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@sina.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

A concurrent program typically forks multiple threads that run concurrently to fulfill a functional task, e.g., computing, sorting, or searching a large volume of static or dynamic data with presumably high speedup. Different threads may interact with each other through shared variables, which require rigorously lock-based or lock-free synchronizations. An explicit lock can enforce a mutually exclusive access to a shared variable, while a lock-free primitive, such as *compare-and-swap*, can ensure an atomic access to a shared variable. Thus, an execution of a concurrent program would highly depend on the runtime thread scheduling, which is nondeterministic by nature. Therefore, it is still challenging to efficiently detect faults, especially concurrency faults, in multi-threaded concurrent programs. Furthermore, test oracles [1] play a fundamental role in determining the correctness of an output produced by a program under test (PUT). However, due to the vast possibilities of thread scheduling, it can be very difficult or expensive, if not impossible, to determine the expected output of a concurrent program *a priori*. This oracle problem [2], [3] arises for concurrent programs inevitably.

Metamorphic testing (MT) [4], [5] has been a successful technique to tackle oracle problems in various applications such as compilers [6], search engines [7], cybersecurity [8] and machine learning-based driverless cars [9]. MT relies on metamorphic relations (MRs) that are derived from the necessary properties or user expectations about the inputs and outputs of the program under test. For instance, consider the function $max(x, y)$ that returns the maximum value between x and y. Obviously, it is commutative, i.e., the order of the two parameters x and y does not affect its result. This property can be identified as a metamorphic relation $MR_{max} : max(x, y) = max(y, x)$, where (x, y) constitutes a source test case, and (y, x) is the follow-up test case resulted by directly swapping the two parameters of the source test case. Then, for a program $P_{max}(x, y)$ that implements the function $max(x, y)$, if the outputs of P_{max} under the source test case (x, y) and the follow-up test case (y, x) do not equal, i.e, $P_{max}(x, y) \neq P_{max}(y, x)$, a fault is thus detected in this program.

It can be seen that MRs are referred to in the two stages of MT: test case generation and test case execution. In the former stage, a follow-up test case can be generated, based on an MR, from an original test case (i.e., a source test case) with or without taking into account the output of the PUT under the source test case; while in the latter stage, the outputs of the PUT under both the source and the follow-up test cases can be inspected against the MR. If the MR is violated, then the PUT is shown to be incorrect, as witnessed by the executions of the source and follow-up test cases.

In this paper, we intend to investigate the fault detection effectiveness of a metamorphic relation in concurrent testing. On one hand, a concurrent program may fulfill a functional task in a more efficient manner than a correspondingly sequential one may, e.g., by taking advantages of multicore processors, while on the other hand, a sequential program can be regarded as a special concurrent program where all the threads run in the pre-specified sequential order. This suggests that metamorphic relations identified based on the functional

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).

requirements or user expectations of the task shall apply to any of its implementation, no matter which is sequential or concurrent. That is, these *functional* metamorphic relations may be exploited to validate the nondeterministic outputs of a concurrent program with multiple inputs. But it is not clear how their fault detection effectiveness in testing a concurrent implementation would differ from that in testing a sequential one, as multiple threads running concurrently may trigger more subtle behaviors in the underlying implementation. Moreover, concurrency faults related to synchronizations only occur in a concurrent implementation, while it is not clear how the metamorphic relations would take effect in detecting these concurrency faults.

Herein we will address these issues through an empirical study with the Top-k query problem, i.e., searching for the least or greatest k elements in a large data repository. It is a quite typical problem in big data processing. In this work we consider a collection of concurrent programs that utilize a concurrent implementation of priority queue as the underlying data structure to solve the query problem. Each concurrent program consists of multiple threads that share a concurrent priority queue object as the data repository, inserting an element into the queue or removing the least or greatest element from the queue concurrently. We also include correspondingly sequential programs, which insert and remove elements sequentially, as the baseline to demonstrate the effectiveness of the functional metamorphic relations in detecting ordinary and concurrency faults.

The rest of the paper is organized as follows.

