Feedback-Directed Metamorphic Testing

CHANG-AI SUN and HEPENG DAI, University of Science and Technology Beijing
HUAI LIU and TSONG YUEH CHEN, Swinburne University of Technology

Over the past decade, metamorphic testing has gained rapidly increasing attention from both academia and industry, particularly thanks to its high efficacy on revealing real-life software faults in a wide variety of application domains. On the basis of a set of metamorphic relations among multiple software inputs and their expected outputs, metamorphic testing not only provides a test case generation strategy by constructing new (or follow-up) test cases from some original (or source) test cases, but also a test result verification mechanism through checking the relationship between the outputs of source and follow-up test cases. Many efforts have been made to further improve the cost-effectiveness of metamorphic testing from different perspectives. Some studies attempted to identify "good" metamorphic relations, while other studies were focused on applying effective test case generation strategies especially for source test cases. In this article, we propose improving the cost-effectiveness of metamorphic testing by leveraging the feedback information obtained in the test execution process. Consequently, we develop a new approach, namely feedback-directed metamorphic testing, which makes use of test execution information to dynamically adjust the selection of metamorphic relations and selection of source test cases. We conduct an empirical study to evaluate the proposed approach based on four laboratory programs, one GNU program, and one industry program. The empirical results show that feedback-directed metamorphic testing can use fewer test cases and take less time than the traditional metamorphic testing for detecting the same number of faults. It is clearly demonstrated that the use of feedback information about test execution does help enhance the cost-effectiveness of metamorphic testing. Our work provides a new perspective to improve the efficacy and applicability of metamorphic testing as well as many other software testing techniques.

CCS Concepts: • **Software and its engineering → Software testing and debugging;**

Additional Key Words and Phrases: Metamorphic testing, metamorphic relation, test execution, feedback control, random testing, adaptive partition testing

ACM Reference format:
Chang-ai Sun, Hepeng Dai, Huai Liu, and Tsong Yueh Chen. 2023. Feedback-Directed Metamorphic Testing. *ACM Trans. Softw. Eng. Methodol.* 32, 1, Article 20 (February 2023), 34 pages.
https://doi.org/10.1145/3533314

This research is supported by the National Natural Science Foundation of China (Grant No. 61872039), the Australian Research Council Discovery Project (Grant No. DP210102447), 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-19-B19).
Authors' addresses: C. Sun and H. Dai, University of Science and Technology Beijing, 30 Xueyuan Road, Haidian District, Beijing 100083, China; emails: casun@ustb.edu.cn, daihepeng@sina.cn; H. Liu and T. Y. Chen, Swinburne University of Technology, John Street, Hawthorn, VIC 3122, Australia; emails: {hliu, tychen}@swin.edu.au.

1049-331X/2023/02-ART20 $15.00
https://doi.org/10.1145/3533314

1 INTRODUCTION

Software testing is a mainstream approach to software quality assurance that should be applied along the whole software development lifecycle. It helps to reduce the business risks by detecting software faults. In dynamic testing, some program inputs are generated as the *test cases*, which are then executed on the **software under test** (**SUT**). The actual execution results are verified against the expectations. Any difference between the actual and expected results implies a failure, which, in turn, indicates the presence of a fault in SUT. Extensive studies [6, 24, 30, 51, 62, 74] have been conducted to develop strategies for generating test cases that are effective in detecting various faults.

Lots of existing test case generation strategies have assumed, at least implicitly, the existence of a *test oracle*, which refers to a systematic mechanism for verifying the test result given any input. However, in many practical situations, the oracle either does not exist or is very difficult to obtain. The *oracle problem* [5, 54] significantly influences the applicability and effectiveness of most test case generation strategies. No matter how "good" the test cases are, they are virtually useless if their test results cannot be verified to decide whether a fault has been detected or not. A few techniques have been proposed to address the oracle problem, such as N-version programming [9], assertion [25], and **metamorphic testing** (**MT**) [15, 18, 59].

