Towards Metamorphic Testing of Concurrent Programs

Peng Wu
State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China, wp@ios.ac.cn

Hepeng Dai
School of Computer and Communication Engineering
University of Science and Technology Beijing
Beijing, China
daihepeng@xs.ustb.edu.cn

Tsong Yueh Chen
Department of Computer Science and Software Engineering
Swinburne University of Technology
Melbourne, Australia
tychen@swin.edu.au

Chang-ai Sun, *Member, IEEE*
School of Computer and Communication Engineering
University of Science and Technology Beijing
Beijing, China
casun@ustb.edu.cn

***Abstract*—Metamorphic testing (MT) is a promising technique to alleviate the oracle problem, which fist defines metamorphic relations (MRs) used to generate new test cases (i.e. follow-up test cases) from the original test cases (i.e. source test cases). Both source and follow-up test cases are executed and their results are verified against the relevant MRs.**

***Index Terms*—metamorphic testing, control test process, partition**

I. Introduction

Testing of a concurrent program, e.g., a multi-threaded program, is still a challenging task due to its nature of nondeterminism. Moreover, a concurrent program typically consists of multiple threads that cooperate closely to fulfill a functional assignment with high efficiency, e.g., computing, sorting, or searching a large volume of static or dynamic data. A test oracle [1] usually provides an exact mechanism of deciding whether an output produced by a program is correct or not. This oracle problem [2], [3] also arises for concurrent programs as it can be very difficult or expensive, if not impossible, to determine their expected outputs *a priori*.

Metamorphic testing (MT) [4], [5] has been a successful technique to tackle oracle problems in various domains. It first defines metamorphic relations (MRs) based on the necessary properties about the inputs and outputs of the program under test. Then, MRs are exploited in two ways: a new test case (i.e., a follow-up test case) can be generated from an original test case (i.e., a source test case) based on an MR; the outputs resulted by executing the source and follow-up test cases can be verified against the corresponding MR. If the MR is violated, the program under test is shown to be incorrect, as witnessed by the pair of the source and follow-up test cases.

The National Natural Science Foundation of China (under Grant No. 61872039), the Beijing Natural Science Foundation (Grant No. 4162040), the Aeronautical Science Foundation of China (Grant No. 2016ZD74004), and the Fundamental Research Funds for the Central Universities (Grant No. FRF-GF-17-B29).

Let us use a simple example to illustrate how MT works. For instance, consider the function $max(x, y)$ that returns the maximum value between two integers x and y. It enjoys a simple and obvious property that the order of the two parameters x and y shall not affect the outputs. This can be identified as the metamorphic relation $MR_{max} : max(x, y) = max(y, x)$, where (x, y) and (y, x) constitutes the source and the follow-up test case, respectively. Then, given a program $P(x, y)$ that implements the function $max(x, y)$, and a source test case $(1, 2)$, if the outputs of P running with this source test case and the follow-up test case $(2, 1)$ do not equal, i.e, $P(1, 2) \neq P(2, 1)$, a fault is detected in P.

In this paper we aim at investigating the applicability of metamorphic testing to concurrent programs. This is motivated by the observation that the functional requirement or user expectation to a concurrent program can be identified as metamorphic relations for verifying the outputs of its non-deterministic executions under different inputs.

The rest of the paper is organized as follows.

II. Metamorphic Testing of Concurrent Programs

A. Concurrent Programs

Consider a function $\text{TOP}(V, n)$ that returns in the ascending order the minimal n values in a list V, while V may contain duplicate elements. For a list of large size, a sequential search program may not implement this function in a highly efficient way. Many concurrent search structures have been proposed to take advantages of multi-processor architectures, which are already very popular nowadays. Concurrent implementations of the *priority queue* data structure are typically adopted to implement multi-threaded programs for the function $\text{TOP}(V, n)$. In this work, we use the 5 concurrent priority queue classes presented in the textbook [6]: SimpleLinear, SimpleTree, SequentialHeap, FineGrainedHeap, and SkipQueue.

The SimpleLinear class is an array-based bounded implementation, while the SimpleTree class is a tree-based one.

