AMT: An Efficient and Novel Method to Improve the Fault-detecting Efficiency of Metamorphic Testing

Chang-ai Sun, *Senior Member, IEEE,* Hepeng Dai, Huai Liu, *Member, IEEE,* and Tsong Yueh Chen, *Member, IEEE,*

Abstract—Metamorphic testing (MT) is a promising technique to alleviate the oracle problem, which first defines metamorphic relations (MRs) that are then used to generate new test cases (i.e. follow-up test cases) from the original test cases (i.e. source test cases), and verify the results of source and follow-up test cases. Many efforts have been reported to improve MT's efficiency by either generating better MRs that are more likely to be violated or selecting different test case selection strategies to generate source test cases. Unlike these efforts, we investigate how to improve the efficiency of MT in terms of test executions. Furthermore, traditional MT techniques often employ the random testing strategy (RT) to select source test cases for execution, which could be inefficient because the feedback information during the testing process is not leveraged. Consequently, we propose an adaptive metamorphic testing (AMT) technique to improve the efficiency of MT through controlling the execution process of MT. We conducted an empirical study to evaluate the efficiency of the proposed technique with three real-life programs. Empirical results show that AMT outperforms traditional MT in terms of fault-detecting efficiency.

Index Terms—metamorphic testing, control test process, feedback, random testing, adaptive random testing, partition testing, adaptive partition testing

1 Introduction

Test result verification is an important part of software testing. A test oracle [1] is a mechanism that can exactly decide whether the output produced by a programs is correct. However, there are situations where it is difficult to decide whether the result of the software under test (SUT) agrees with the expected result. This situation is known as oracle problem [2], [3]. In order to alleviate the oracle problem, several techniques have been proposed, such as N-version testing [4], metamorphic testing (MT) [5], [6], assertions [7], and machine learning [8]. Among of them, MT obtains metamorphic relations (MRs) according to the properties of SUT. Then, MRs are used to generate new test cases called follow-up test cases from original test cases known as source test cases. Next, both source and follow-up test cases are executed and their result are verified against the corresponding MRs.

MT has been receiving increasing attention in the software testing community, since this technique not only alleviates the oracle problem but also generates new test cases from existing test cases. MT has been successfully applied in a number of application domains and paradigms, including healthcare [9], bioinformatics [10], air traffic control [11], the web service [12], the RESTful web APIs [13], and testing of deep learning system [14]. The applications of MT emphasizes that MT is a new and promising strategy that complements the existing testing approaches.

The successes in applying MT to multiple application domains and paradigms continues to stimulate researchers to develop the theoretical foundations for MT. There are two broad categories of methods to improve the effectiveness of MT: source test cases generating and MRs identifying. The former employs different test case selection strategies to generate source test cases for MT [15], [16]. The latter creates MRs by combining existing relations, or by generating them automatically [17]–[21].

Random testing (RT) that randomly selects test cases from input domain (which refers to the set of all possible inputs of SUT), which is most commonly used strategies in traditional MT [22]. Although RT is simple to implement, RT does not make use of any execution information about SUT or the test history. Thus, traditional MT may be ineffectiveness in some situations. Barus et al. [16] employed adaptive random testing [23] that is a class of testing method aimed to improve the performance of RT by increasing the diversity across a programs input domain of the test cases executed to generate source test cases for MT. The Fixed-Size Candidate Set ART technique (FSCS-ART) [23] was the first ART method, and is also the most widely studied, which selects a next test input from the fixed-size candidate set of tests that is farthest from all previously executed tests. Many other ART algorithms have also been proposed, including RRT [24], [25], DF-FSCS [26], and ARTsum [27], with their effectiveness examined and validated through simulations and experiments.

In contrast to RT, partition testing (PT) attempts to

C.-A. Sun, H. Dai, and G. Wang are with the School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing 100083, China. E-mail: casun@ustb.edu.cn.
H. Liu is with the College of Engineering and Science, Victoria University, Melbourne VIC 8001, Australia. E-mail: Huai.Liu@vu.edu.au.
T.Y. Chen is with the Department of Computer Science and Software Engineering, Swinburne University of Technology, Hawthorn VIC 3122, Australia. Email: tychen@swin.edu.au.

generate test cases in a more systematic way, aiming to use fewer test cases to detect more faults. When conducting PT, the input domain of SUT is divided into disjoint partitions, with test cases then selected from each and every one. Each partition is expected to have a certain degree of homogeneity, that is, test cases in the same partition should have similar software execution behavior: If one input can detect a fault, then all other inputs in the same partition should also be able to detect a fault. RT and PT have their own characteristics. Therefore, it is a nature thought to investigate the integration of them for developing new testing techniques. Cai et al. [28] proposed dynamic random testing (DRT) that takes advantages of testing information to control the testing process, with the goal of benefitting from the advantages of RT and PT. Sun et al. proposed adaptive partition testing [29], where test cases are randomly selected from some partition whose probability of being selected is adaptively adjusted along the testing process, and developed two algorithms, Markov-chain based adaptive partition testing and reward-punishment based adaptive partition testing, to implement the proposed approach.

The set of identified MRs is the other foremost component of MT, which impacts the fault-detection effectiveness of MT, and that a common opinion has been identification of these MRs appears to be a manual and tough task [30]: "Even experts may find it difficult to discover metamorphic relations". To address this question, a number of methods have been proposed such as MR composition [31], [32], Machine-learning based MR detection [33], Graph-kernel based MR prediction [34], Search-based MR inference [35], data-mutation-directed MR acquisition [36]. More recently, a more systematical and effective method (METRIC) [20] was proposed, which make use of complete test frame constructed by categories and choices [37]. Those methods help test engineers create a set of MRs, while cannot guide select the next MR during the test.

In previous and existing work, researchers investigated the effectiveness of MT by either choosing different MRs alone, or choosing different test case selection strategies to generate source test cases alone. Different from the previous methods, this paper proposed an innovative method (AMT) that makes use of feedback information to select both source test case and corresponding MR, aiming to maximize the fault-detection effectiveness of MT.

AMT is built on top of three insights: i) source test cases is one of factors that effects the fault-detecting efficiency of MT. However, random testing is commonly used to generate source test cases, and only a little researches related to use appropriate test cases selection strategies to generate source test cases; ii) most of existing work help testers create MRs, aiming to improve the fault-detection efficiency of MT, while there seems to be no investigation to study how to choose an MR based on the generated set of MRs; iii) an appropriate test cases selection strategy can generate "better" test cases (i.e. those test cases have a higher probability to detect a fault). In MT, if source test cases cannot detect a fault, and follow-up test case generated by an MR detect a fault, we still can conclude that this execution detect a fault. Therefore, based on a selected source test case, choosing a "better" MR can improve the fault-detection efficiency.

In short, AMT collects feedback information during the test, and makes use of those information to select source test case and MRs.

In this study, we investigate how to make use of feedback information in the previous tests to control the execution process of MT in terms of both selecting source test cases and MRs. As a result, a process-feedback metamorphic testing framework is proposed to improve the fault-detecting efficiency of MT. An empirical study was conducted to evaluate the efficiency of the proposed technique. Main contributions made in this paper are the following:

1 From a new perspective, we proposed a adaptive metamorphic testing method. This includes a universal framework (AMT) that indicates how to make use of history testing information to select next source test case and MR.
2 We particular developed two kinds of algorithms, partition based AMT (P-AMT) and random based AMT (R-AMT), to implement the proposed framework. P-AMT includes two methods: i) An MRs-centric P-AMT (MP-AMT), which first randomly selects an MR to generate source and follow-up test cases of related input partitions, and then updates the test profile of input partitions according to the result of test execution. Second, a partition is selected according to updated test profile, and an MR is selected based on proposed strategies from the set of MRs whose source test cases belong to selected partition; ii) A partition-centric P-AMT (PP-AMT), which leverages MRs as a mechanism for verifying the test results. First, PP-AMT selects a partition according to the test profile. Second, a test case is randomly selected in the selected partition, its follow-up test cases are generated based on the involved MRs, and their results are verified against the involved MRs. Third, PP-AMT updates the test profile based on the test results. R-AMT include one method: we employed ARTsum to select source test cases and then an MR is selected based on proposed strategies from the set of MRs whose source test cases is selected source test case.
3 In order to support proposed methods, we proposed two algorithms to update the test profile, and two algorithms to select an MR form the candidate MRs.
4 We evaluated the performance of AMT through a series of empirical studies on 12 programs. These studies show that AMT has significantly higher fault-detection efficiency than traditional MT that randomly selects source test cases and a revised MT method that employed ART method to select source test cases, ignoring to select a "good" MR.

The rest of this paper is organized as follows. Section 2 introduces the underlying concepts for DRT, web services and mutation analysis. Section **??** presents the DRT framework for web services, guidelines for its parameter settings, and a prototype that partially automates DRT. Section **??** describes an empirical study where the proposed DRT is used to test three real-life web services, the results of which are summarized in Section 5. Section 6 discusses related work and Section **??** concludes the paper.

2 BACKGROUND

In this section, we present some of the underlying concepts for MT, DRT, and ART.

2.1 Metamorphic Testing (MT)

MT is a novel technique to alleviate the oracle problem: Instead of applying an oracle, MT uses a set of MRs (corresponding to some specific properties of the SUT) to verify the test results. MT is normally conducted according to the following steps:

Step1. Identify an MR from the specification of the SUT.
Step2. Generate the source test case stc using the traditional test cases generation techniques.
Step3. Derive the follow-up test case ftc from the stc based on the MR.
Step3. execute stc and ftc and get their outputs O_s and O_f.
Step4. Verify stc, ftc, O_s, and O_f against the MR: If the MR does not hold, a fault is said to be detected.

The above steps can be repeated for a set of MRs.

Let us use a simple example to illustrate how MT works. For instance, consider the mathematic function $f(x, y)$ that can calculate the maximal value of two integers x and y. There is a simple yet obvious property: the order of two parameters x and y does not affect the output, which can be described as the follow metamorphic relation (MR): $f(x, y) = f(y, x)$. In this MR, (x, y) is source test case, and (y, x) is considered as follow-up test case. Suppose P denotes a program that implements the function $f(x, y)$, P is executed with a test cases $(1, 2)$ and $(2, 1)$. Then we check $P(1, 2) = P(2, 1)$: If the equality does not hold, then we consider that P at least has one fault.

2.2 Category Partition Method (CPM)

The proposed AMT selects both source test cases and corresponding MRs by making use of information which is collected during the test process that means, we need to establish a connection between the source test cases and MRs. The connection provides a guideline for us to obtain the set of candidate MRs according to a source test case or select a source test case for an MR.

Category partition method (CPM) is specification-based testing technique developed by Ostrand and Balcer [38]. It helps software testers create test cases by refining the functional specification of a program into test specifications. The method consists of the following steps.