II. Metamorphic Testing of Concurrent Programs

A. Concurrent Programs

Without loss of generality, let $\mathbb{D}$ be a value domain with a total order $\leq$ defined on it. Consider a search function $\textsc{Top}(V, k)$ that accepts a multiset V of values from $\mathbb{D}$ and returns the least distinct $k > 0$ values in V. Note that V may contain duplicate values, but $\textsc{Top}(V, k)$ may not. In case that the size of V is very large, it is rather time-consuming to validate the results of a program that implements $\textsc{Top}(V, k)$.

Apparently this function can be implemented in a straightforward way, i.e., all the values in V are stored in a data repository q (as shown in Algorithm 1), followed by repeatedly removing and collecting the least value in the current repository until k distinct values are collected (as shown in Algorithm 3). A *priority queue* object is typically utilized herein as the data repository for better search efficiency. Its add(x) method inserts value x into the repository, while its removeMin() method removes and returns the least value in the repository.

Concurrent data structures [10] have been developed to further take advantages of multi-processor architectures that are already very popular nowadays. A concurrent priority queue object can then be utilized as the data repository (q) to be shared by multiple threads, which concurrently insert values into the queue (as shown in Algorithm 2), or concurrently remove values from the queue (as shown in Algorithm 4). In Algorithms 2 and 4, the iterations in each **parallel-for** loop run concurrently with separate $t > 1$ threads.

Algorithm 1 sequentialAdd(V)

```
1: for all x in V do
2:     q.add(x);
3: end for
```

Algorithm 2 concurrentAdd(V)

```
1: Partition V into t disjoint parts V_1, ..., V_t;
2: parallel for i ← 1 to t do
3:     sequentialAdd(V_i);
4: end parallel for
```

Algorithm 3 sequentialRemove(k)

```
1: R ← ∅;
2: repeat
3:     x ← q.removeMin();
4:     if x ∉ R then
5:         R ← R ∪ {x};
6:     end if
7: until |R| = k
8: return R;
```

Algorithm 4 concurrentRemove(k)

```
1: R ← ∅;
2: parallel for i ← 1 to t do
3:     c_i ← 0;
4:     repeat
5:         x ← q.removeMin();
6:         if x ∉ R then
7:             R ← R ∪ {x};
8:             c_i ← c_i + 1;
9:         end if
10:    until c_i = k/t
11: end parallel for
12: return R;
```

Thus, there exist four categories of implementations for function $\textsc{Top}(V, k)$

1) SASR(V, k) sequentially insert values to the data repository, followed by sequentially removing the least values from the data repository, as shown in Algorithm 5.
2) SACR(V, k) sequentially insert values to the data repository, followed by concurrently removing the least values from the data repository, as shown in Algorithm 6.
3) CASR(V, k) concurrently insert values to the data repository, followed by sequentially removing the least values from the data repository, as shown in Algorithm 7.
4) CACR(V, k) concurrently insert values to the data repository, followed by concurrently removing the least values from the data repository, as shown in Algorithm 8.

Algorithm 5 SASR(V, k)

```
1: sequentialAdd(V);
2: return sequentialRemove(k);
```

Algorithm 6 SACR(V, k)

```
1: sequentialAdd(V);
2: return concurrentRemove(k);
```

Algorithm 7 CASR(V, k)

```
1: concurrentAdd(V);
2: return sequentialRemove(k);
```

Algorithm 8 CACR(V, k)

```
1: concurrentAdd(V);
2: return concurrentRemove(k);
```

In this work, we use the 5 concurrent priority queue classes: `SimpleLinear`, `SimpleTree`, `SequentialHeap`,

FineGrainedHeap, and SkipQueue, presented in the textbook [10] to implement the above algorithms. 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 classes all explicitly use locks for synchronization, except the SkipQueue class, which is based on a skiplist that uses an atomic primitive (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, in which source test case and follow-up test case are denoted as I_s and I_f, respectively.

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.
In this MR, $I_f = V'$ is constructed by randomly adjusting the positions of elements in V. For example, $V = [1, 2, 3, 4, 5]$, then a V' could be $[5, 4, 3, 1, 2]$. Since all elements in I_f are same as I_s, we have $\text{TOP}(V', n) = \text{TOP}(V, n)$.