The SequentialHeap class is a coarse-grained implementation based on a unbounded heap, while the FineGrainedHeap class is a fine-grained one. These four classes all explicitly use locks for synchronization, while the SkipQueue class does not. It is based on a skiplist that uses atomic primitives (i.e., compareAndSet) for synchronization.

B. Metamorphic Relations

Let VW be the concatenation of lists V and W, and V^k the concatenation of k V's. Let $head(V)$ denote the first element of list V, and $tail(V)$ the list resulted by removing the first element of list V. Therefore, $V = head(V) :: tail(V)$. The higher-order function (map f V) applies the given function f to each element of list V, returning a list of the results in the same order, i.e., (map f V) = $f(head(V))$::(map f $tail(V)$).

Assume $\text{TOP}(V, n) = [y_1, \ldots, y_n]$ with $y_1 < \cdots < y_n$, where $y_1, \ldots, y_n$ are the minimal n values in list V. Then, metamorphic relations of $\text{TOP}(V, n)$ can be defined as follows.

1) Permutation: A permutation of V is a rearrangement of the elements of list V. Such rearrangement shall not affect the result of TOP. This property is identified as the following metamorphic relations, where MR_2 constitutes a special case of MR_1.

MR-1 $\text{TOP}(V', n) = \text{TOP}(V, n)$ for any permuation V' of V.
MR-2 (Commutative Law) $\text{TOP}(VW, n) = \text{TOP}(WV, n)$.

2) Insertion: Adding duplicate elements into list V shall not affect the result of TOP. This general property is identified as the following metamorphic relations, which choose duplicate elements from different perspectives.

MR-3 $\text{TOP}(V^k, n) = \text{TOP}(V, n)$ for any $k > 1$.
MR-4 $\text{TOP}(VV', n) = \text{TOP}(V, n)$ for any $V' \subseteq V$.
MR-5 $\text{TOP}(V[y], n)$=$\text{TOP}(V, n)$ for any y in $\text{TOP}(V, n)$.
MR-6 $\text{TOP}(VV_1 \cdots V_k, n) = \text{TOP}(V, n)$ for any $k \geq 1$, where $V_i \subseteq \text{TOP}(V, n)$ for every $1 \leq i \leq k$.
MR-7 $\text{TOP}(V[y_1, \ldots, y_n], n) = \text{TOP}(V, n)$.
MR-8 $\text{TOP}(V[y_n, \ldots, y_1], n) = \text{TOP}(V, n)$.
MR-9 $\text{TOP}(VW, n) = \text{TOP}(V, n)$ for any permutation W of $\text{TOP}(V, n)$.
MR-10 $\text{TOP}([y_1, \ldots, y_n]V, n) = \text{TOP}(V, n)$
MR-11 $\text{TOP}([y_n, \ldots, y_1]V, n) = \text{TOP}(V, n)$
MR-12 $\text{TOP}(WV, n) = \text{TOP}(V, n)$ for any permutation W of $\text{TOP}(V, n)$.

The following metamorphic relations add to list V fresh elements that are no greater than the last element y_n of $\text{TOP}(V, n)$. These elements shall occur in the follow-up result of TOP.

MR-13 $\text{TOP}(V[y], n) = [y_1, \ldots, y_i, y, y_{i+1}, \ldots, y_{n-1}]$ if $y \notin \text{TOP}(V, n)$ and there exists $1 \leq i < n$ such that $y_i < y < y_{i+1}$.
MR-14 $\text{TOP}(V[y'_1, \ldots, y'_k], n) = [y'_1, \ldots, y'_k, y_1, \ldots, y_{n-k}]$, where $k \leq n$ and $y'_1 < \cdots < y'_k < y_1$.

On the contrary, the following metamorphic relation *MR*-15 adds to list V fresh elements that are greater than y_n. This insertion shall not affect the result of TOP.

MR-15 $\text{TOP}(V[y''_1, \ldots, y''_k], n) = \text{TOP}(V, n)$, where $y_n < y''_1 \leq \cdots \leq y''_k$.

