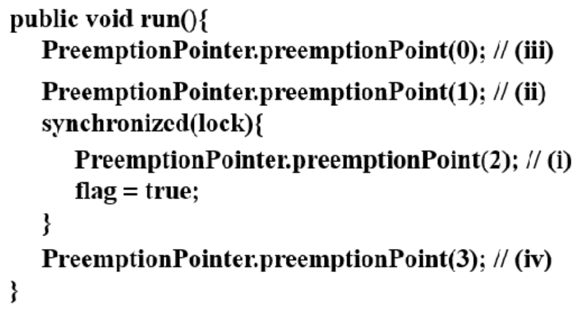
In this chapter, we explain the kinds of bugs that our proposed method targets.

2.1. *Target bug in multi-threaded programs*

The kinds of bug patterns in concurrency programs are classified[2]. The patterns are classified into dead lock, atomicity violation, order violation, and so on.

Order violation means that some threads can be executed as an access to a certain memory in an unexpected order. It occurs when a synchronization protocol for several threads is deficient. Fig.1 shows an example of a source code which includes order violation. Consider the case which methodA and methodB are executed in different threads each other. Statement B expects that ham has been initialized by statement A before statement B accesses to ham. However, this program can execute statement B before statement A. That leads the incident. The cause of this problem seems like atomicity of flag. However, Accessing to ham is unacceptable before ham has been initialized. Even if accessing to ham is serialized, the incident occurs because the initial value of ham is null.

In this research, we propose a testing method which can detect order violation as above.



```
public void run(){
    PreemptionPointer.preemptionPoint(0); // (iii)
    PreemptionPointer.preemptionPoint(1); // (ii)
    synchronized(lock){
        PreemptionPointer.preemptionPoint(2); // (i)
        flag = true;
    }
    PreemptionPointer.preemptionPoint(3); // (iv)
}
```

Fig. 2. An example of a source code which is inserted the probes.

2.2. *Number of thread used in testing*

In multi-threaded programs, the number of thread used in testing is important. Table 1 is a result of the research[2] which shows how many threads are necessary to detect the distinctive bugs in a concurrent program.

Table 1 shows that two threads can detect the most number of bugs. Hence, our proposed method uses two threads that can detect bugs the most effectively.

3. Proposal Method

3.1. *The four places which can lead the order violation easily*

In order to detect the distinctive bugs in a concurrent program, we have to enforce the testing which relates on timing of execution between threads. However, each bug pattern has own causes each other. Therefore, we need to enforce effective testing the each bug pattern.

In this research, we propose the four places which can lead order violation easily in source codes. We detect the order violation by executing the interleaving which can test the all places.

The places and reasons that we choose them are following.

(i) Substitution for a shared flag variable used for synchronization of threads
 This place can detect the bugs caused by incorrect place of statements that rewrite the value of a flag variable.
(ii) The entry point of synchronized blocks
 This place can detect the bugs caused by defectiveness of logic for a synchronization protocol implemented in a synchronized block.

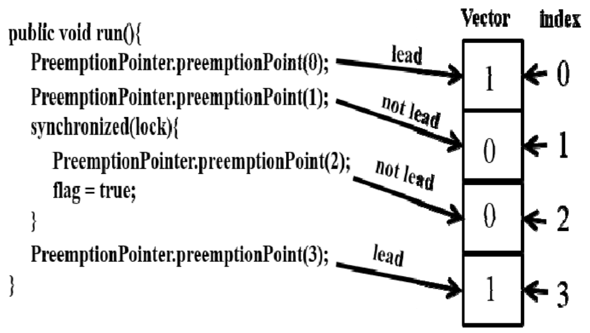

public void run(){
PreemptionPointer.preemptionPoint(0);
PreemptionPointer.preemptionPoint(1);
synchronized(lock){
PreemptionPointer.preemptionPoint(2);
flag = true;
}
PreemptionPointer.preemptionPoint(3);
}
lead
not lead
not lead
lead
Vector
index
1
0
0
1
0
1
2
3

Fig. 3. An example of a vector generation.

(iii) The start of execution of a thread
This place can detect the bugs caused when synchronization protocol is not implemented because programmers suppose that a program is executed in only one interleaving.

(iv) The end of execution of a thread
This place can detect the bugs caused by a logic which depends on the termination of other threads.

3.2. *Test codes*

In this research, we execute the interleaving which can test all places explained in section 3.1. Therefore, we describe the test code inserted some probes at the places in a source code of a test target program.

Fig.2 shows an example of the test code. "preemptionPoint(int number)" method which is member of "PreemptionPointer" class is probes. We explain integer values of arguments in the next section. This method can lead a context switch, as needed. This provides various interleaving which can test each place. The numbers in comments correspond to the numbers which are labeled each place in section 3.1.

3.3. *Reduction of number of test execution by using similarity of interleaving*

Our proposed method improves efficiency of testing by reducing interleaving which is similar to executed one already. It is necessary to change interleaving to the value which can be calculated similarity. Therefore, we generate vectors from the order and the number of probes in a test code. The values of the generated vectors are 1 or 0. 1 shows that the probe leads a context switch and 0 shows that the probe does not.