MR-2 (Commutative Law) $\text{TOP}(WV^*, n) = \text{TOP}(V^*W, n)$.
Consider the V is composed of two sublist V^* and W. I_f could be obtained by exchanging the V^* and W. For example, $I_s = V^*W$, where $V^* = [1, 2, 3]$ and $W = [4, 5]$, then I_f is obtained by concatenating V^* and W, that is $I_f = WV^* = [4, 5, 1, 2, 3]$. All elements in I_f are same as I_s, hence, we have $\text{TOP}(WV^*) = \text{TOP}(V^*W)$.

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 $k > 1$.
In this MR, I_f is constructed by concatenating all elements in $I_s = V$ to the end of I_s one by one for k times. As a consequence, I_f has same elements as I_s, but is k times the length of I_s. Therefore, we have $\text{TOP}(V^k) = \text{TOP}(V, n)$

MR-4 $\text{TOP}(VV', n) = \text{TOP}(V, n)$ for any $V' \subseteq V$.
Suppose V' is any non-empty sublist of $I_s = V$. I_f is constructed by concatenating all elements in V' to the end of I_s one bu one, denoted as $I_f = VV'$. As a consequence, $\forall e_i \in I_f$, we have $e_i \in I_s$. Therefore, $\text{TOP}(VV', n) = \text{TOP}(V, n)$.

MR-5 $\text{TOP}(V[y], n)$=$\text{TOP}(V, n)$ for any y in $\text{TOP}(V, n)$.
Suppose $y \in \text{TOP}(V, n)$. I_f is constructed by concatenating y to the end of $I_s = V$. Because $y \in I_s$, therefore, $\forall e_i \in I_f$, $e_i \in I_s$. Thus, $\text{TOP}(V[y], n) = \text{TOP}(V, n)$.

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 $1 \leq i \leq k$.
Suppose $V_i \subseteq \text{TOP}(V, n)$, where $1 \leq i \leq k$, and $k \geq 1$. I_f is constructed by concatenating elements in V_i to the end of $I_s = V$ one by one. Because $V_i \subseteq \text{TOP}(V, n), \text{TOP}(V, n) \subseteq I_s$, therefore, $\forall e_i \in V_i$, we have $e_i \in I_s$, that is $\forall e_i \in I_f$, $e_i \in I_s$. Thus, we have $\text{TOP}(VV_1 \cdots V_k, n) = \text{TOP}(V, n)$.

MR-7 $\text{TOP}(V[y_1, \ldots, y_n], n) = \text{TOP}(V, n)$.
Suppose $[y_1, \ldots, y_n]$ is the tesing result of source test case. I_f is constructed by concatenating elements in $[y_1, \ldots, y_n]$ to the end of $I_s = V$ one by one. Because $[y_1, \ldots, y_n] \subseteq I_s$, therefore, $\forall e_i \in I_f$, we have $e_i \in I_s$. Thus, we have $\text{TOP}(V[y_1, \ldots, y_n], n) = \text{TOP}(V, n)$.

MR-8 $\text{TOP}(V[y_n, \ldots, y_1], n) = \text{TOP}(V, n)$.
Suppose $[y_1, \ldots, y_n]$ is the tesing result of source test case, and $[y_n, \ldots, y_1]$ is the rearrangement of $[y_1, \ldots, y_n]$ from large to small. I_f is constructed by concatenating elements in $[y_n, \ldots, y_1]$ to the end of $I_s = V$ one by one. Because $[y_n, \ldots, y_1] \subseteq I_s$, therefore, $\forall e_i \in I_f$, we have $e_i \in I_s$. Thus, we have $\text{TOP}(V[y_n, \ldots, y_1], n) = \text{TOP}(V, n)$.

MR-9 $\text{TOP}(VW, n) = \text{TOP}(V, n)$ for any permutation W of $\text{TOP}(V, n)$.
Suppose $[y_1, \ldots, y_n]$ is the tesing result of source test case, and W is any permuation of $[y_1, \ldots, y_n]$ (Note that MR-8 is a special case of MR-9). I_f is constructed by concatenating elements in W to the end of $I_s = V$ one by one. Because $W \subseteq I_s$, therefore, $\forall e_i \in I_f$, we have $e_i \in I_s$. Thus, we have $\text{TOP}(VW, n) = \text{TOP}(V, n)$.