3) Deletion: Deleting an element from list V will change the result of TOP if it also occurs in $\text{TOP}(V, n)$. The follow-up result of TOP only differs from the original one in those elements deleted.

MR-16 Let $X = [x_1, \ldots, x_k]$ for arbitrary $k \geq 1$ elements $x_1, \ldots, x_k$, $V' = V \backslash X$, $X' = X \cap \text{TOP}(V, n)$.
- Then, $\text{TOP}(V, n) \backslash \text{TOP}(V', n) \subseteq X'$
- Specially, if $X' = \emptyset$, $\text{TOP}(V', n) = \text{TOP}(V, n)$;

where $V \backslash X$ removes from V only one occurrence of each element in X (if any).

4) Transformation: Transforming all the elements of list V will make the result of TOP transformed in the same way.

MR-17 TOP((map f V),n)=(map f TOP(V,n)). For example, $f(x) = x + c$ for any constant c.

5) Splitting: Suppose $V = V_1V_2$. If y is one of the minimal n values in list V, then it is also one of the minimal n values in list V_1 or V_2. This property is identified as the following metamorphic relation *MR*-18.

MR-18 $\text{TOP}(V, n) \subseteq \text{TOP}(V_1, n) \cup \text{TOP}(V_2, n)$.

If y is one of the minimal n values in list V_1, and also one of the minimal m values in list V_2, then it is one of the minimal $n+m$ values in list V. This property is identified as the following metamorphic relations, which differ only in m.

MR-19 $\text{TOP}(V_1, n) \cap \text{TOP}(V_2, n) \subseteq \text{TOP}(V, 2n)$.
MR-20 $\text{TOP}(V_1, n) \cap \text{TOP}(V_2, m) \subseteq \text{TOP}(V, n + m)$ for $1 \leq m < n$.
MR-21 $\text{TOP}(V_1, n) \cap \text{TOP}(V_2, m) \subseteq \text{TOP}(V, n+m)$ for $m > n$.

The concatenation of the results of TOP on V_1 and V_2 preserves the result of TOP on V. This property is identified as the following metamorphic relation *MR*-22.

MR-22 $\text{TOP}(V, n)$=$\text{TOP}(\text{TOP}(V_1, n)\text{TOP}(V_2, n), n)$.

6) Sublisting: $\text{TOP}(V, n)$ is a prefix of $\text{TOP}(V, m)$ if $n < m$, while $\text{TOP}(V, n)$ is an extension of $\text{TOP}(V, m)$ if $n > m$. Specially, we consider the boundary values of m in the following metamorphic relations.

MR-23 $\text{TOP}(V, n)$ is a prefix of $\text{TOP}(V, n + 1)$.
MR-24 $\text{TOP}(V, n)$ is a prefix of $\text{TOP}(V, m)$ if $m > n + 1$.
MR-25 $\text{TOP}(V, n)$ is an extension of $\text{TOP}(V, n - 1)$.
MR-26 $\text{TOP}(V, n)$ is an extension of $\text{TOP}(V, m)$ if $1 \leq m < n - 1$.

III. Empirical Study

In this section we present the research questions concerned in this work, and the case study on the concurrent implementations of $\text{TOP}(V, n)$. Then, we discuss the experimental results with detailed analysis.

A. Research Questions

RQ-1 How effective MT is at detecting faults of concurrent programs? What is the actual overhead for MT detecting these faults, in terms of time consumption and the number of test cases executed?

Fault-detection effectiveness and efficiency are key for evaluating the performance of a testing technique. In our study, we adopt three state-of-the-art concurrent priority queue classes to implement the function TOP(V, n) within various concurrent scenarios. Then, we apply mutation analysis to evaluate the performance of MT in detecting the faults seeded, especially the concurrency faults related to locks and atomic primitives.

RQ-2 What is the fault-detecting capability of an individual metamorphic relation in different concurrent scenarios? How are metamorphic relations relevant to programs faults, especially concurrency faults?

RQ-3 How many threads are sufficient for MT detecting a concurrency fault?