Among all these testing methods, MT [18, 59] contains both a test case generation strategy and a test result verification mechanism. Its core element is a set of ***metamorphic relations*** (**MRs**), which are the necessary properties of SUT in the form of relationships among multiple program inputs and their expected outputs. For test case generation, MRs are utilized to transform some original test cases (termed as *source test cases*) into new test cases (termed as *follow-up test cases*). On the other hand, after the execution of both source and follow-up test cases, their corresponding results are verified against MRs. The violation of an MR implies that there is a fault. Since its invention, MT has been popularly applied as an effective approach to the oracle problem and successfully detected lots of real-life bugs in various application domains and paradigms, such as bioinformatics [16], cybersecurity [20, 46, 66], RESTful Web APIs [60], artificial intelligence [48, 67, 73], machine translation software [31], natural language processing [44, 56], information visualization [50], computer vision [71], simulations [1, 32], datalog engines [47], and deep neural networks [14]. The high fault-detection effectiveness of MT is not only because it provides an alternative test result verification mechanism but also it can generate test cases that are complementary to those created by existing testing techniques [18, 23, 34, 40, 59].

Despite its popularity, there are still quite a few aspects for further improving MT. Among them, one is related to the cost-effectiveness; in this area, the state-of-the-art techniques can be classified into two major categories, namely, the identification/selection of "effective" MRs and the generation/selection of effective source test cases. For the former, Chen et al. [19] suggested that all available MRs should be used as part of the test strategy. Given that the resources for software testing are always limited, some "good" MRs should be selected for testing. For example, Chen et al. [17] proposed that good MRs should be those that can make the multiple executions of the program as different as possible. Mayer and Guderlei [49] found that MRs in the form of equalities or linear equations as well as those closely correlated to how programs are implemented had limited effectiveness, based on the investigation of six subject programs for matrix determinant computation with seeded faults. They further conjectured that good MRs could be selected based on the semantics of the program under test. Asrafi et al. [3] conducted a case study, which showed that the more different execution behaviors (measured by code coverage) the source and follow-up test cases had caused, the more effective MRs were in detecting faults. Just et al. [36] assessed the applicability of MT into the system and integration testing for an image encoder and observed

that the MRs derived from the components of a system were usually better at detecting faults than those derived from the whole system, and thus should be chosen for testing.

In addition to MRs, the quality of source test cases also influences MT's performance. Since follow-up test cases are constructed based on source test cases, previous studies of MT's test case generation were focused on the strategies of generating and/or selecting proper source test cases. In a survey, Segura et al. [59] reported that before 2016, 57% of MT work employed ***random testing*** (**RT**) to generate source test cases, and 34% selected source test cases from existing test pools. There exist a few studies that applied other testing techniques in MT's source test case generation/selection. For example, Barus et al. [7] employed an enhancement of RT, namely ***adaptive random testing*** (**ART**) [21], to select effective source test cases for MT. It was shown that ART did help MT to detect faults with fewer test cases than RT.

Although all above studies showed promising results in improving MT's performance from different perspectives, there still exist some challenges. For example, in previous work [3, 17, 19, 36, 49] of identifying effective MRs, all test cases were executed to measure the "goodness" of each MR, which incurred a high computing overhead. Although some systematic testing techniques were proposed to generate/select source test cases, the source test case generation/selection time and thus the overall testing time for MT were increased accordingly. There are cases where a lot of MRs can be identified, which may also lead to long testing time. Furthermore, even for a small set of identified MRs, some techniques, such as the MR composition [55], can be used to derive a large number of new MRs, which could achieve high testing effectiveness and efficiency. Given the large number of MRs, one way to improve the cost-effectiveness of MT is to select the most appropriate MRs at each step of testing. However, previous studies considered either the sole process of test case generation/selection or the identification of effective MRs, but not both at the same time. What is more, they have omitted the test execution process, which also contains some critical information useful for achieving a high cost-effectiveness.

In this article, we propose an innovative approach, namely ***feedback-directed metamorphic testing*** (**FDMT**), which makes use of the feedback information provided in test executions to guide the selection of both appropriate source test cases and MRs. In line with software cybernetics [11, 13, 69], FDMT treats the whole testing procedure as a feedback control system with feedback-directed strategies for selecting source test cases and MRs. It collects the online information during the test execution process, which, in turn, is feedbacked to dynamically control the processes of selecting appropriate test cases and MRs. Intuitively speaking, FDMT improves the cost-effectiveness of MT from the following three perspectives:

— Different from all previous studies of MT, FDMT employs a new testing technique, namely ***adaptive partition testing*** (**APT**) [62], to dynamically select "better" test cases based on the feedback information of test executions;
— Instead of arbitrarily selecting MRs, a new strategy, namely diversity-oriented strategy of MR selection (DOMR), is developed in FDMT to choose the most appropriate MR for each step of testing;
— A comprehensive FDMT framework delivers a strong integration of test case and MR selection as well as test execution, building up a systematic foundation for optimizing MT's cost-effectiveness.