Step1. Decompose the functional specification into functional units that can be tested independently, then Identify the parameters (the explicit inputs to a functional unit) and environment conditions (the state of the system at the time of execution) that are known as *categories*.
Step2. Partition each *category* into *choices*, which include all the different kinds of values that are possible for that *category*.
Step3. Determine the constraints among the *choices* of different *categories*, and Write the test specification (which is a list of categories, choices, and constraints in a predefined format) using the test specification language TSL.
Step4. Use a generator to produce *test frames* from the test specification. Each generated *test frame* is a set of choices. Then, create a test case by selecting a single element from each *choice* in each generated *test frame*.

Based on the *categories* and *choices*, Chen et al. [20] proposed a systematical method (METRIC) to generate a set of MRs. In METRIC, only two distinct complete *test frames* that are abstract test cases defining possible combinations of inputs, are considered by the tester to generate an MR. In general, METRIC has the following steps to identify MRs:

Step1. Users select two relevant and distinct complete *test frames* as a *candidate pair*.
Step2. Users determine whether or not the selected *candidate pair* is useful for identifying an MR, and if it is, then provide the corresponding MR description.
Step3. Restart from step 1, and repeat until all candidate pairs are exhausted, or the predefined number of MRs to be generated is reached.

In this study, we made use of CPM and METRIC to create test cases and MRs, respectively. There are relationships between source test cases and MRs, since both source test cases and MRs are built upon categories and choices.

2.3 Dynamic Random Testing (DRT)

Cai et al [28] proposed dynamic random testing (DRT), which adjusts the test profile according to the result of current test. In DRT, the input domain is first divided into m partitions (denoted $s_1, s_2, \ldots, s_m$), and each s_i is assigned a probability p_i. During the test process, the selection probabilities of partitions are dynamically updated. Suppose that a test case tc from s_i $(i = 1, 2, \ldots, m)$ is selected and executed. The process of DRT adjusting the value of p_i is as follows: If tc detects a fault, $\forall j = 1, 2, \ldots, m$ and $j \neq i$, we then set

$$p'_j = \begin{cases} p_j - \dfrac{\epsilon}{m-1} & \text{if } p_j \geq \dfrac{\epsilon}{m-1} \\ 0 & \text{if } p_j < \dfrac{\epsilon}{m-1} \end{cases}, \tag{1}$$

where ϵ is a probability adjusting factor, and then

$$p'_i = 1 - \sum_{\substack{j=1 \\ j \neq i}}^{m} p'_j. \tag{2}$$

Alternatively, if tc does not detect a fault, we set

$$p'_i = \begin{cases} p_i - \epsilon & \text{if } p_i \geq \epsilon \\ 0 & \text{if } p_i < \epsilon \end{cases}, \tag{3}$$

and then for $\forall j = 1, 2, \ldots, m$ and $j \neq i$, we set

$$p'_j = \begin{cases} p_j + \dfrac{\epsilon}{m-1} & \text{if } p_i \geq \epsilon \\ p_j + \dfrac{p'_i}{m-1} & \text{if } p_i < \epsilon \end{cases}. \tag{4}$$

The detailed DRT algorithm is given in Algorithm 1. In DRT, a test case is selected from a partition that has been randomly selected according to the test profile $\{\langle s_1, p_1 \rangle, \langle s_2, p_2 \rangle, \ldots, \langle s_m, p_m \rangle\}$ (Lines 2 to 4). If a fault is detected, then Formulas 1 and 2 are used to adjust the values

of p_i (Line 6), otherwise Formulas 3 and 4 are used (Line 8). This process is repeated until a termination condition is satisfied (Line 1). Termination conditions could be set as "when the testing resources are consumed exhausted", "when a certain number of test cases have been executed", or "when the first fault is detected".

In AMT, when a source test case and corresponding follow-up test case belong to same partition s_i, DRT is employed to update the test profile as follows: If a fault is detected (i.e., their results violate the related MR), then Formulas 1 and 2 are used to adjust the values of p_i, otherwise Formulas 3 and 4 are used. During the testing process, there is the other situation where a source test case and corresponding follow-up test case do not belong to same partition. In order to adjust the test profile in this situation, we proposed a new strategy that is described in Section 3.3.

Algorithm 1 DRT

Input: $\epsilon, p_1, p_2, \ldots, p_m$

1: **while** termination condition is not satisfied **do**
2: Select a partition s_i according to the test profile $\{\langle s_1, p_1\rangle, \langle s_2, p_2\rangle, \ldots, \langle s_m, p_m\rangle\}$.
3: Select a test case tc from s_i.
4: Test SUT using tc.
5: **if** a fault is detected by tc **then**
6: Update test profile according to Formulas 1 and 2.
7: **else**
8: Update test profile according to Formulas 3 and 4.
9: **end_if**
10: **end_while**

2.4 Adaptive Random Testing (ART)

To improve the fault-detection efficiency of RT, Chen et al. proposed adaptive random testing [23] based on the intuition that two test cases close to each other were more likely to have the same failure behavior than two test cases that were widely separated [39], that means faults tend to cause erroneous behaviour to occur across contiguous regions of the input domain. Hence, if a method that spread test cases more evenly would identify failures using fewer test cases.

There are many possible ways to apply the principle of ART. One that has been widely used is distance-based ART, also known as Fixed-Size Candidate Set ART (FSCS-ART) [23]. It makes use of the Euclidean distance as a distance measure to ensure test cases are spread out evenly. At first, two types of sets are defined: candidate set and executed set. The candidate set contains a predefined number n candidate inputs, whereas the later set stores all test cases that have been executed. Both sets are initially empty. Then n elements of candidate set are selected randomly from the existing input domain. For the first round, a random test case is selected, executed, and if the execution does not reveal any failures, it would be added to the executed set whilst the candidate set is reset as empty. For subsequent iterations, n test cases are randomly selected and placed again in the candidate set. Then, the minimum distance of each candidate test case to the executed test cases is calculated, that is the nearest executed test case. The candidate having maximum minimum distance is then selected as the next test case. This is known as the max-min criterion. The testing stops whenever an executed test case reveals a failure.

ART has been empirically shown to be effective on software with numeric inputs [40]–[44] and Non-numeric inputs [16], [27], [45]. However, as Ciupa et al. [46] observe, test case selection overhead can result in FSCS-ART having poorer overall cost-effectiveness than RT. To reduce the overhead of ART, various algorithms have been proposed [47]–[49]. More recently, Barus et al. [27] proposed a linear-order ART algorithm called ARTsum.

Before presenting the details of ARTsum, we first introduce a new metric called category-partition-based metric (CP-distance) that measures the distance between the candidate test cases and executed test cases. CP-distance makes use of the concepts of categories and choices from the CPM method described in Section 2.2, which has the capacity to reflect the difference among the candidate test cases and executed test cases, since the categories and choices are defined based on the software functionalities.

More formally, let us denote the set of categories by $A = \{A_1, A_2, \ldots, A_g\}$, where g denotes the total number of categories. For each A_i, its choices are denotes by $P_i^{A_i} = \{p_1^{A_i}, p_2^{A_i}, \ldots, p_h^{A_i}\}$, where h denotes the number of choices for A_i. For input x, let us denote the corresponding non-empty subset by $A(x) = \{A_1^x, A_2^x, \ldots, A_q^x\}$, where q refers to the number of categories associated with x. Since categories are distinct and their choices are disjoint, input x in fact consists of values chosen from a non-empty subset of choices, denoted as $P(x) = \{p_1^x, p_2^x, \ldots, p_q^x\}$, where $p_i^x (i = 1, 2, \ldots, q)$ is the choice of the category A_i^x for x.

For any two inputs x and y, we define $DP(x, y)$ as the set that contains elements in either $P(x)$ or $P(y)$ but not both. That is,

$$DP(x, y) = (P(x) \cup P(y))(P(x) \cap P(y)),$$

where "n" is the set difference operator. Now, we define

$$DA(x, y) = \{A_m | A_i if p_j^i \in DP(x, y)\}.$$

In other words, $DA(x, y)$ is the set of categories in which inputs x and y have different choices. Then, the distance measure between x and y is defined as $|DA(x, y)|$ (the size of $DA(x, y)$); that is, the number of categories that appear in either x or y but not both, or in which the choices in x and y differ.

Suppose that n test cases have been selected and executed, denoted by $E = \{e_1, e_2, \ldots, e_n\}$. Each test case e_j ($j = \{1, 2, \ldots, n\}$) is associated with a set of choices $P(e_j) = \{p_1^{e_j}, p_2^{e_j}, \ldots, p_q^{e_j}\}$. $P(e_j)$ can be rewritten as a tuple $V(e_j) = \{v_1^{e_j}, v_2^{e_j}, \ldots, v_g^{e_j}\}$, where g is the total number of categories, $v_i^{e_j}$ means that e_j is not associated with category A_i, and $v_i^{e_j} = l$ means that e_j is associated with the lth choice of A_i. Similarly, a candidate c can also be associated with the tuple $V(c) = \{v_1^c, v_2^c, \ldots, v_g^c\}$.

Define a function as follows:

$$D(i, j) = \begin{cases} 0 & \text{if } v_i^c = v_i^{e_j} \\ 1 & \text{if } v_i^c \neq v_i^{e_j} \end{cases} \quad (5)$$

where $i = 1, 2, \ldots, g$ and $j = 1, 2, \ldots, n$. The distance between c and e_j can then be calculated as $dist(c, e_j) = \sum_{i=1}^{g} D(i, j)$. Note that $dist(c, e_j)$ is effectively equal to $|DA(c, e_j)|$. Therefore, the sum of the distances from c to all executed test cases can be calculated as:

$$sum\_dist(c, E) = \sum_{j=1}^{n}(\sum_{i=1}^{g} D(i, j)) \quad (6)$$

Then we present the Theorem 2.4 that is the foundation stone of ARTsum.

Theorem 1. Define a tuple of integers $S = (s_1^0, s_1^1, \ldots, s_1^{h_1}, s_2^0, s_2^1, \ldots, s_2^{h_2}, \ldots, s_g^0, s_g^1, \ldots, s_g^{h_g})$, where g is the total number of categories, h_i is the total number of choices for the ith category A_i $(i = 1, 2, \ldots, g)$, s_i^0 denotes the number of previously executed test cases that are not associated with the ith category A_i, $s_i^{l_i}$ $(l_i = 1, 2, \ldots, h_i)$ denotes the number of previously executed test cases associated with the l_ith choice of A_i. Let n denote the number of previously executed test cases, that is, $n = |E|$. By definition, $\sum_{y=0}^{h_i} s_i^y = n, \forall 1 \leq i \leq g$. For a candidate c associated with $V(c) = \{v_1^c, v_2^c, \ldots, v_g^c\}$, the sum distance between c and E is:

$$sum\_dist(c, E) = \sum_{j=1}^{n}(n - s_i^{v_i^c}) \quad (7)$$

Theorem 2.4 implies that if Equation 7 is used, the selection of a next test case requires a constant time. The details of the ARTsum is given in the Algorithm 2.