Fig.3 shows an example of vector generation. When the number of probes is n, the number of generated vectors is the n-th power of two. We treat this vector as interleaving and enforce testing each generated vector. The method "preemptionPoint(int number)" leads context switch to execute any interleaving by using generated vector. The integer number of arguments expresses index of elements that a vector has. "preemptionPoint(int number)" chooses to lead context switch or not to lead by using the elements at number of arguments. Here, the first index of elements is 0.

Then, we calculate the similarity between vectors. In this research, we use cosine similarity to calculate similarity between vectors. When the similarity between any vector and the vector which has used already is more than a threshold, "preemptionPoint(int number)" does not use that vector. This reduces the interleaving which leads the same result as the result of executed one already. Therefore, we can improve the efficiency of testing in multi-threaded programs.

Table 2. Result of experiments.

Number of Places	All or Reduced	Number of Testing	Number of Testing in Buggy Interleaving
4	All	16	7.4
4	Reduced	9	4.8
7	All	128	66.4
7	Reduced	59	27.9

4. Discussion

4.1. *Experiment for Confirmation*

We have conducted experiments to confirm efficiency of our proposed method.

The method in experiment is that we prepare two programs which include an order violation. And we execute the test codes that have some probes at places which can lead order violation easily in the source codes of those programs. They include four places and seven places in their source codes. We enforce two tests that use all generated vectors and reduce the vectors which are similar to vector which has used already. We confirm the efficiency of proposed method by comparing the number of testing that are result of each testing and by verifying that a bug can be detected when tests are reduced. Here, threshold used by deciding to reduce vectors is 0.8.

Table 2 shows the results of the experiments. Each number is the mean value of the result of executing the programs 10 times for each condition. "Number of Testing Buggy Interleaving" on the line of "All" shows that the proposed method can detect the bug definitely. Therefore, we confirmed that the proposed method can detect the order violation, which is one of the distinctive bugs in a concurrent program definitely.

"Number of Testing" shows that proposed method can reduce the number of each test execution by 56% and 46%. And "Number of Testing Buggy Interleaving" on the line of "Reduced" shows that the proposed method can execute the interleaving which can lead the bug definitely even if tests are reduced. These results show that the proposed method reduces the number of execution of testing by about 51% and can test to detect order violation enough. Therefore, we confirmed that proposed method can improve the efficiency of testing in multi-threaded programs.

4.2. *Related works*

Concuerror[3] is a testing tool for Erlang. Concuerror can test programs by executing plural interleaving and can reduce the interleaving by preemption bounding[4]. Therefore, Concuerror can enforce testing efficiency. However, executed interleaving is not focused on detecting the distinctive bugs in a concurrent program. Therefore, it has possibility not to detect such bugs.

Our proposed method executes the interleaving which can test all places which can lead order violation easily. Therefore, our proposed method can test more efficiently for the distinctive bugs in concurrent programs.

ConTest[5] is a testing tool for Java multi-threaded programs. ConTest can enforce testing that relates on timing by executing plural interleaving. However, it does not ensure to detect bugs definitely even if increasing the number of tests execution because it executes plural interleaving in an ad hoc basis. Therefore, the efficiency of its testing is poor.

Our proposed method executes the interleaving which can test the all places which can lead order violation easily. It reduces testing in the interleaving which is similar to executed one already and can detect order violation definitely. Therefore, our proposed method tests more efficiently.

5. Conclusion

In this research, we have proposed a testing method for order violation using similarity of interleaving. In order to realize it, we have defined four places which can lead order violation easily in source codes. The proposed method reduces the interleaving by changing interleaving to vectors which can be calculated and calculating the similarity between a vector and used vectors in testing already by cosine similarity. We conducted experiments that use the proposed method. The results show that the proposed method can detect the order violation definitely. And it can also detect the bugs even if it reduces the number of test execution by about 51%. That shows it is effective for improving the efficiency of testing in multi-threaded programs to use our proposed method.

Future issues are as following.

- Considering the testing methods which can detect other bug patterns
  Our proposed method is focused on only order violation. Therefore, it is not enough to detect other bug patterns like dead lock or atomicity violation. We need to consider the testing methods which can detect these bugs efficiently.
- Investigating the threshold of similarity for reducing the vectors
  Depending on the length of generated vector, there is a case that the number of generated vector which has similarity less than threshold is too large. In such a case, the number of vectors which is reduced is much the same as the number of vectors which are not reduced. We need to investigate the appropriate threshold which is based on length of the vectors.
- Consider the testing method which use more than two threads
  Our proposed method uses two threads for testing. Therefore, it cannot detect the bugs which occur in testing using more than two threads. We need to consider it.

References

1. J. K. Ousterhout, Why Threads Are A Bad Idea(Usenix Annual Technical Conference, San Diego, 1996) http://www.softpanorama.org/People/Ousterhout/Threads
2. S. Lu, S. Park, E. Seo and Y. Zhou, Learning from Mistakes - A Comprehensive Study on Real World Concurrency Bug Characteristics, (ASPLOS '08, Washington, 2008), pp. 329-339.
3. A. Gotovos, M. Christakis, and K. Sagonas, Test-DrivenDevelopment of Concurrent Programs using Concuerror, (Erlang'11, New York, 2011), pp.51-61.
4. M. Madan, Q. Shaz, Iterative Context Bounding for Systematic Testing of Multithreaded Programs, (PLDI '07, Sam Diego, 2007), pp.11-13.
5. ConTest - A Tool for Testing Multi-threaded Java Applications, https://www.research.ibm.com/haifa/projects/verification/contest/