Our study has the following three major contributions:

(1) We propose an innovative **FDMT** framework, which makes use of feedback information to simultaneously select source test cases and MRs that both have high probabilities of fault detection. FDMT provides a new perspective of implementing MT and improving its performance.

(2) Based on the proposed general FDMT framework, different techniques are developed. Two algorithms, namely MAPT* and RAPT*, are proposed for the source test case selection. MAPT* and RAPT* correspond to two variants of APT, **Markov-chain based APT** (**MAPT**) and **reward-punishment based APT** (**RAPT**), respectively. Due to the ability of addressing the oracle problem, the development of MAPT* and RAPT* also enhances the applicability of APT which assumes the existence of test oracle. MRs can be selected either using DOMR or in a random manner (termed as random MR selection strategy—RMRS in this study). The combinations of these techniques result in four variants of FDMT, namely MR-FDMT (MAPT* + RMRS), RR-FDMT (RAPT* + RMRS), MD-FDMT (MAPT*+ DOMR), and RD-FDMT (RAPT* + DOMR).
(3) The performance of FDMT is evaluated through a series of empirical studies on six programs, including four laboratory applications, GNU `grep`, and Alibaba `FastJson`. As observed from these experiments, FDMT has significantly higher cost-effectiveness than the traditional MT that randomly generates source test cases and arbitrarily uses MRs.

The rest of this article is organized as follows. Section 2 introduces the underlying concepts for MT, APT, and other relevant techniques. Section 3 presents the FDMT framework, the algorithms for the selection of source test cases and MRs. Section 4 describes the empirical studies for evaluation, the results of which are discussed in Section 5. Section 6 presents the related work and Section 7 concludes the article.

2 BACKGROUND

In this section, we present underlying concepts for MT. Also introduced are other techniques used in our FDMT approach, such as **category partition method** (**CPM**) and APT.

2.1 Metamorphic Testing (MT)

Basically speaking, MT is an effective technique to alleviate the oracle problem. Instead of applying an oracle, MT uses a set of MRs (the necessary properties of SUT) to verify the test results [18, 59] across multiple and related test cases. MT is normally implemented according to the following steps:

Step 1. Identify/select an MR for SUT.
Step 2. Generate the source test case(s) *stc* using some traditional test case generation technique.
Step 3. Construct the follow-up test case(s) *ftc* from the *stc* based on the MR.
Step 4. Execute *stc* and *ftc* and get their corresponding outputs O_s and O_f.
Step 5. 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 could be repeated for a set of MRs. Note that for some MRs, the construction of follow-up test cases may require the outputs of source test cases [18]. In this article, we assume that the follow-up test case construction process is independent of the execution results of source test cases, as most of MRs used in our study were constructed using an approach called METRIC [22], which also has a similar assumption.

The following simple example illustrates how MT works. Suppose a program $P(G, a, b)$ attempts to find the shortest path between nodes a and b in an undirected graph G. When G is nontrivial, the test result could be difficult to verify, that is, the oracle problem may be met. Nevertheless, we can perform MT as follows: Let (G_1, a_1, b_1) and (G_2, a_2, b_2) be two inputs, where G_2 is a permutation of G_1 (that is, G_2 and G_1 are isomorphic), and (a_1, b_1) in G_1 correspond to (a_2, b_2) in G_2. Then an MR can be identified as follows: $|P(G_1, a_1, b_1)| = |P(G_2, a_2, b_2)|$, where $|\cdot|$ refers to the

length of a path. MT can use this MR by running P twice, namely, an execution on the source test case (G_1, a_1, b_1) and an execution on the follow-up test case (G_2, a_2, b_2). If the MR is violated (i.e., $|P(G_1, a_1, b_1)| \neq |P(G_2, a_2, b_2)|$), there is a failure, which implies a fault in P. In some situation, follow-up test cases are dependent on the outputs of source test cases. For instance, consider the following MR: $|P(G, a, c)| + |P(G, c, b)| = |P(G, a, b)|$, where (G, a, b) is a source test case, and c is a node appearing in the output of P, namely the shortest path from a to b in graph G. In this case, the follow-up test cases (G, a, c) and (G, c, b) are dependent on the output of the source test case (G, a, b).