Algorithm 2 ARTsum

1: Initialize S (as defined in Theorem 2.4) by setting each $s_i^{l_i} = 0$, where $i = 1, 2, \ldots, g$ (g denotes the total number of categories).
2: Set $n = 0$ and $E = \{\}$.
3: Define an integer $k > 0$ as the number of candidates to be generated.
4: **while** termination condition is not satisfied **do**
5: Increment n by 1.
6: **if** $n = 1$ **then**
7: Randomly generated a test case e_n.
8: **else**
9: Randomly generate k candidates $c_1, c_2, \ldots, c_k$.
10: **for** all c_u $(u = 1, 2, \ldots, k)$ **do**
11: Calculate $sum\_dist(c, E)$ according to Equation 7.
12: **end for**
13: Set $e_n = c_0$, where $\forall u$, $sum\_dist(c_0, E) \geq sum\_dist(c_u, E)$.
14: **end_if**
15: Add e_n into E.
16: Update S by incrementing each $s_i^{v_i^{e_n}}$ by 1, where $i = 1, 2, \ldots, g$.
17: **end_while**

3 Adaptive Metamorphic Testing (AMT)

In this section, we describe a framework for applying feedback mechanism to select source test cases and MRs for MT.

3.1 Motivation

Since MT was first published, a considerable number of studies have been reported from various aspects [22]. To improve the efficiency of MT, most of studies have paid their attention to identify the better MRs, which are more likely to be violated. For the efficiency of MRs, several factors such as the difference between the source and follow-up test cases [17], [18] and the the detecting-faults capacity of MRs compared to existing test oracles [50], have been investigated.

Since the follow-up test cases are generated based on source test cases and MRs, in addition to the so-called good MRs, source test cases also have an impact on the efficiency of MT. However, 57% of existing studies employed RT to select test cases, and 34% of existing studies used existing test suites according to a survey report by Segura et al. [22]. In this study, we investigate the strategies of selection test cases and MRs, and its impacts on the fault-detecting efficiency of MT.

It has been pointed out that fault-detecting inputs tend to cluster into "continuous regions" [51], [52], that is, the test cases in some partitions are more likely to detect faults than the test cases in other partitions. Inspired by the observation, AMT takes full advantage of feedback information to update the test profile, aiming at increasing the selection probabilities of partitions with larger failure rates. Accordingly, the MRs whose sources test cases belonging to the partitions with larger failure rates, are more likely to be selected and violated. Therefore, AMT is expected to detect faults more efficient than traditional MT.

3.2 Framework

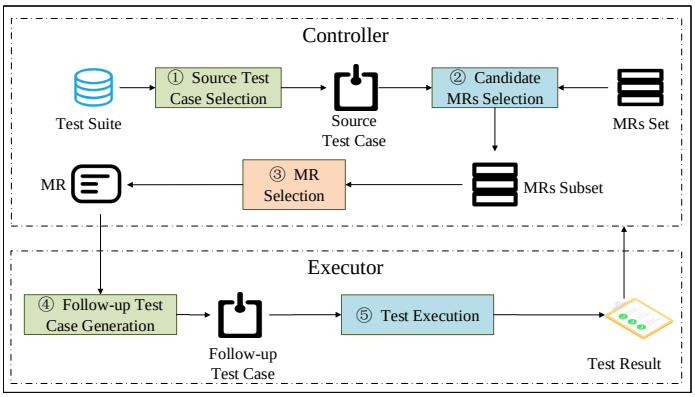

Fig. 1. The Framework of AMT

1 *Source Test Case Selection*.
2 *candidate MRs Selection*.
3 *MR Selection*.
4 *Follow-up Test Case Generation*.
5 *Test Case Execution*. The relevant DRT component receives the generated test case, converts it into an input message, invokes the web service(s) through the SOAP protocol, and intercepts the test results (from the output message).

3.3 Updating Test Profile

When the source test case stc and follow-up test case ftc related to an metamorphic relation MR belong to the

same partition s_i, DRT is suitable to adjust the test profile. However, during the test process, there are some scenarios where the stc is not in the same partition as ftc. In such scenarios, DRT cannot be used to adjust the test profile. To solve this problem, we proposed metamorphic dynamic random testing (MDRT).

Suppose that source test case stc and follow-up test case ftc related to an metamorphic relation MR, belong to s_i and s_f ($f \in \{1, 2, \ldots, m\}, f \neq i$), respectively. If their results violate the MR, $\forall j = 1, 2, \ldots, m$ and $j \neq i, f$, we set

$$p_j' = \begin{cases} p_j - \dfrac{2\epsilon}{m-2} & \text{if } p_j \geq \dfrac{2\epsilon}{m-2} \\ 0 & \text{if } p_j < \dfrac{2\epsilon}{m-2} \end{cases}, \tag{8}$$

where ϵ is a probability adjusting factor, and then

$$p_i' = p_i + \frac{(1 - \sum_{j=1, j\neq i,m}^{m} p_j^{'}) - p_i - p_f}{2}, \tag{9}$$

$$p_f' = p_f + \frac{(1 - \sum_{j=1, j\neq i,m}^{m} p_j^{'}) - p_i - p_f}{2}. \tag{10}$$

Alternatively, if their results hold the MR_h, we set

$$p_i' = \begin{cases} p_i - \epsilon & \text{if } p_i \geq \epsilon \\ 0 & \text{if } p_i < \epsilon \end{cases}, \tag{11}$$

$$p_f' = \begin{cases} p_f - \epsilon & \text{if } p_f \geq \epsilon \\ 0 & \text{if } p_f < \epsilon \end{cases}, \tag{12}$$

and then for $\forall j = 1, 2, \ldots, m$ and $j \neq i, f$, we set

$$p_j' = p_j + \frac{(p_i - p_i^{'}) + (p_f - p_f^{'})}{m-2} \tag{13}$$

The detailed MDRT algorithm is given in Algorithm 3. In MDRT, the source test case is selected from a partition that has been randomly selected according to the test profile $\{\langle s_1, p_1\rangle, \langle s_2, p_2\rangle, \ldots, \langle s_m, p_m\rangle\}$, and an metamorphic relation is select according to some strategies (Lines 2 to 4 in Algorithm 1). During the testing, if the source and follow-up test cases are not in same partition, then the test profile is updated by changing the p_i: If a fault is detected, then Formulas 8, 9, and 10 are used to adjust the values of p_i (Line 8), otherwise Formulas 11, 12, and 13 are used (Line 10). The testing process is stopped as long as the termination condition is satisfied.

Algorithm 3 MDRT

Input: $\epsilon, p_1, p_2, \ldots, p_m, MR_1, MR_2, \ldots, MR_n$

1: **while** termination condition is not satisfied **do**
2: Select a partition s_i according to the test profile $\{\langle s_1, p_1\rangle, \langle s_2, p_2\rangle, \ldots, \langle s_m, p_m\rangle\}$.
3: Select a source test case stc_i from s_i, and an metamorphic relation MR_h ($h \in \{1, 2, \ldots, n\}$).
4: Based on the MR_h, follow-up test case ftc_i is generated from stc_i, belonging to partition s_f.
5: Test the SUT using stc_i and ftc_i.
6: **if** $i \neq f$ **then**
7: **if** the results of stc_i and ftc_i violate the MR **then**
8: Update $\{\langle s_1, p_1\rangle, \langle s_2, p_2\rangle, \ldots, \langle s_m, p_m\rangle\}$ according to Formulas 8, 9, and 10.
9: **else**
10: Update $\{\langle s_1, p_1\rangle, \langle s_2, p_2\rangle, \ldots, \langle s_m, p_m\rangle\}$ according to Formulas 11, 12, and 13.
11: **end_if**
12: **end_if**
13: **end_while**

3.4 Selecting MR Strategies

In AMT, once a source test case is selected, the corresponding candidate MRs are available. We first proposed a simple strategy based on random mechanism to select an MR from the set of candidate MRs, then we proposed an innovative strategy to select an MR that can make the execution of follow-up test case as different as possible from the source test case.

3.4.1 Randomly MR-Selection Strategy (RMRS)

RSMR randomly selects an from the set of candidate MRs, which is a straightforward method without considering extra information.

3.4.2 Properties-Based strategy of MR selection (PBMR)

The fault-detection efficiency of MT highly dependent on the specific MRs that are used, and selecting effective MRs is thus a critical step when applying MT. Chen et al. reported that good metamorphic relations are those that can make the execution of the source-test case as different as possible to its follow-up test case. Moreover, They defined the "difference among execution" as any aspects of program runs (e.g., paths traversed). Before presenting the properties-based strategy of MR selection (PBMR), we first introduce a new metric called category-partition-based metric (CP-distance) that measures the distance between the source test cases and corresponding follow-up test cases. CP-distance makes use of the concepts of categories and choices from the CPM method described in Section 2.2. CP-distance have the capacity to reflect the difference among the source test cases and follow-up test cases, since the categories and choices are defined based on the software functionalities.

More formally, let us denote the set of categories by $A = \{A_1, A_2, \ldots, A_g\}$, where g denotes the total number of categories. For each A_i, its choices are denotes by $P_i^{A_i} = \{p_1^{A_i}, p_2^{A_i}, \ldots, p_h^{A_i}\}$, where h denotes the number of choices for A_i. For input x, let us denote the corresponding non-empty subset by $A(x) = \{A_1^x, A_2^x, \ldots, A_q^x\}$, where q refers to the number of categories associated with x. Since categories are distinct and their choices are disjoint, input x in fact consists of values chosen from a non-empty subset of choices, denoted as $P(x) = \{p_1^x, p_2^x, \ldots, p_q^x\}$, where $p_i^x (i = 1, 2, \ldots, q)$ is the choice of the category A_i^x for x.

For any two inputs x and y, we define $DP(x, y)$ as the set that contains elements in either $P(x)$ or $P(y)$ but not both. That is,

$$DP(x, y) = (P(x) \cup P(y))(P(x) \cap P(y)),$$

where "n" is the set difference operator. Now, we define

$$DA(x, y) = \{A_m | A_i if p_j^i \in DP(x, y)\}.$$

In other words, $DA(x, y)$ is the set of categories in which inputs x and y have different choices. Then, the distance

measure between x and y is defined as $|DA(x,y)|$ (the size of $DA(x,y)$); that is, the number of categories that appear in either x or y but not both, or in which the choices in x and y differ.

obviously, a greater value of CP-distance representing more dissimilar execution. After selecting a source test case stc_i and obtaining a set of candidate MRs $\mathcal{R} = \{r_1^{stc_i}, r_2^{stc_i}, \ldots, r_g^{stc_i}\}$ (g is the number of MRs whose source test case could be stc_i), PBMR generate a set of candidate folow-up test cases $FC = \{ftc_{r_1^{stc_i}}, ftc_{r_2^{stc_i}}, \ldots, ftc_{r_g^{stc_i}}\}$ according to every MR belonged to FC. Then, the distance $CP_{i,h}$ ($h \in \{1,2,\ldots,g\}$) between stc_i and each follow-up test case $ftc_{r_h^{stc_i}}$ is calculated. Finally, the MR $r_h^{stc_i}$ is selected as long as the following condition is hold:

$$CP_{i,h} = max\{CP_{i,1}, CP_{i,2}, \ldots, CP_{i,g}\}.$$

The details of PBMR is given in Algorithm 4.