MR-10 $\text{TOP}([y_1, \ldots, y_n]V, n) = \text{TOP}(V, n)$
Suppose $[y_1, \ldots, y_n]$ is the tesing result of source test case. I_f is constructed by concatenating elements in $[y_1, \ldots, y_n]$ to the head of $I_s = V$ one by one. Because $[y_1, \ldots, y_n] \subseteq I_s$, therefore, $\forall e_i \in I_f$, we have $e_i \in I_s$. Thus, we have $\text{TOP}(V[y_1, \ldots, y_n], n) = \text{TOP}(V, n)$.

MR-11 $\text{TOP}([y_n, \ldots, y_1]V, n) = \text{TOP}(V, n)$
Suppose $[y_1, \ldots, y_n]$ is the tesing result of source test case, and $[y_n, \ldots, y_1]$ is the rearrangement of $[y_1, \ldots, y_n]$ from large to small. I_f is constructed by concatenating elements in $[y_n, \ldots, y_1]$ to the head of $I_s = V$ one by one. Because $[y_n, \ldots, y_1] \subseteq I_s$, therefore, $\forall e_i \in I_f$, we have $e_i \in I_s$. Thus, we have $\text{TOP}(V[y_n, \ldots, y_1], n) = \text{TOP}(V, n)$.

MR-12 $\text{TOP}(WV, n) = \text{TOP}(V, n)$ for any permutation W of $\text{TOP}(V, n)$.
Suppose $[y_1, \ldots, y_n]$ is the tesing result of source

test case, and W is any permuation of $[y_1, \ldots, y_n]$ (Note that MR-11 is a special case of MR-12). I_f is constructed by concatenating elements in W to the head of $I_s = V$ one by one. Because $W \subseteq I_s$, therefore, $\forall e_i \in I_f$, we have $e_i \in I_s$. Thus, we have $\text{TOP}(VW, n) = \text{TOP}(V, n)$.

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 $y_i < y < y_{i+1}$.
In this MR, I_f is constructed by concatenating element y to the end of $I_s = V$.

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 $y'_1 < \cdots < y'_k < y_1$.
In this MR, I_f is constructed by concatenating elements in $[y'_1, \ldots, y'_k]$ to the head of $I_s = V$ one by one.

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 $y_n < y''_1 \leq \cdots \leq y''_k$.
In this MR, I_f is constructed by concatenating elements in $[y''_1, \ldots, y''_k]$ to the head of $I_s = V$ one by one.

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).

In this MR, I_f is constructed by delecting elements in $V \cap X$. To explore the relations between $\text{TOP}(V, n)$ and $\text{TOP}(V', n)$, we first consider the following situations.

Situation 1: $V \cap X = \emptyset$, then we have $V' = V$. Because $\text{TOP}(V, n) \subseteq V$ and $V \cap X = \emptyset$, therefore $X' = \emptyset$. Also, because $\text{TOP}(V, n) = \text{TOP}(V', n)$, therefore $\text{TOP}(V, n) \backslash \text{TOP}(V', n) = \emptyset$. Obviously, $\text{TOP}(V, n) \backslash \text{TOP}(V', n) \subseteq X'$.

Situation 2: $V \cap X \neq \emptyset, X \cap \text{TOP}(V, n) = \emptyset$, then $X' = \emptyset$. $\forall e'_i \in \text{TOP}(V, n)$, we have $e'_i \notin X$, $e'_i \in V'$, which gives $\text{TOP}(V, n) \subseteq V'$. Because, $V' \subseteq V$, therefore, $\text{TOP}(V', n) = \text{TOP}(V, n)$.

Situation 3: $V \cap X \neq \emptyset, X \cap \text{TOP}(V, n) \neq \emptyset$, that is $\exists e_i \in \text{TOP}(V, n)$, subject to $e_i \notin V'$, and $e_i \in X'$. Suppose $B = \text{TOP}(V, n) \backslash X'$, we have $B \subseteq V'$. Because $B \subseteq \text{TOP}(V', n)$, therefore, $B \subseteq \text{TOP}(V', n)$. Also, because $B + X' = \text{TOP}(V, n)$, therefore, $\text{TOP}(V, n) \backslash \text{TOP}(V', n) = X'$, and thus $\text{TOP}(V, n) \backslash \text{TOP}(V', n) \subseteq X'$.