2.2 Category Partition Method (CPM)

CPM is a widely-used specification-based testing technique [51]. It helps software testers create test cases by refining the functional specification of a program into test specifications. The method consists of the following basic steps.

Step 1. Decompose the functional specification into functional units that can be tested independently, and 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*.

Step 2. Partition each category into *choices*, each of which represents a different type/range of values that are possible for that category.

Step 3. 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 a specific test specification language.

Step 4. Use a generator to produce *complete test frames* from the test specification. Each generated complete test frame is a set of choices. Then, create a test case by allocating a concrete value to each choice in a complete test frame.

Consider the **Aviation Consignment Management System** (**ACMS**), which aims at helping 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 (30 kg); otherwise, it is 20 kg. Each aircraft offers three cabin classes to choose from (economy, business, and first) and no fees for infant, with passengers in different classes having different allowances. Note that the airfare for redemption flight is zero. When CPM is used, categories of ACMS are identified, including *air class*, *region*, *is student*, *weight of luggage*, and *airfare*. For each category, their choices are further identified. Table 1 shows the identified categories and their associated choices for ACMS. Next, constraints are identified for each pair of choices, and a test specification can be written using a specific test specification language (refer to TSL [51] for more details). Based on the test specification, complete test frames can be produced. For ACMS, totally 40 complete test frames are generated. For example, a complete test frame is $\{air\ class_{first}, region_{international}, is\ student_{yes}, weight\ of\ luggage_{<=threshold}, airfare_{>0}\}$, where *threshold* denotes the maximum weight of baggage that can be checked in free of charge. Finally, test cases are generated by allocating concrete value to each choice of complete test frames. For the complete test frame mentioned above, a test case can be {*air class* = first, *region* = international, *is student* = yes, *weight of luggage* = 10 kg, *airfare* = 1000CNY}.

The concepts of category, choice, and complete test frame have been popularly used in many testing techniques. In our FDMT approach, we make use of categories and choices to measure the distance between source and follow-up test cases, and select the MR that helps achieve the largest distance. The detailed algorithm, namely the **diversity-oriented MR** (**DOMR**) selection, will be presented in Section 3.3.

Table 1. Categories and Choices for ACMS

Categories	Associated choices
air class	$air\ class_{first}$, $air\ class_{business}$, $air\ class_{economy}$, $air\ class_{infant}$
region	$region_{international}$, $region_{domestic}$
is student	$is\ student_{yes}$, $is\ student_{no}$
weight of luggage	$weight\ of\ luggage_{<=threshold}$, $weight\ of\ luggage_{>threshold}$
airfare	$airfare_{=0}$, $airfare_{>0}$

2.3 Adaptive Partition Testing (APT)

In addition to MR selection, the basic concepts of CPM are also applied in FDMT's source test case selection, which is mainly based on a recently proposed technique, namely APT [62]. APT selects test cases from partitions that have higher fault-detection potentials. In this study, CPM is used for partition testing. It divides the input domain into partitions through the notion of complete test frames which are combinations of choices. In other words, partitions and complete test frames are equivalent concepts in the context of CPM. Since the term "partitions" is more general than the term "test frames" which is specific for CPM, we use "partitions" rather than "test frames" in the context of partition testing. The details of APT are elaborated in the following.

Software cybernetics [11, 13, 69] aims at exploring the interplay between software engineering and control theory. Inspired by the basic idea of software cybernetics, Sun et al. [62] proposed APT, where test cases are randomly selected from some partition c_i (or complete test frame in the context of CPM), whose probability p_i of being selected (also known as the selection probability) is adaptively adjusted along the testing process. APT maintains a test profile to determine the probability of each partition to be selected, as defined in the following:

Definition 1. **Test profile (TP)** is a set of partitions and corresponding selection probabilities, denoted by $TP = \{< c_1, p_1 >, \ldots, < c_i, p_i >, \ldots, < c_m, p_m >\}$, where c_i refers to a partition, p_i represents the selection probability of c_i, and m is a positive integer denoting the number of partitions.

In the dynamic adjustment of test profile, APT attempts to increase the selection probabilities of the partitions that have higher fault-detection potentials: If a fault is detected by some test cases from partition c_i, then c_i is considered to have a higher fault-detection potential and its p_i should be increased; Otherwise, p_i is decreased.

Two algorithms were particularly developed for APT, namely the **Markov-chain based APT** (**MAPT**) and the **reward-punishment based APT** (**RAPT**).