Algorithm 4 PBMR

Input: $stc_i, \mathcal{R} = \{r_1^{stc_i}, r_2^{stc_i}, \ldots, r_g^{stc_i}\}, CParray = null$
Output: CP_{max} ($max \in \{1,2,\ldots,g\}$)

1: **for** $h = 1 \rightarrow h = g$ **do**
2: Generate the follow-up test case ftc_h according to the stc_i and $r_h^{stc_i}$
3: Calculate the distance $CP_{i,h}$ between the stc_i and ftc_h
4: Add $CP_{i,h}$ to the list $CParray$
5: **end for**
6: Calculate the maximal value CP_{max} ($max \in \{1,2,\ldots,g\}$) in the list $CParray$

3.5 Partition Based AMT (P-AMT)

3.5.1 A partition-centric P-AMT

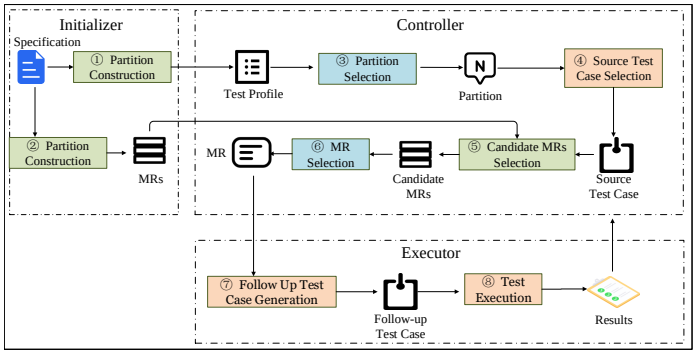

Fig. 2. framework

3.5.2 A MRs-centric P-AMT (MP-AMT)

MRs-centric adaptive metamorphic testing (MP-AMT) adds a feedback mechanism to control the execution process of traditional MT. First, MP-AMT randomly selects an MR and a source test case belonging to a partition that is selected according to the test profile, generating the follow-up test case depend on the source test case, and then updates the test profile according to the result of test execution. Second,

Algorithm 5 PP-AMT

Input: $\epsilon, p_1, \ldots, p_m, MR_{s_1}, \ldots, MR_{s_m}, \mathcal{R}, counter$

1: Initialize $counter = 1$.
2: **while** termination condition is not satisfied **do**
3: **if** $counter = 1$ **then**
4: Randomly select an MR from $\mathcal{R}$, and $MR \in MR_{s_i}$.
5: Increment counter by 1.
6: Randomly select a source test case from the partition s_i, and generate the follow-up test case belonged to partition s_f ($f \in 1,2,\ldots,m$).
7: Execute the source and follow-up test cases.
8: **if** the results of source and follow-up test cases violate MR **then**
9: **if** $i = f$ **then**
10: Update the test profile according to Formulas 1 and 2.
11: **else**
12: Update the test profile according to Formulas 8, 9, and 10.
13: **end_if**
14: **else**
15: **if** $i = f$ **then**
16: Update the test profile according to Formulas 3 and 4.
17: **else**
18: Update the test profile according to Formulas 11, 12, and 13
19: **end_if**
20: **end_if**
21: **else**
22: Select a partition s_i according to the updated test profile, and then select an $MR^{'}$ from MR_{s_i}.
23: Randomly select a source test case from the partition s_i, and generate the follow-up test case belonged to partition s_f ($f \in 1,2,\ldots,m$).
24: Execute the source and follow-up test cases.
25: **if** the results of source and follow-up test cases violate $MR^{'}$ **then**
26: **if** $i = f$ **then**
27: Update the test profile according to Formulas 1 and 2.
28: **else**
29: Update the test profile according to Formulas 8, 9, and 10.
30: **end_if**
31: **else**
32: **if** $i = f$ **then**
33: Update the test profile according to Formulas 3 and 4.
34: **else**
35: Update the test profile according to Formulas 11, 12, and 13
36: **end_if**
37: **end_if**
38: **end_if**
39: **end_while**

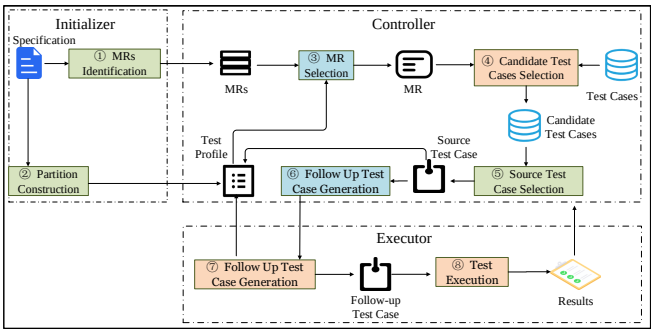

Fig. 3. The Framework of MP-AMT

a partition is selected according to updated test profile, and an MR is randomly selected from the set of MRs whose source test case belong to selected partition.

Suppose that the input domain of SUT is divided into m ($m > 2$) partitions $(s_1, s_2, \ldots, s_m)$. A set MR_{s_i} included metamorphic relations whose source test cases belong to the partition s_i, and $\mathcal{R} = \bigcup_{i=1}^{m} MR_{s_i}$ is a set that include all of metamorphic relations. The detailed MP-AMT algorithm is given in Algorithm 6. In MP-AMT, the first metamorphic relation MR is randomly selected from $\mathcal{R}$, and a source test case stc is randomly selected from a partition s_i related to the selected metamorphic relation, then the follow-up test case ftc is generate based on MR and stc, belonging to partition s_f ($f \in \{1, 2, \ldots, m\}$) (Lines 3 to 7 in Algorithm 6). If the results of stc and ftc violate the MR, there are following two situation, denoted δ_1, δ_2, respectively (Lines 8 to 12):

Situation 1 (δ_1): If $i = f$, then the test profile is updated according to Formulas 1 and 2.
Situation 2 (δ_2): If $i \neq f$, then the test profile is updated according to Formulas 8, 9, and 10.

Alternatively, if their results satisfy the MR, there are following two situation, denoted $\delta_1^{'}, \delta_2^{'}$, respectively (Lines 14 to 18):

Situation 1 ($\delta_1^{'}$): If $i = f$, then the test profile is updated according to Formulas 3 and 4.
Situation 2 ($\delta_2^{'}$): If $i \neq f$, then the test profile is updated according to Formulas 11, 12, and 13.

Next, MP-AMT randomly selects a partition according to updated test profile, then a source test case is randomly selected from the selected partition and an metamorphic relation is randomly selected from a MR_{s_i} ($i \in \{1, 2, \ldots, m\}$) whose source test cases belong to the selected partition. On the basis of MR_{s_i}, the follow-up test case is generated from the source test case (Lines 22 and 23). After the execution of the source and follow-up test cases, their results are verified against the MR_{s_i}, then the test profile is updated (Lines 24 to 35). This process is repeated until a termination condition is satisfied (Line 2).

We developed a tool called AMTester to the best of our knowledge. AMTester has features such as termination condition setting, partition setting, test execution, and test report generation.

Algorithm 6 MP-AMT

Input: $\epsilon, p_1, \ldots, p_m, MR_{s_1}, \ldots, MR_{s_m}, \mathcal{R}, counter$

1: Initialize $counter = 1$.
2: **while** termination condition is not satisfied **do**
3: **if** $counter = 1$ **then**
4: Randomly select an MR from $\mathcal{R}$, and $MR \in MR_{s_i}$.
5: Increment counter by 1.
6: Randomly select a source test case from the partition s_i, and generate the follow-up test case belonged to partition s_f ($f \in 1, 2, \ldots, m$).
7: Execute the source and follow-up test cases.
8: **if** the results of source and follow-up test cases violate MR **then**
9: **if** $i = f$ **then**
10: Update the test profile according to Formulas 1 and 2.
11: **else**
12: Update the test profile according to Formulas 8, 9, and 10.
13: **end_if**
14: **else**
15: **if** $i = f$ **then**
16: Update the test profile according to Formulas 3 and 4.
17: **else**
18: Update the test profile according to Formulas 11, 12, and 13
19: **end_if**
20: **end_if**
21: **else**
22: Select a partition s_i according to the updated test profile, and then select an $MR^{'}$ from MR_{s_i}.
23: Randomly select a source test case from the partition s_i, and generate the follow-up test case belonged to partition s_f ($f \in 1, 2, \ldots, m$).
24: Execute the source and follow-up test cases.
25: **if** the results of source and follow-up test cases violate $MR^{'}$ **then**
26: **if** $i = f$ **then**
27: Update the test profile according to Formulas 1 and 2.
28: **else**
29: Update the test profile according to Formulas 8, 9, and 10.
30: **end_if**
31: **else**
32: **if** $i = f$ **then**
33: Update the test profile according to Formulas 3 and 4.
34: **else**
35: Update the test profile according to Formulas 11, 12, and 13
36: **end_if**
37: **end_if**
38: **end_if**
39: **end_while**

3.6 Random Based AMT (R-AMT)

Random Testing (RT) tests software by randomly generating inputs, which is a standard black-box software testing technique. The IEEE Guide to the Software Engineering Body of Knowledge (SWEBOK) [53] lists RT as one of four common input-domain based testing techniques, along with equivalence partitioning, pairwise testing, and boundary value analysis. Arcuri et al. [54] observe that RT is one of the most used automated testing techniques in practice. Therefore, we particularly develop one algorithm, random based AMT (R-AMT), to implement the proposed AMT. The framework of R-AMT is illustrated in Figure 4.