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

In this MR, I_f is constructed by splitting V into V_1 and V_2. Because $V_1 \cup V_2 = V$, therefore, $\exists A \subseteq \text{TOP}(V_1, n), B \subseteq \text{TOP}(V_2, n)$, subject to $A \cup B = \text{TOP}(V, n)$. Aboviously, $A \cup B \subseteq \text{TOP}(V_1, n) \cup \text{TOP}(V_2, n)$, from which we can obtain $\text{TOP}(V, n) \subseteq \text{TOP}(v_1, n) \cup \text{TOP}(V_2, n)$.

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 five state-of-the-art concurrent priority queue classes to implement the function $\text{TOP}(V, n)$ 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?

RQ-4 How many metamorphic relations are sufficient for MT detecting the faults that remain in a concurrent program? (probably not in this paper)

B. Object Programs

In our study, we selected 5 concurrent priority queues named `SimpleLinear`, `SimpleTree`, `SequentialHeap`, `FineGrainedHeap`, and `SkipQueue`, utilized as the data repository to be shared by multiple threads, which are present in textbook [10]. All source codes of used object programs can be obtained by visiting the following address: https://booksite.elsevier.com/9780123973375/. We employed mutation analysis [11], [12] to evaluate the fault-detection effectiveness of proposed method. We first used the tool MuJava [13] and Major [14] to generate traditional mutants, and then artificial created concurrent mutants according to 24 concurrent mutation operators proposed by [15]. Each traditional mutant and concurrent mutant was created by applying a syntactic change (using one of all applicable mutation operators provided by MuJava, Major and [15], respectively) to the original program. The same mutants generated by Major and Mujava are identified and removed by created script. Equivalent mutants are identified and removed by executing all test cases and artifical checking mutants that cannot be killed. Table I summarizes the basic information of the used programs and their mutants. A detailed description of each program is given in the following.

TABLE I
SUBJECT PROGRAMS

Programs	LOC	Number of traditional mutants	Number of concurrent mutants
`SimpleLinear`	119	22	2
`SimpleTree`	109	50	4
`SequentialHeap`	155	240	2
`FineGrainedHeap`	259	222	40
`SkipQueue`	312	129	8

C. Experimental Results

1) Mutants:

2) Concurrent Scenarios:

3) Response to RQ-1:

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

When testing a software system, the oracle problem appears in some situations where either an oracle does not exist for the tester to verify the correctness of the computed results; or an oracle does exist but cannot be used. The oracle problem often occurs in software testing, which renders many testing techniques inapplicable [2]. To alleviate the oracle problem, Chen et al. [4] proposed a technique named metamorphic testing (MT) that has been receiving increasing attention in the software testing community [2], [5], [16]. The main contributions to MT in the literature focused on the following aspects: i) MT theory; ii) combination with other techniques; iii) application of MT.

1 *Theoretical development of MT:* The MRs and the source test cases are the most important components of MT. However, defining MRs can be difficult. Chen et al. [17] proposed a specification-based method and developed a tool called MR-GENerator for identifying MRs based on category-choice framework [18]. Zhang et al. [19] proposed a search-based approach to automatic inference of polynomial MRs for a software under test, where a set of parameters is used to represent polynomial MRs, and the problem of inferring MRs is turn into a problem of searching for suitable values of the parameters. Then, particle swarm optimization is used to solve the search problem. Sun et al. [20] proposes a data-mutation directed metamorphic relation acquisition methodology, in which data mutation is employed to construct input relations and the generic mapping rule associated with each mutation operator to construct output relations. Liu et al. [21] proposed to systematically construct MRs based on some already identified MRs.
Without doubt, "good" MRs can improve the fault detection efficiency of MT. Chen et al. [22] reported that good MRs are those that can make the execution of the source-test case as different as possible to its follow-up test case. This perspective has been confirmed by the later studies [23], [24]. Asrafi et al. [25] conduct a case study to analyze the relationship between the execution behavior and the fault-detection effectiveness of metamorphic relations by code coverage criteria, and the results showed a strong correlation between the code coverage achieved by a metamorphic relation and its fault-detection effectiveness.
Source test cases also have a important impact on the fault detection effectiveness of MT. Chen et al. [26] compared the effects of source test cases generated by special value testing and random testing on the effectiveness of MT, and found that MT can be used as a complementary test method to special value testing. Batra and Sengupta [24] integrated genetic algorithms into MT to select source test cases maximising the the paths traversed in the software under test. Dong et al. [23] proposed a Path–Combination–Based MT method that first generates symbolic input for each executable paths and minis relationships among these symbolic inputs and their outputs, then constructs MRs on the basis of these relationships, and generates actual test