2.3.1 MAPT. According to the concept of Markov chain, given two states i and j, the probability of transitioning from i to j is represented by $p_{i,j} = Pr\{j|i\}$. For $1 \leq i, j \leq m$ (where m refers to the total number of partitions), we can then construct a Markov matrix $\mathcal{P}$ as follows:

$$\begin{Bmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,m} \\ p_{2,1} & p_{2,2} & \cdots & p_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ p_{m,1} & p_{m,2} & \cdots & p_{m,m} \end{Bmatrix}.$$

In the concept of Markov chain, the transition from the current state to the next state is determined by the transition probability, which is affected by the behavior in the current state. Correspondingly, one key issue of APT is to determine the partition for selecting the next source test case based on the behavior of the test cases from the partition, e.g., fail/pass of the test result. Therefore, when applying the Markov chain into partition testing, we establish an intuitive mapping between

the state of the former to the partition of the latter, where the transition probability is dynamically adjusted based on the testing results (i.e., whether the current test case passes or fails). If a partition c_i is selected for generating a test case, the probability of selecting c_j for generating the next test case will be $p_{i,j}$. MAPT will adaptively adjust the value of each $p_{i,j}$ according to the testing result. Interested readers can refer to the original work [62] for detailed MAPT algorithm.

2.3.2 RAPT. Based on the reward and punishment mechanism, RAPT attempts to quickly select the fault-detecting test cases. Two parameters Rew_i and Pun_i are used in RAPT to determine to what extent a partition c_i can be rewarded and punished, respectively. If a test case in c_i reveals a fault, Rew_i will be incremented by 1 and Pun_i will become 0, and test cases will be repeatedly selected from c_i until a non-fault-revealing test case is selected from c_i. If a test case selected from c_i does not reveal a fault, Rew_i will become 0 and Pun_i will be incremented by 1. If Pun_i reaches a preset bound value Bou_i, c_i will be regarded to have a very low fault-detection rate, and its corresponding p_i (probability to be selected) will become 0. The algorithm of RAPT can be found in the original study [62].

Note that both MAPT and RAPT algorithms assume the presence of a test oracle, so their applicability will be hindered if the oracle problem is met; in other words, MAPT and RAPT cannot be directly used for selecting source test cases in MT. As shown in the following section, we propose two new algorithms, namely MAPT* and RAPT*, which can be adopted in the combination with MT and thus effectively address the oracle problem of APT. The difference between MAPT/RAPT and our proposed MAPT*/RAPT* is further elaborated in Section 3.

3 FEEDBACK-DIRECTED METAMORPHIC TESTING

In this section, we first describe the framework for implementing the FDMT, then give concrete algorithms for selecting source test cases and MRs, and finally illustrate FDMT via an example.

3.1 Framework

Leveraging the principles of software cybernetics and the features of MT, we propose FDMT framework, as illustrated in Figure 1. The starting point of FDMT assumes that there already exist some MRs, which could have been either identified in an arbitrary and ad hoc way or constructed using some systematic approaches such as METRIC [22] and data mutation [65]. Another assumption is the availability of test partitions (such as categories/choices in CPM or equivalent classes in equivalent partitioning), from which a test suite is constructed. In other words, the proposed FDMT framework is focused on the feedback-directed selection of appropriate source test cases and MRs. There is a feedback loop in the FDMT framework, which consists of basic components of MT, SUT, the database for storing historical test data, and the controller with concrete FDMT strategies. Particularly, in the controller, the historical testing data are leveraged to guide the selections of source test cases and MRs. The historical data can also be used to improve the underlying testing strategies. The improvement may cause the partition with higher/lower fault-detection capability to have higher/lower selection probability. Interactions between FDMT components are also depicted in the framework. Details of the individual components in the framework are discussed as follows.

The top box of the FDMT framework (Figure 1) is actually the traditional MT, which is composed of the following three components.

— *Follow-Up Test Case Generation.* The follow-up test case is constructed by transforming the source test case based on the selected MR.
— *Test Execution.* This component receives the source and follow-up test cases, and executes them on SUT.