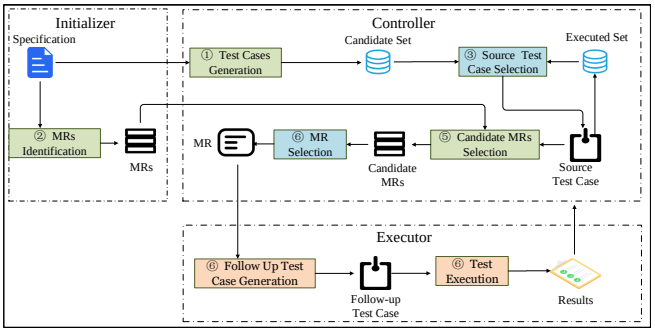

Fig. 4. The Framework of R-AMT

In R-AMT, we make use the ARTsum algorithm to generate source test cases. Accordingly, a set of candidate MRs $\mathcal{CR}$ can be obtained according to the selected source test case. Then, an MR is selected by using the proposed RMRS or PBMR strategy. The details of R-AMT is given in Algorithm 7

Algorithm 7 R-AMT

Input: $\mathcal{R}$

1: **while** termination condition is not satisfied **do**
2: Generate a source test case stc_i using the Algorithm 2.
3: Select the MR from $\mathcal{R}$, whose source test case can be stc_i and add the selected MR to $\mathcal{CR}$
4: Select an MR MR_s form $\mathcal{CR}$ using the proposed strategy RMRS or PBMR.
5: Generate the follow-up test case ftc_i according to stc_i and MR_s.
6: Test the software using the stc_i and ftc_i.
7: Verify the outputs of stc_i and ftc_i against the MR_s.
8: **end_while**

4 Empirical Study

We conducted a series of empirical studies to evaluate the performance of DRT.

4.1 Research Questions

In our experiments, we focused on addressing the following three research questions:

RQ1 **How efficient is AMT at detecting faults?**
Fault-detection efficiency is a key criterion for evaluating the performance of a testing technique. In our study, we chose three real-life programs, and applied mutation analysis to evaluate the fault-detecting efficiency.

RQ2 **What is the actual test case selection overhead when using the AMT technique?**
We evaluate the test case selection overhead of M-AMT and compare with traditional MT in detecting software faults.

4.2 Object Programs

In order to evaluate the fault-detection effectiveness of proposed methods in different scales, different implementation languages, and different fields, we chose to study four sets of object programs: three laboratorial programs that were developed according to corresponding specifications, seven programs from the Software-artifact Infrastructure Repository (SIR) [55], the regular expression processor component of the larger utility program `GUN grep`, and a Java library developed by Alibaba, which can be used to convert Java Objects into their JSON representation, and convert a JSON string to an equivalent Java object. The details of twelve programs are summarized in Table 1.

There four sets of programs present complementary strengths and weaknesses as experiment objects. The Laboratorial programs implement simple functions and their interfaces are easy to understand. The test engineers can easily generate source test cases, and identify MRs for testing laboratorial programs. However, these programs are small and there are a limited number of faulty versions available for each programs. The SIR repositories provides object programs, including a number of pre-existing versions with seeded faults, as well as a test suite in which test cases were randomly generated. It is a challenge task to partition the input domain of those programs. The `grep` program is a much larger system for which mutation faults could be generated. The `FastJson` is also a much larger system, and the faults of that are obtained on GitHub. We provide further details on each of those sets of object programs next.

4.2.1 Laboratorial Programs

Based on three real-lift specifications, We developed three systems, respectively. `China Unicom Billing System (CUBS)` provides an interface through which customers can know how much they need to pay according to plans, month charge, calls, and data usage. The details of two cell-phone plans are summarized in Tables 2 and 3. `Aviation Consignment Management System (ACMS)` aims to help airline companies check the allowance (weight) of free baggage, and the cost of additional baggage. Based on the destination, flights are categorized as either domestic or international. For international flights, the baggage allowance is greater if the passenger is a student (30kg), otherwise it is 20kg. Each aircraft offers three cabins classes from which to choose (economy, business, and first), with passengers in different classes having different allowances. The detailed price rules are summarized in Table 4, where $price_0$ means economy class fare. `Expense Reimbursement System (ERS)` assists the sales Supervisor of a company with handling the following tasks: i)

TABLE 1
Twelve Programs as Experimental Objects

programs	Source	Language	LOC	All Faults	Used Faults	Number of MRs	Number of Partitions
CUBS	Laboratory	Java	107	187	4	184	8
ACMS	Laboratory	Java	128	210	9	142	4
ERS	Laboratory	Java	117	180	4	1130	12
grep	GUN	c	10,068	20	20	12	–
printtokens	SIR	c	483	7	7	3	–
printtokens2	SIR	c	402	10	10	3	–
replace	SIR	c	516	31	31	3	–
schedule	SIR	c	299	9	9	3	–
schedule2	SIR	c	297	9	9	3	–
tcas	SIR	c	138	41	41	3	–
totinfo	SIR	c	346	23	23	3	–
FastJson	Alibaba	Java	204125	6	6	–	–

Calculating the cost of the employee who use the cars based on their titles and the number of miles actually traveled; ii) accepting the requests of reimbursement that include airfare, hotel accommodation, food and cell-phone expenses of the employee.

TABLE 2
Plan A

Plan details		Month charge (CNY)			
		$option_1$	$option_2$	$option_3$	$option_4$
ExtraBasic	Free calls (min)	50	96	286	3000
	Free data (MB)	150	240	900	3000
	Dialing calls (CNY/min)	0.25	0.15	0.15	0.15
	Data (CNY/KB)	0.0003			

TABLE 3
Plan B

Plan details		Month charge (CNY)			
		$option_1$	$option_2$	$option_3$	$option_4$
ExtraBasic	Free calls (min)	120	450	680	1180
	Free data (MB)	40	80	100	150
	Dialing calls (CNY/min)	0.25	0.15	0.15	0.15
	Data (CNY/KB)	0.0003			

4.2.2 SIR Programs

The seven object programs selected from the SIR [24] repository were printtokens, printtokens2, replace, schedule, schedule2, tcas, and totinfo. These programs were originally compiled by Hutchins et al. [56] at Siemens Corporate Research. We used these programs for several reasons:

1 Faulty versions of the programs are available.
2 The programs are of manageable size and complexity for an initial study.
3 All programs and related materials are available from the SIR, and the MRs of each program are defined in [57].

The printtokens and printtokens2 are independent implementations of the same specification. They each implement a lexical analyzer. Their input files are split into lexical tokens according to a tokenizing scheme, and their output is the separated tokens. The replace program is a command-line utility that takes a search string, a replacement string, and an input file. The search string is a regular expression. The replacement string is text, with some metacharacters to enable certain features. The replace program searches for occurrences of the search string in the input file, and produces an output file where each occurrence of the search string is replaced with the replacement string. The schedule and schedule2 programs are also independent implementations of the same specification. They each implement a priority scheduler that takes three non-negative integers and an input file. The integers indicate the number of initial processes at the three available scheduling priorities. The schedule and schedule2 programs then take as input a command file that specifies certain actions to be taken. The tcas program is a small part of a much larger aeronautical collision avoidance system. It performs simple analysis on data provided by other parts of the system and returns an integer indicating what action, if any, the pilot should take to avoid collision.

4.2.3 GUN grep

The used version of the grep programs is 2.5.1a [58]. This program searches one or more input files for lines containing a match to a specified pattern. By default, grep prints the matching lines. We chose grep for our study for several reasons:

1 grep program is wide used in Unix system, providing a opportunity to demonstrate the real world relevance of our techniques.
2 grep program, and its input format, are of greater complexity than the the programs in the other test sets, but still manageable as a target for automated test case generation.

The inputs of the grep were categorize into three components: options, which consist of a list of commands to modify the searching process, pattern, which is the regular expression to be searched for, and files, which refers to the input files to be searched.

The scope of functionality of this program is larger, which leads to construct test infrastructure to test all of functionality would have been impractical. Therefore, we restricted our focus to the regular expression analyzer of the grep.

4.2.4 Real-World Popular Programs

FastJson is a Java library that can be used to convert Java Objects into their JSON representation. It can also be used to

TABLE 4
ACMS Baggage Allowance and Pricing Rules

	Domestic flights			International flights		
	First class	Business class	Economy class	First class	Business class	Economy class
Carry on (kg)	5	5	5	7	7	7
Free checked-in (kg)	40	30	20	40	30	20/30
Additional baggage pricing (kg)	$price_0 * 1.5\%$			$price_0 * 1.5\%$		

convert a JSON string to an equivalent Java object. We chose `FastJson` for our study for several reason:

1 `FastJson` is wide used in real-word projects, providing a opportunity to demonstrate the real world relevance of our techniques.
2 `FastJson` is more complex than other programs, and it is a open source project, that is, we can obtain its faults freely and conveniently.
3 The chosen version of `FastJson` is 1.2.48, which was reported that had many faults that can support our research.

4.3 Variables

4.3.1 Independent Variables

The independent variable in our experiment is the different strategies that improve the fault-detection efficiency. As choices for this variable, we include, of course, proposed PP-AMT, MP-AMT, and R-AMT. As baseline techniques for use in comparison, we selected two additional techniques, traditional MT and MT-ARTsum.

TABLE 5
The Details of Independent Variables

Techniques		The strategy of selecting Source Test Case	The strategy of selecting MRs
Proposed Techniques	PP-AMT	partition	RMRS
	PP-AMT*	partition	PBMR
	MP-AMT	partition	RMRS
	MP-AMT*	partition	PBMR
	R-AMT	ARTsum	RMRS
	R-AMT*	ARTsum	PBMR
Baseline Techniques	MT	Random	–
	MT-ARTsum	ARTsum	–

Table 5 summarizes the details information of selecting source test cases and MRs of different techniques. Traditional MT is a natural baseline choice, because all proposed techniques is designed as an enhancement to MT, and assessing whether proposed techniques including PP-AMT, MP-AMT, and R-AMT is more cost-effective than MT is important.

There exist some techniques, such as quasi-random testing (QRT) [46], RBCVT-Fast [49], ARTmif [47], and ARTsum [27] that can achieve a computation overhead as low as $O(n)$. However, QRT and RBCVT-Fast can be only to test software with an exclusively numeric input domain, and therefore could not be compared to proposed techniques. Moreover, Barus et al. [27] has been empirically shown that ARTsum had a much lower selection overhead than ARTmif, and its overhead was close to that of RT. Thus, we used ARTsum to select the source test cases of MT as one of baseline techniques.

4.3.2 Dependent Variables

The choice of a metric to use in comparing the effectiveness of testing techniques is non-trivial.

The dependent variable for RQ1 is the metric for evaluating the fault-detection effectiveness. Several effectiveness metrics exist, including: the P-measure [59] (the probability of at least one fault being detected by a test suite); the E-measure [60] (the expected number of faults detected by a test suite); the F-measure [29] (the expected number of test case executions required to detect the first fault); and the T-measure [61] (the expected number of test cases required to detect all faults). Since the F- and P-measures have been widely used for evaluating the fault-detection efficiency and effectiveness of MT, ART, and DRT-related testing techniques [16], [27], [29], they are also adopted in this study. We use F and P to represent the F-measure and the P-measure of a testing technique.

We define $F\text{-}count$ as the number of test cases needed to detect a failure in a specific test run. The F-measure is the expected $F\text{-}count$ for a testing method:

$$F = \overline{F - count}. \tag{14}$$

The F-measure is particularly appropriate for measuring the failure-detection effectiveness of adaptive testing methods, such as ART and DRT, in which the generation of new test cases depends on the previously executed test cases or the execution results of executed test case.