cases corresponding to the symbolic inputs.

Different from the above investigates, we focused on performing test cases and MRs with fault revealing capabilities as quickly as possible by making use of feedback information. We first divided the input domain into disjoint partitions, and randomly selected an MR to generate follow-up test cases depended on source test case of related input partitions, then updated the test profile of input partitions according to the results of test execution. Next, a partition was selected according to updated test profile, and an MR was randomly selected from the set of MRs whose source test cases belong to selected partition.

2 *Combination with other techniques:* In order to improve the applicability and effectiveness of MT, it has been integrated into other techniques. Xie et al. [27] combined the MT with the spectrum-based fault localization (SBFL), extend the application of SBFL to the common situations where test oracles do not exist. Dong et al. [28] proposed a method for improving the efficiency of evolutionary testing (ET) by considering MR when fitness function is constructed. Liu et al. [29] introduced MT into fault tolerance and proposed a theoretical framework of a new technique called Metamorphic Fault Tolerance (MFT), which can handle system failure without the need of oracles during failure detection. In MFT, the trustworthiness of a test case depends on the number of violations or satisfactions of metamorphic relations. The more relations are satisfied and the less relations are violated, the more trustable test case is.

3 *Application of MT:* Sun et al. [30], [31] proposed a metamorphic testing framework for web services taking into account the unique features of SOA, in which MRs are derived from the description or Web Service Description Language (WSDL) [30] of the Web service, and on the basis on of MRs, follow-up test cses are generated depended on source test cases that are randomly generated according to the WSDL. Segura et al. [32] present a metamorphic testing approach for the detection of faults in RESTful Web APIs where they proposed six abstract relations called Metamorphic Relation Output Patterns (MROPs) that can then be instantiated into one or more concrete metamorphic relations. To evaluate this approach, they used both automatically seeded and real faults in six subject Web APIs.

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] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," *SIGPLAN Not.*, vol. 49, no. 6, pp. 216–226, Jun. 2014. [Online]. Available: http://doi.acm.org/10.1145/2666356.2594334

[7] Z. Q. Zhou, S. Xiang, and T. Y. Chen, "Metamorphic testing for software quality assessment: A study of search engines," *IEEE Transactions on Software Engineering*, vol. 42, no. 3, pp. 264–284, March 2016.

[8] T. Y. Chen, F.-C. Kuo, W. Ma, W. Susilo, D. Towey, J. Voas, and Z. Q. Zhou, "Metamorphic testing for cybersecurity," *Computer*, vol. 49, no. 6, pp. 48–55, Jun. 2016. [Online]. Available: https://doi.org/10.1109/MC.2016.176

[9] Z. Q. Zhou and L. Sun, "Metamorphic testing of driverless cars," *Commun. ACM*, vol. 62, no. 3, pp. 61–67, Feb. 2019. [Online]. Available: http://doi.acm.org/10.1145/3241979

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

[11] 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.

[12] C.-A. Sun, F. Xue, H. Liu, and X. Zhang, "A path-aware approach to mutant reduction in mutation testing," *Information and Software Technology*, vol. 81, pp. 65–81, 2017.

[13] 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.

[14] R. Just, "The Major mutation framework: Efficient and scalable mutation analysis for Java," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'14)*, San Jose, CA, USA, July 23–25 2014, pp. 433–436.