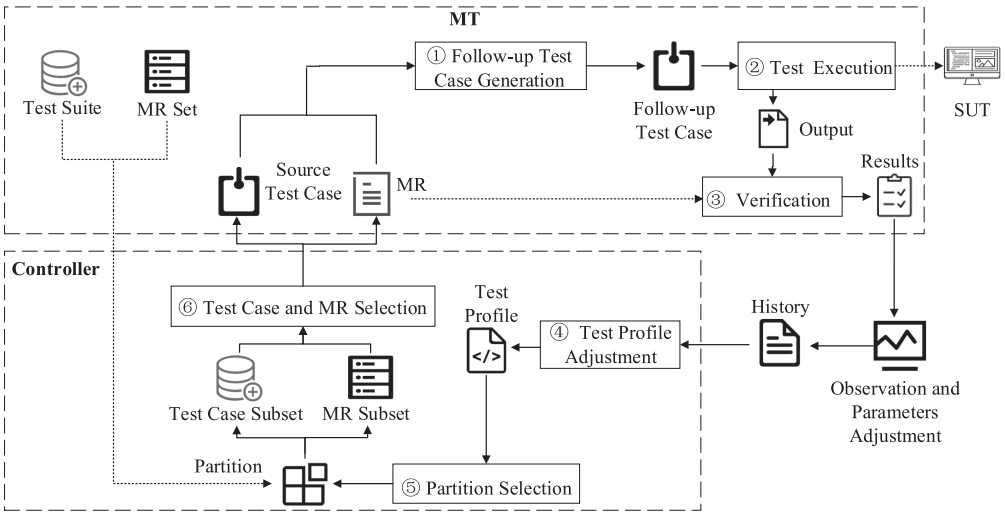


Fig. 1. The framework of FDMT.

— *Verification.* The outputs of source and follow-up test cases are verified against the corresponding MR. MT checks whether the relevant MR holds among multiple executions. If the MR does not hold, then a fault is detected.

At the beginning, FDMT randomly selects a test case and a corresponding MR for execution. The testing results will be output to the component *Observation and Parameters Adjustment*, which is responsible for providing information to the controller. After executing each *metamorphic test group* (which refers to one group of source and follow-up test cases) for an MR, the pass or fail status (represented by whether the MR is satisfied or violated, respectively) is collected, and the information is used to adjust the test profile and update the values of parameters (probability adjusting factors) for the strategies deployed in the controller.

The controller (the bottom box in Figure 1) is responsible for guiding the selections of source test case and MRs. In this study, the source test case selection is based on APT [62]. After constructing partitions (also known as defining complete test frames in CPM), testers need to initialize the test profile. A simple initialization would be a uniform distribution $p_1 = p_2 = \cdots = p_m$, where $m > 0$ denotes the number of partitions, and p_i $(i = 1, 2, \ldots, m)$ represents the probability of selecting the i^{th} partition. Note that some MRs may incur specific conditions for what types of source test cases can be used. Therefore, testers are required to determine a subset R_i $(i = 1, 2, \ldots, m)$ of MRs for each partition c_i so that any test case belonging to c_i can be used to construct follow-up test cases according to any MR in R_i. The controller particularly consists of the following components:

— *Test Profile Adjustment.* The controller updates the test profile using the historical data, so that the partitions with higher fault-detection potentials would be more likely to be selected.
— *Partition Selection.* FDMT randomly selects a partition according to the test profile.
— *Test Case and MR Selection.* Once a partition is chosen, a source test case and MR can be selected based on the strategies proposed in the following Sections 3.2 and 3.3, respectively.

3.2 Source Test Case Selection

Theoretically speaking, any partition-based test case selection strategies can be integrated into the FDMT framework. This can be done through replacing the components "Test Profile Adjustment", "Partition Selection", and "Test Case and MR Selection". Among various partition testing

techniques, APT is a latest one and can deliver higher fault-detection effectiveness with lower test case selection overhead [62] as compared with related partition testing techniques, such as **Random Partition Testing** (**RPT**) and **Dynamic Random Testing** (**DRT**) [42]. Thus, our source test case selection strategies used in FDMT are developed based on two variants of APT, namely MAPT and RAPT. Originally, both MAPT and RAPT make use of the test execution result (i.e., the pass or fail status of a test case) to adjust the test profiles, that is, they rely on the presence of a test oracle to get the information necessary for the adjustment. In the context of MT, the pass or fail of a test case corresponds to the satisfaction or violation of an MR. This fundamental correspondence results in the development of two new testing techniques, namely MAPT* and RAPT*, as presented in the following.