The P can characterize the testing process. Suppose that a particular method is used to generate a test suite with n test cases $\{t_1, t_2, \ldots, t_n\}$, the P is defined as the probability of at least one failure being detected by the test suite:

$$P(n) = Prob(\exists t_i \text{ that reveals a failure}). \tag{15}$$

where $i = 1, 2, \ldots, n$. A larger P reflects better failure-detection effectiveness.

The rate of occurrence of failures (ROCOF) that is the probability that a failure (not necessarily the first) occurs in a given interval of executions, which is relevant for operating systems where the system has to process a large number of similar requests that are relatively frequent. ROCOF reflects the rate of occurrence of failure or anomalies in the system [62].

Holm-Bonferroni method [63] to determine which pair of testing techniques have significant difference in terms of F. Across the whole study, for each pair of testing techniques, denoted by technique a and technique b, the null hypothesis (H_0) was that a and b had similar performance in terms of one metric; whereas the alternative hypothesis (H_1) was that a and b had significantly different performance in terms of F. All the null hypotheses were ordered by their corresponding p-values, from lowest to largest; in

TABLE 6
Categories and choices of ACMS

Category	Associated choices
aircraft cabin	*Economy, Business, First Class*
flight region	*International, Domestic*
baggage weight	*Above Limit, Below Limit*

other words, for null hypotheses $H_0^1, H_0^2, \ldots, H_0^{h-1}$, we had $p_1 \leq p_2 \leq ...$ For the given confidence level $\alpha = 0.05$, we found the minimal index h such that $p_h > \frac{\alpha}{N+1-h}$ (where N is the total number of null hypotheses). Then, we rejected $H_0^1, H_0^2, \ldots, H_0^{h-1}$, that is, we regarded the pair of techniques involved in each of these hypotheses to have statistically significant difference in terms of a certain metric. On the other hand, we considered the pair of techniques involved in each of $H_0^h, H_0^{h+1}, ...$ not to have significant difference with respect to one metric, as these hypotheses were accepted.

An obvious metric for RQ2 is the time required to detect faults. Corresponding to the T-measure, in this study we used $F\text{-}time$ denote the time required (i.e. test cases generation, test cases selection, and test cases execution) to detect the first fault. Obviously, a smaller values of $F\text{-}time$ indicate a better performance.

4.4 Experimental Settings

4.4.1 Partitioning

The proposed techniques PP-AMT and MP-AMT both based on the partition testing that is a mainstream family of software testing techniques, which can be realized in different ways, such as Intuitive Similarity, Equivalent Paths, Risk-Based, and Specified As Equivalent (two test values are equivalent if the specification says that the program handles them in the same way) [64]. In this study, we made use of the CPM to conduct the partitioning. Our previous work [29] found that there is not a strong correlation between the granularity level in partitioning and the performance of APT. Therefore, we Select two categories that have fewer choices annotated with [single] or [error], and partition the input domain according to the combinations of their choices.

We use an explanatory example to illustrate the method of partitioning. According to the description of Section 4.2.1, we identify categories and associated choices of `ACMS`, as shown in Table 6. With these categories/choices and constraints among choices, we further derive a set of complete test frames and each of them corresponds to a partition. Note that the number of partitions may vary with the granularity level. To ease the illustration, we derive partitions without considering the baggage weight category. Table 7 shows the resulting partitions for BB, where $weight_*$ indicates no classification on the baggage weight.

4.4.2 Initial Test Profile

The proposed techniques PP-AMT and MP-AMT make use of feedback information to adjust the test profile. Then the updated test profile guide those techniques to select next source test case or MR. Before executing the test, a concrete test profile should be initialized. In our previous work [29], we have compared the equal and proportional initial test profile in terms of associated performance metrics (F-measure, F2-measure [29], and T-measure). The comparison results that there was no significant difference between these two types of initial test profiles. And as a result, we used a uniform probability distribution as the initial testing profile in this study.

TABLE 7
Partitions of ACMS

Partition	Complete Test Frame
c_1	$\{International, Economy, weight_*\}$
c_2	$\{International, Business, weight_*\}$
c_3	$\{International, FirstClass, weight_*\}$
c_4	$\{Domestic, Economy, weight_*\}$
c_5	$\{Domestic, Business, weight_*\}$
c_6	$\{Domestic, FirstClass, weight_*\}$

4.4.3 Constants

In PP-AMT and MP-AMT, there is a parameter ϵ that is a probability adjusting factor. We followed studies [65]–[67] to set $\epsilon = 0.05$.

In R-AMT, ARTsum is employed to generate source test cases, which has a parameter k — the size of the candidate set. Previous work [23] has shown that at least for numeric programs — that failure detection effectiveness improves as k increase up to about 10, and then does not improve much further. In addition, Barus [27] applied ARTsum to non-numeric programs where the value of k was set 10. Therefore, our experiments were all conducted with k set to 10.

4.5 Generation of Categories and Choices for Object Programs

The categories and choices used for the laboratorial programs and Real-World Popular Programs considered in this study were designed by the authors. On the other hand, the categories and choices used for the SIR Programs and GUN Program were designed by Barus [27]. In large part, the selection of appropriate categories and choices is at a testers discretion; we chose what we regarded as simple approaches for emulating that process. Precise details on the categories and choices used in our study are provided as following.

4.5.1 The categories and choices of Laboratorial Programs

Precise details on the categories and choices of SIR and GUN programs are provided in Tables A4 to A15 in the Appendices, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TC.2016.2547380.

4.6 Generation of Test Cases for Object Programs

4.6.1 Generation of Test Cases for Laboratorial Programs

For `CUBS`, `ACMS`, and `ERS`, we used a generator that is based on the complete test frame devised for MT-related techniques selection. We systematically generated a test case for each complete test frame, which were collectively guaranteed to cover each category and choice. The final test suites contained 284, 1470, and 2260, respectively.

TABLE 8
Definition of Categories and Choices for CUBS

#	Category	Choice
1	plan	A
		B
2	options	46CNY
		96CNY
		286CNY
		886CNY
		126CNY
		186CNY
3	calls	calls $<$ free calls
		calls $\geq$ free calls
4	data	data $<$ free data
		data $\geq$ free data

TABLE 9
Definition of Categories and Choices for ACMS

#	Category	Choice
1	class	first calss
		Business class
		Economy class
		infant
2	region	Domestic flights
		International flights
3	isStudent	True
		False
4	luggage	luggage $<$ Free checked-in
		luggage $\leq$ Free checked-in
5	fee	fee $= 0$
		fee ≤ 0

4.6.2 Generation of Test Cases for SIR Programs

Each of the SIR programs had an existing pool of test cases, but these pools were not large enough (having a few thousand test cases per program) to ensure sufficient randomness. Thus, rather than sampling test cases from the existing pools, we used a number of techniques to dynamically generate test cases on demand. Our approach has some similarities to fuzz testing. We first analyzed the existing test pools to obtain the probability distributions of certain parameters. Then, according to the probability distributions, the concrete values of these parameters could be randomly chosen. The detailed procedure for test case generation for each object program can be found in Appendix A, available in the online supplemental material.

TABLE 10
Definition of Categories and Choices for ERS

#	Category	Choice
1	title	senior sales manager
		sales manager
		sales executive
2	mileage	$0 \leq mileage \leq 3000$
		$3000 \leq mileage \leq 4000$
		$mileage \geq 4000$
3	sales	$0 \leq sales \leq 50,000$
		$50,000 \leq sales \leq 80,000$
		$80,000 \leq sales \leq 100,000$
		$sales \geq 100,000$
4	airline ticket	$airlineticket = 0$
		$airlineticket \leq 0$
5	other expenses	$otherexpenses = 0$
		$otherexpenses \leq 0$

TABLE 11
Definition of Categories and Choices for FastJson

#	Category	Choice

4.6.3 Generation of Test Cases for GUN Program

For grep, we used a generator that was itself based on the categories and choices devised for ART selection. We systematically generated random candidate test cases, which were collectively guaranteed to cover each category and choice. Our test generator does not randomly sample from the entire input domain of grep; rather, it samples a small subset of the input space, as our purpose is to test the regular expression analyzer of grep. We further filtered the randomly generated pool to remove duplicate entries. The final pool contained 171, 634 elements. Readers can refer to Appendix B, available in the online supplemental material, for more technical details on the random test case generation process for `grep`.

4.6.4 Generation of Test Cases for Alibaba Program

4.7 Identification of MRs for Object Programs

4.7.1 The MRs of Laboratorial Programs

In our study, to obtain metamorphic relations, we made use of METRIC [68], which is based on the categoryCchoice framework [38]. In METRIC, only two distinct complete test frames that are abstract test cases defining possible combinations of inputs, are considered by the tester to generate an MR. In general, METRIC has the following steps to identify MRs:

Step1. Users select two relevant and distinct complete test frames as a *candidate pair*.
Step2. Users determine whether or not the selected candidate pair is useful for identifying an MR, and if it is, then provide the corresponding MR description.
Step3. Restart from step 1, and repeat until all candidate pairs are exhausted, or the predefined number of MRs to be generated is reached.

Following the above guidelines, we identified MRs for CUBS, ACMS, and ERS. Table 12 presents a part of MRs of those programs.

TABLE 12
Part of MRs of Laboratorial Programs

Program	MRs		
	Source Test Cases	Follow-up Test Cases	Relations
CUBS	$\{1a, 2a, 3a, 4a\}$	$\{1a, 2a, 3a, 4b\}$	$O_1 \geq O_2$
	$\{1a, 2a, 3a, 4a\}$	$\{1a, 2a, 3b, 4a\}$	$O_1 \geq O_2$
	$\{1a, 2a, 3a, 4a\}$	$\{1a, 2a, 3b, 4b\}$	$O_1 \geq O_2$
	$\{1a, 2a, 3a, 4a\}$	$\{1a, 2b, 3a, 4a\}$	$O_1 \geq O_2$
	$\{1a, 2a, 3a, 4a\}$	$\{1a, 2b, 3a, 4b\}$	$O_1 \geq O_2$
ACMS	$\{1a, 2a, 3a, 4a, 5a\}$	$\{1a, 2a, 3a, 4a, 5b\}$	$O_1 = O_2 = 0$
	$\{1a, 2a, 3a, 4a, 5a\}$	$\{1a, 2a, 3a, 4b, 5a\}$	$O_1 = O_2 = 0$
	$\{1a, 2a, 3a, 4a, 5a\}$	$\{1a, 2b, 3a, 4a, 5a\}$	$O_1 = O_2 = 0$
	$\{1a, 2a, 3a, 4a, 5a\}$	$\{1b, 2a, 3a, 4a, 5a\}$	$O_1 = O_2 = 0$
	$\{1a, 2a, 3a, 4a, 5a\}$	$\{1c, 2a, 3a, 4a, 5a\}$	$O_1 = O_2 = 0$
CUBS	$\{1a, 2a, 3a, 4a\}$	$\{1a, 2a, 3a, 4b\}$	$O_1 \leq O_2$
	$\{1a, 2a, 3a, 4a\}$	$\{1a, 2a, 3b, 4a\}$	$O_1 = O_2 = 0$
	$\{1a, 2a, 3a, 4a\}$	$\{1a, 2a, 3c, 4a\}$	$O_1 = O_2 = 0$
	$\{1a, 2a, 3a, 4a\}$	$\{1a, 2b, 3a, 4a\}$	$O_1 = O_2 = 0$
	$\{1a, 2a, 3a, 4a\}$	$\{1a, 2c, 3a, 4a\}$	$O_1 \geq O_2$

4.7.2 The MRs of SIR Programs

$Print\_tokens$ and $print\_tokens2$. These two programs perform lexical parsing. They both read a sequence of strings from a file, group these strings into tokens, identify token categories and print out all the tokens and their categories in order. The main difference between these two programs is that $print\_tokens$ uses a hard-coded DFA; while $print\_tokens2$ does not. Suppose the input files in source test case and follow-up test case are denoted as I_s and I_f, respectively, and their respective outputs are denoted as O_s and O_f. Each element in O_s and O_f has two attributes: the token category (e.g. keyword, identifier, etc.) and the string of this token. For these two programs, we define MRs as follows.