B. Experimental Results

1) Mutants: Mutation analysis [7] has been widely used to asses the adequacy of test suites and the effectiveness of testing techniques. Mutation operators are used to seed various faults into the program under test, and thus generate a set of variants (i.e. mutants). If a test case causes a mutant to behave differently to the program under test, then we say that the mutant is "killed" by the test case. The mutation score (MS) is used to measure how thoroughly a test suite "kills" the mutants. The MS is defined as:

$$MS(p, ts) = \frac{N_k}{N_m - N_e} \quad (1)$$

where p is the source program; ts is the test suite; N_k is the number of killed mutants; and N_e is the number of equivalent mutants whose behavior is always same as that of p.

We selected three state-of-the-art concurrent priority queue classes from [8] as the subject programs for our study: `SimpleLinear`, `SimpleTree`, and `SequentialHeap`. Those programs are written in Java language, and mainly implement two functions: adding elements with corresponding priorities to the data structure and removing elements from the data structure according to the priorities.

We have generated 312 traditional mutants for studied programs using automated mutant generator tool Mujava [9] and Major [10]. Besides, we artificially generated 8 concurrent mutants according to the concurrent mutation operators [11] Table I summarizes the basic information of the used programs and their mutants.

TABLE I
OBJECT PROGRAMS

Programs	LOC	Number of Traditional Mutants	Number of Concurrent Mutants
`SimpleLinear`	52	22	2
`SimpleTree`	85	52	4
`SequentialHeap`	131	240	2

2) Concurrent Scenarios: The selected programs have two main interfaces called `add` and `remove`, we thus constructed four scenarios: i) sequentially adding elements and sequentially removing elements; ii) sequentially adding elements and concurrently removing elements; iii) concurrently adding elements and sequentially removing elements; and iv) concurrently adding elements and concurrently removing elements. We used random testing technique to construct 5 test cases as source test cases, and each test case includes 10,000 elements then corresponding follow-up test cases were generated according to each MR. To make statistical results significant, each testing session was repeated 5 times with 5 different seeds.

3) Response to RQ-1: After the execution of all test cases and all MRs in different scenarios the values of MS are shown in Table

4) Response to RQ-2:

5) Response to RQ-3:

IV. Related Work

In this section, we describe related work of MT.

A. Metamorphic Testing

V. Conclusion

In our future work, we plan to conduct experiments on more real-life programs to further validate the effectiveness of MT, and identify the limitations of our approach.

Acknowledgment

References

[1] E. J. Weyuker, "On testing non-testable programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, 1982.

[2] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015.

[3] K. Patel and R. M. Hierons, "A mapping study on testing non-testable systems," *Software Quality Journal*, vol. 26, no. 4, pp. 1373–1413, 2018.

[4] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: A new approach for generating next test cases," Department of Computer Science , Hong Kong University of Science and Technology, Hong Kong, Tech. Rep. HKUST-CS98-01, Tech. Rep., 1998.

[5] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. Tse, and Z. Q. Zhou, "Metamorphic testing: A review of challenges and opportunities," *ACM Computing Surveys*, vol. 51, no. 1, p. 4, 2018.

[6] M. Herlihy and N. Shavit, "Priority queues," in *The Art of Multiprocessor Programming*. Elsevier, 2012, ch. 15, pp. 351–368.

[7] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[8] M. Herlihy and N. Shavit, *The art of multiprocessor programming*. Morgan Kaufmann, 2011.

[9] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "Mujava: an automated class mutation system," *Software Testing, Verification and Reliability*, vol. 15, no. 2, pp. 97–133, 2005.

[10] R. Just, B. Kurtz, and P. Ammann, "Inferring mutant utility from program context," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'17)*. ACM, 2017, pp. 284–294.

[11] J. S. Bradbury, J. R. Cordy, and J. Dingel, "Mutation operators for concurrent java (j2se 5.0)," in *Proceedings of the 2nd Workshop on Mutation Analysis, Co-located with the 17th International Symposium on Software Reliability Engineering (ISSRE'06)*, pp. 83–92.