[15] J. S. Bradbury, J. R. Cordy, and J. Dingel, "Mutation operators for concurrent java (j2se 5.0)," in *The 2nd International Workshop on Mutation analysis (Mutation'06), Co-located with 17th International Symposium on Software Reliability Engineering (ISSRE'06)*. IEEE, 2006, pp. 11–11.

[16] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés, "A survey on metamorphic testing," *IEEE Transactions on Software Engineering*, vol. 42, no. 9, pp. 805–824, 2016.

[17] T. Y. Chen, P.-L. Poon, and X. Xie, "Metric: Metamorphic relation identification based on the category-choice framework," *Journal of Systems and Software*, vol. 116, pp. 177–190, 2016.

[18] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating fuctional tests," *Communications of the ACM*, vol. 31, no. 6, pp. 676–686, 1988.

[19] J. Zhang, J. Chen, D. Hao, Y. Xiong, B. Xie, L. Zhang, and H. Mei, "Search-based inference of polynomial metamorphic relations," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE'14)*, 2014, pp. 701–712.

[20] C.-A. Sun, Y. Liu, Z. Wang, and W. K. Chan, "μmt: A data mutation directed metamorphic relation acquisition methodology," in *Proceedings of the 1st International Workshop on Metamorphic Testing (MET'16), Co-located with the 38th International Conference on Software Engineering (ICSE'16)*, 2016, pp. 12–18.

[21] H. Liu, X. Liu, and T. Y. Chen, "A new method for constructing metamorphic relations," in *Proceedings of the 12th International Conference on Quality Software (QSIC'12)*, 2012, pp. 59–68.

[22] T. Y. Chen, D. Huang, T. Tse, and Z. Q. Zhou, "Case studies on the selection of useful relations in metamorphic testing," in *Proceedings of the 4th lberoAmerican Symposium on Software Engineering and Knowledge Engineer-ing (JIISIC'04)*, 2004, pp. 569–583.

[23] G. Dong, T. Guo, and P. Zhang, "Security assurance with program path analysis and metamorphic testing," in *Proceedings of the 4th IEEE International Conference on Software Engineering and Service Science (ICSESS'13)*, 2013, pp. 193–197.

[24] G. Batra and J. Sengupta, "An efficient metamorphic testing technique

using genetic algorithm," in *International Conference on Information Intelligence, Systems, Technology and Management*, 2011, pp. 180–188.

[25] M. Asrafi, H. Liu, and F.-C. Kuo, "On testing effectiveness of metamorphic relations: A case study," in *Proceedings of the 15th International Conference on Secure Software Integration and Reliability Improvement (SSIRI'11)*, 2011, pp. 147–156.

[26] T. Y. Chen, F.-C. Kuo, Y. Liu, and A. Tang, "Metamorphic testing and testing with special values." in *Proceedings of the 5th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD'04)*, 2004, pp. 128–134.

[27] X. Xie, W. E. Wong, T. Y. Chen, and B. Xu, "Metamorphic slice: An application in spectrum-based fault localization," *Information and Software Technology*, vol. 55, no. 5, pp. 866–879, 2013.

[28] G. Dong, S. Wu, G. Wang, T. Guo, and Y. Huang, "Security assurance with metamorphic testing and genetic algorithm," in *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT'10)*, vol. 3, 2010, pp. 397–401.

[29] H. Liu, I. I. Yusuf, H. W. Schmidt, and T. Y. Chen, "Metamorphic fault tolerance: An automated and systematic methodology for fault tolerance in the absence of test oracle," in *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*, 2014, pp. 420–423.

[30] C.-A. Sun, G. Wang, B. Mu, H. Liu, Z. Wang, and T. Y. Chen, "Metamorphic testing for web services: Framework and a case study," in *Proceedings of the 9th IEEE International Conference on Web Services (ICWS'11)*, 2011, pp. 283–290.

[31] ——, "A metamorphic relation-based approach to testing web services without oracles," *International Journal of Web Services Research*, vol. 9, no. 1, pp. 51–73, 2012.

[32] S. Segura, J. A. Parejo, J. Troya, and A. Ruiz-Cortés, "Metamorphic testing of restful web apis," *IEEE Transactions on Software Engineering*, vol. 44, no. 11, pp. 1083–1099, 2018.