1 **MR1: Changing lower case into upper case**. In this MR, If is constructed from Is by changing all characters in Is with lower cases into their upper cases. Since $print\_tokens$ attempts to identify tokens and their categories, we have the size of O_f equal to the size of O_s. Besides, since all "keywords" are case-sensitive, all the elements with categories of "keyword" in O_s become "identifier" in O_f. For the non-keyword elements of O_s the corresponding categories remain the same in O_f.

2 **MR2: Deletion of comments**. In this MR, If is constructed from I_s by deleting all comments in I_s. Then, we have $O_s = O_f$.

3 **MR3: Insertion of comments**. In this MR, If is constructed from Is by inserting the comment symbol ";" at the very beginning of some arbitrarily chosen lines. Then, we have $O_f \subseteq O_s$.

`replace` performs regular expression matching and substitutions. It takes a regular expression r, a replacement string s and an input file as input parameters. It produces an output file resulting from replacing any substring in the input file that matched by r, with s. Instead of adopting the widely used Perl regular expression syntax, `replace` has its own syntax of regular expression.

1 **MR1: Extension and permutation in brackets**. The syntax of regular expression in replace also supports the bracketed sub-expression "$[x_1 \dots x_n]$", where x_i is a single character. Such an expression has an equivalent format "$[x1Cxn]$", if "$x1, \dots, xn$" are continuous for the current locale. Thus, this MR constructs r_f with two types of replacements.

- The condensed sub-expression "$[x_1$ C- $x_n]$" in r_s is replaced by its extended equivalent "$[x_1 \dots x_n]$".
- The extended sub-expression "$[x_1 \dots x_n]$" in r_s is replaced by "$[x_{i1} \dots x_{in}]$", where "$[x_{i1} \dots x_{in}]$" is a permutation of "$[x_1 \dots x_n]$".

2 **MR2: Bracketing simple characters**. Apart from the reserved words with special meanings, any simple character should be equivalent to itself enclosed by the brackets, that is, "a" is equivalent to "[a]" if a is not a reserved word. r_f is constructed by replacing some simple characters in r_s with their bracketed formats. For example, if r_s contains a sub-expression "abc", then we have "[a][b][c]" instead in r_f.

3 **MR3: Replacement of escape character with bracketed structure**. In `replace`, symbol "@" is the escape character. Any reserved character, like "$" or "%" after "@" stands for its original meaning. Actually there is another way to preserve the original meaning for these reserved words, by bracketing the reserved characters individually. For example "$ and "[$]" both mean "$". r_f is constructed by replacing the escaped reserved words with "@" in r_s with the bracketed format.

`Schedule` and `schedule2`. These two programs perform priority scheduling, which internally maintain four mutually exclusive job lists:

- Three priority job-lists P_1, P_2 and P_3, with P_3 and P_1 indicating the highest and lowest priorities, respectively: each list contains a list of jobs with the same priority.
- One blocked job list PB: it contains all jobs currently suspended.

Given the same a_i ($1 \leq i \leq 3$), the input files containing a series of commands in the source test case and follow-up test case are denoted as C_s and C_f, respectively. For schedule, we define the MRs as follows.

1 **MR1: Substitution of the quantum expire command**. In schedule, command coded in integer "5" is called "$QUANTUM\_EXPIRE$" command, which releases the currently activated job and puts it to the end of the corresponding job list. Command coded in integer "3" is called "BLOCK" command, which puts the currently activated job to the block list PB. Its reverse command "UNBLOCK" command is coded in "4" that takes a ratio r as a parameter. The index of the selected job to be unblocked from PB is determined by multiplying the length of PB by the ratio r. When processing command "4 r", schedule first releases the selected job from PB, and then put it to the end of the corresponding job list. Therefore, command 5 can be re-interpreted as command "3", followed by "4 1.00". By using "3" and "4 1.00", schedule first blocks the currently activated job (denoted as pa), puts it at the end of PB, then unblocks the last job in PB, i.e. pa, (since the length of PB multiplied by 1.00 indicates the last index in PB) and puts this job to the end of the corresponding job list. Obviously, consecutively processing these two commands has the same effect as solely processing command "5". Thus, C_f is constructed by replacing command "5" in C_s with commands "3" and "4 1.00". The output of the follow-up test case (denoted as O_f) should be the same as the output of the source test case (denoted as O_s).

2 **MR2: Substitution of the adding job command**. In schedule, command coded in "1" is called "NEW_JOB" command, which takes an integer i ($1 \leq i \leq 3$) as its parameter to specify the priority, and adds a new job to the end of P_i. And command coded in "2" is the "$UPGRADE\_PRIO$" command, which promotes a job from its current priority job list (P_i) into the next higher priority job list (P_{i+1}), where $i = 1$ or 2. This command takes two parameters. The first one is an integer i of 1 or 2, which specifies the current priority job list P_i. The second parameter is a ratio r, and the index of the selected job in P_i to be upgraded is determined by multiplying the length of P_i by the ratio r. As a consequence, directly adding a new job with priority $(i + 1)$ has the same effect as adding a job with priority i and then promoting it to

the job list with priority $(i+1)$. Thus, C_f is constructed by replacing command "1 $i+1$" in C_s with command "1 i" followed by "2 i 1.00" ($i = 1$ or 2). Of should be the same as O_s.

3 **MR3: Substitution of block and unblock commands**.

The details of MRs can be obtained in [57].

4.7.3 The MRs of GUN Program

We used MRs for `grep` that have been defined in [45].

4.7.4 The MRs of Alibaba program

The MRs of `FastJason` have not been defined.

4.8 The Faults

4.8.1 The faults of Laboratorial Programs

We used mutation analysis [26], [69], [70] to generate a total of 1563 mutants. Each mutant was created by applying a syntactic change (i.e. mutation operator) to the original program. We removed equivalent mutants, and mutants that were too easily detected — deleting mutants that could be detected with less than 20 randomly generated test cases.

4.8.2 The faults of SIR programs

SIR has several released mutants for used SIR programs.

4.8.3 The faults of GUN program

The details of mutants for `grep` can be found in [45].

4.8.4 The faults of Alibaba program

The faults can be found on the GitHub at https://github.com/alibaba/fastjson.

4.9 Experimental Environment

Our experiments were conducted on a virtual machine running the Ubuntu 18.04 64-bit operating system. In this system, there were two CPUs and a memory of 2GB. The test scripts were generated using Java, bash shell, and Python. In the experiments, we repeatedly ran the testing using each technique 30 times [71] with different random seeds to guarantee the statistically reliable mean values of the metrics.

4.10 Experiment Architecture

4.11 Threats To Validity

4.11.1 Internal Validity

4.11.2 External Validity

4.11.3 Construct Validity

4.11.4 Conclusion Validity

5 Experimental Results

6 Related Work

In this section, we describe related work from two perspectives: related to testing techniques for web services; and related to improving RT and PT.

Acknowledgment

This research is supported by the National Natural Science Foundation of China (Grant Nos. 61872039 and 61872167), 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).

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] S. S. Brilliant, J. C. Knight, and P. Ammann, "On the performance of software testing using multiple versions," in *Proceedings of the 20th International Symposium on Fault-Tolerant Computing (FTCS'90)*, 1990, pp. 408–415.

[5] 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.

[6] 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.

[7] K. Y. Sim, C. S. Low, and F.-C. Kuo, "Eliminating human visual judgment from testing of financial charting software," *Journal of Software*, vol. 9, no. 2, pp. 298–312, 2014.

[8] W. K. Chan, S.-C. Cheung, J. C. Ho, and T. Tse, "Pat: A pattern classification approach to automatic reference oracles for the testing of mesh simplification programs," *Journal of Systems and Software*, vol. 82, no. 3, pp. 422–434, 2009.

[9] C. Murphy, M. S. Raunak, A. King, S. Chen, C. Imbriano, G. Kaiser, I. Lee, O. Sokolsky, L. Clarke, and L. Osterweil, "On effective testing of health care simulation software," in *Proceedings of the 3rd workshop on software engineering in health care*. ACM, 2011, pp. 40–47.

[10] T. Y. Chen, J. W. Ho, H. Liu, and X. Xie, "An innovative approach for testing bioinformatics programs using metamorphic testing," *BMC bioinformatics*, vol. 10, no. 1, p. 24, 2009.

[11] Z. Hui, S. Huang, Z. Ren, and Y. Yao, "Metamorphic testing integer overflow faults of mission critical program: A case study," *Mathematical Problems in Engineering*, vol. 2013, 2013.

[12] Z. Q. Zhou, S. Zhang, M. Hagenbuchner, T. Tse, F.-C. Kuo, and T. Y. Chen, "Automated functional testing of online search services," *Software Testing, Verification and Reliability*, vol. 22, no. 4, pp. 221–243, 2012.

[13] 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.

[14] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," in *Proceedings of the 40th international conference on software engineering*. ACM, 2018, pp. 303–314.

[15] 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.

[16] A. C. Barus, T. Y. Chen, F.-C. Kuo, H. Liu, and H. W. Schmidt, "The impact of source test case selection on the effectiveness of metamorphic testing," in *Proceedings of the 1st International Workshop on Metamorphic Testing (MET'16), Co-located with the 38th International Conference on Software Engineering (ICSE'16)*. ACM, 2016, pp. 5–11.

[17] 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.

[18] Y. Cao, Z. Q. Zhou, and T. Y. Chen, "On the correlation between the effectiveness of metamorphic relations and dissimilarities of test case executions," in *Proceedings of the 13th International Conference on Quality Software (QSIC'13)*, 2013, pp. 153–162.

[19] 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.

[20] T. Chen, P. Poon, and X. Xie, "Metric: Metamorphic relation identification based on the category-choice framework," *Journal of Systems and Software*, vol. 116, pp. 190–14, 2014.

[21] X. Xie, J. Li, C. Wang, and T. Y. Chen, "Looking for an mr? try metwiki today," 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. 1–4.

[22] 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.

[23] T. Y. Chen, H. Leung, and I. Mak, "Adaptive random testing," in *Proceedings of the 9th Asian Computing Science Conference (ASIAN'04), ser. Lecture Notes in Computer Science*. Springer, 2004, pp. 320–329.

[24] K. P. Chan, T. Y. Chen, and D. Towey, "Restricted random testing," in *Proceedings of the 7th European Conference on Software Quality (ECSQ'02)*. Springer, 2002, pp. 321–330.

[25] ——, "Restricted random testing: Adaptive random testing by exclusion," *International Journal of Software Engineering and Knowledge Engineering*, vol. 16, no. 04, pp. 553–584, 2006.

[26] C. Mao, T. Y. Chen, and F.-C. Kuo, "Out of sight, out of mind: A distance-aware forgetting strategy for adaptive random testing," *Science China Information Sciences*, vol. 60, no. 9, pp. 092 106:1–092 106:21, 2017.

[27] A. C. Barus, T. Y. Chen, F.-C. Kuo, H. Liu, R. Merkel, and G. Rothermel, "A cost-effective random testing method for programs with non-numeric inputs," *IEEE Transactions on Computers*, vol. 65, no. 12, pp. 3509–3523, 2016.

[28] K. Cai, H. Hu, C.-h. Jiang, and F. Ye, "Random testing with dynamically updated test profile," in *Proceedings of the 20th International Symposium On Software Reliability Engineering (ISSRE'09)*, 2009, pp. 1–2.

[29] C.-A. Sun, H. Dai, H. Liu, T. Y. Chen, and K.-Y. Cai, "Adaptive partition testing," *IEEE Transactions on Computers*, vol. 68, no. 2, pp. 157–169, 2019.

[30] A. Groce, T. Kulesza, C. Zhang, S. Shamasunder, M. Burnett, W.-K. Wong, S. Stumpf, S. Das, A. Shinsel, F. Bice *et al.*, "You are the only possible oracle: Effective test selection for end users of interactive machine learning systems," *IEEE Transactions on Software Engineering*, vol. 40, no. 3, pp. 307–323, 2014.

[31] 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.

[32] G.-W. Dong, B.-W. Xu, L. Chen, C.-H. Nie, and L.-L. Wang, "Case studies on testing with compositional metamorphic relations," *Journal of Southeast University (English Edition)*, vol. 24, no. 4, pp. 437–443, 2008.

[33] U. Kanewala and J. M. Bieman, "Using machine learning techniques to detect metamorphic relations for programs without test oracles," in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2013, pp. 1–10.

[34] U. Kanewala, J. M. Bieman, and A. Ben-Hur, "Predicting metamorphic relations for testing scientific software: a machine learning approach using graph kernels," *Software testing, verification and reliability*, vol. 26, no. 3, pp. 245–269, 2016.

[35] 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.

[36] C.-a. Sun, Y. Liu, Z. Wang, and W. Chan, "μmt: A data mutation directed metamorphic relation acquisition methodology," in *2016 IEEE/ACM 1st International Workshop on Metamorphic Testing (MET)*. IEEE, 2016, pp. 12–18.

[37] T. Y. Chen, P.-L. Poon, and T. Tse, "A choice relation framework for supporting category-partition test case generation," *IEEE transactions on software engineering*, vol. 29, no. 7, pp. 577–593, 2003.

[38] 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.

[39] P. E. Ammann and J. C. Knight, "Data diversity: an approach to software fault tolerance," *IEEE Transactions on Computers*, vol. 37, no. 4, pp. 418–425, 1988.

[40] T. Y. Chen, F.-C. Kuo, and Z. Q. Zhou, "On favourable conditions for adaptive random testing," *International Journal of Software Engineering and Knowledge Engineering*, vol. 17, no. 06, pp. 805–825, 2007.

[41] T. Y. Chen and R. Merkel, "Quasi-random testing," *IEEE Transactions on Reliability*, vol. 56, no. 3, pp. 562–568, 2007.

[42] Y. Liu and H. Zhu, "An experimental evaluation of the reliability of adaptive random testing methods," in *2008 Second International Conference on Secure System Integration and Reliability Improvement*. IEEE, 2008, pp. 24–31.

[43] J. Mayer, "Lattice-based adaptive random testing," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. ACM, 2005, pp. 333–336.

[44] A. F. Tappenden and J. Miller, "A novel evolutionary approach for adaptive random testing," *IEEE Transactions on Reliability*, vol. 58, no. 4, pp. 619–633, 2009.

[45] A. C. Barus, "An in-depth study of adaptive random testing for testing program with complex input types," Ph.D. dissertation, Ph. D. Thesis. Faculty of Information and Communication Technologies , 2010.

[46] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "Artoo: Adaptive random testing for object-oriented software," in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 71–80.

[47] K. P. Chan, T. Chen, and D. Towey, "Forgetting test cases," in *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, vol. 1. IEEE, 2006, pp. 485–494.

[48] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and S. P. Ng, "Mirror adaptive random testing," *Information and Software Technology*, vol. 46, no. 15, pp. 1001–1010, 2004.

[49] A. Shahbazi, A. F. Tappenden, and J. Miller, "Centroidal voronoi tessellations-a new approach to random testing," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 163–183, 2013.

[50] H. Liu, F.-C. Kuo, D. Towey, and T. Y. Chen, "How effectively does metamorphic testing alleviate the oracle problem?" *IEEE Transactions on Software Engineering*, vol. 40, no. 1, pp. 4–22, 2014.

[51] P. E. Ammann and J. C. Knight, "Data diversity: An approach to software fault tolerance," *IEEE Transactions on Computers*, no. 4, pp. 418–425, 1988.

[52] G. B. Finelli, "Nasa software failure characterization experiments," *Reliability Engineering & System Safety*, vol. 32, no. 1-2, pp. 155–169, 1991.

[53] P. Bourque, R. E. Fairley *et al.*, *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0*. IEEE Computer Society Press, 2014.

[54] A. Arcuri, M. Z. Iqbal, and L. Briand, "Random testing: Theoretical results and practical implications," *IEEE Transactions on Software Engineering*, vol. 38, no. 2, pp. 258–277, 2012.

[55] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.

[56] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow-and control-flow-based test adequacy criteria," in *Proceedings of 16th International conference on Software engineering*. IEEE, 1994, pp. 191–200.

[57] 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.

[58] T. G. Project, "Grep home page. http://www.gnu.org/software/grep," 2006.

[59] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *IEEE transactions on Software Engineering*, no. 4, pp. 438–444, 1984.

[60] T. Y. Chen and Y.-T. Yu, "Optimal improvement of the lower bound performance of partition testing strategies," *IEE Proceedings-Software Engineering*, vol. 144, no. 5, pp. 271–278, 1997.

[61] L. Zhang, B.-B. Yin, J. Lv, K.-Y. Cai, S. S. Yau, and J. Yu, "A history-based dynamic random software testing," in *Proceedings of the 38th International Computer Software and Applications Conference Workshops (COMPSACW'14)*, 2014, pp. 31–36.

[62] M. R. Lyu *et al.*, *Handbook of software reliability engineering*. IEEE computer society press CA, 1996, vol. 222.

[63] S. Holm, "A simple sequentially rejective multiple test procedure," *Scandinavian journal of statistics*, pp. 65–70, 1979.

[64] C. Kaner and S. Padmanabhan, "An introduction to the theory and practice of domain testing," 2010.
[65] J. Lv, H. Hu, and K.-Y. Cai, "A sufficient condition for parameters estimation in dynamic random testing," in *Proceedings of the 35th IEEE Annual International Computer Software and Applications Conference Workshops (COMPSACW'11)*, 2011, pp. 19–24.
[66] Z. Yang, B. Yin, J. Lv, K.-Y. Cai, S. S. Yau, and J. Yu, "Dynamic random testing with parameter adjustment," in *Proceedings of the 38th IEEE Annual International Computer Software and Applications Conference Workshops (COMPSACW'14)*, 2014, pp. 37–42.
[67] Y. Li, B.-B. Yin, J. Lv, and K.-Y. Cai, "Approach for test profile optimization in dynamic random testing," in *Proceedings of the 39th International Computer Software and Applications Conference (COMPSAC15)*, vol. 3, 2015, pp. 466–471.
[68] 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.
[69] J. Chen, L. Zhu, T. Y. Chen, D. Towey, F.-C. Kuo, R. Huang, and Y. Guo, "Test case prioritization for object-oriented software: An adaptive random sequence approach based on clustering," *Journal of Systems and Software*, vol. 135, pp. 107–125, 2018.
[70] J. Chen, F.-C. Kuo, T. Y. Chen, D. Towey, C. Su, and R. Huang, "A similarity metric for the inputs of oo programs and its application in adaptive random testing," *IEEE Transactions on Reliability*, vol. 66, no. 2, pp. 373–402, 2017.
[71] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, 2011, pp. 1–10.

Chang-ai Sun is a Professor in the School of Computer and Communication Engineering, University of Science and Technology Beijing. Before that, he was an Assistant Professor at Beijing Jiaotong University, China, a postdoctoral fellow at the Swinburne University of Technology, Australia, and a postdoctoral fellow at the University of Groningen, The Netherlands. He received the bachelor degree in Computer Science from the University of Science and Technology Beijing, China, and the PhD degree in Computer Science from Beihang University, China. His research interests include software testing, program analysis, and Service-Oriented Computing.

Hepeng Dai is a PhD student in the School of Computer and Communication Engineering, University of Science and Technology Beijing, China. He received the master degree in Software Engineering from University of Science and Technology Beijing, China and the bachelor degree in Information and Computing Sciences from China University of Mining and Technology, China. His current research interests include software testing and debugging.

Tsong Yueh Chen is a Professor of Software Engineering at the Department of Computer Science and Software Engineering in Swinburne University of Technology. He received his PhD in Computer Science from The University of Melbourne, the MSc and DIC from Imperial College of Science and Technology, and BSc and MPhil from The University of Hong Kong. He is the inventor of metamorphic testing and adaptive random testing.

Huai Liu is a Lecturer of Information Technology at the College of Engineering & Science in Victoria University, Melbourne, Australia. Prior to joining VU, he worked as a research fellow at RMIT University and a research associate at Swinburne University of Technology. He received the BEng in physioelectronic technology and MEng in communications and information systems, both from Nankai University, China, and the PhD degree in software engineering from the Swinburne University of Technology, Australia. His current research interests include software testing, cloud computing, and end-user software engineering.