3.2.1 MAPT.* Suppose that the source test case *stc* and the corresponding follow-up test case *ftc* belong to partitions c_s and c_f, respectively. Let $p_{i,j} = Pr\{j|i\}$ denote the probability of transitioning from partition c_i to c_j, where $1 \leq i, j \leq m$ and m refer to the total number of partitions. Furthermore, let $p'_{i,j}$ denote the updated probability of the transition from c_i to c_j.

MAPT* considers the following four scenarios that will appear after executing test cases.

Scenario 1. *A fault is detected, and stc and ftc belong to the same partition ($c_s = c_f$, that is, $s = f$):* We need to update the s^{th} (the same as the f^{th}) row of Markov matrix $\mathcal{P}$. Specifically, we first update the transition probabilities from c_s to c_j ($j \neq s$) according to Formula 1. We then update the transition probability from c_s to c_s according to Formula 2. The hyperparameter γ is the probability adjusting factor, and its value can be changed adaptively based on the values of $p_{i,j}$. The purpose of this operation is to make partitions with higher fault-detection capabilities to gain the priority. In this scenario, MAPT* uses the same method as MAPT to update $\mathcal{P}$, as source and follow-up test cases belong to the same partition.

Scenario 2. *A fault is detected, and stc and ftc do not belong to the same partition ($s \neq f$):* The triggered fault may reside in either the partition containing the source test case, or that containing the follow-up test case, or both. However, it is not trivial to determine which test cases (and correspondingly which partitions) triggered the fault [35]. This article does not investigate this issue, and hence we increase both of the selection probabilities of c_s and c_f. Accordingly, we need to update both the s^{th} and f^{th} rows of Markov matrix $\mathcal{P}$. Specifically, the s^{th} row of $\mathcal{P}$ is updated by using Formulas 1 and 2. Let $s = f$, Formulas 1 and 2 can be used to update the f^{th} row of $\mathcal{P}$. The increase of $p_{s,s}$ and $p_{f,f}$ is related to the size of their own value—the larger the value, the greater the increase; and vice versa.

$$p'_{s,j} = \begin{cases} p_{s,j} - \dfrac{\gamma \times p_{s,s}}{m-1} & \text{if } p_{s,j} > \dfrac{\gamma \times p_{s,s}}{m-1} \\ p_{s,j} & \text{if } p_{s,j} \leq \dfrac{\gamma \times p_{s,s}}{m-1}, \end{cases} \tag{1}$$

$$p'_{s,s} = 1 - \sum_{j=0, j\neq s}^{m} p'_{s,j}, \tag{2}$$

Scenario 3. *No fault has been detected, and stc and ftc belong to the same partition ($s = f$):* We need to update the s^{th} (the same as the f^{th}) row of Markov matrix $\mathcal{P}$. Specifically, we first update the transition probabilities from c_s to c_j ($j \neq s$) by using Formula 3, and then update the transition probability from c_s to c_s by using Formula 4. The hyperparameter τ is the probability adjusting factor, and its value can be changed adaptively based on the values of $p_{i,j}$. The purpose of this operation is to make partitions with lower fault revealing capabilities to give up the priority. MAPT* uses the same method as MAPT to update $\mathcal{P}$, as source and follow-up test cases belong to the same partition.

Scenario 4. *No fault has been detected, and stc and ftc do not belong to the same partition ($s \neq f$):* We need to update both the s^{th} and f^{th} rows of Markov matrix $\mathcal{P}$. We first employ Formulas 3 and 4 to update the transition probabilities from c_s to c_j ($j \neq s$), and transition probability from c_s to c_s, respectively. Let $s = f$, Formulas 3 and 4 are used to update the transition probabilities from c_f to c_j ($j \neq f$), and transition probability from c_f to c_f, respectively. The reduction of $p_{s,s}$ and $p_{f,f}$ is related to the size of their own value—the larger the value, the smaller the reduction; and vice versa.

$$p'_{s,j} = \begin{cases} p_{s,j} + \dfrac{\tau \times p_{s,j}}{m-1} & \text{if } p_{s,s} > \dfrac{\tau \times (1-p_{s,s})}{m-1} \\ p_{s,j} & \text{if } p_{s,s} \leq \dfrac{\tau \times (1-p_{s,s})}{m-1}, \end{cases} \tag{3}$$

$$p'_{s,s} = \begin{cases} p_{s,s} - \dfrac{\tau \times (1-p_{s,s})}{m-1} & \text{if } p_{s,s} > \dfrac{\tau \times (1-p_{s,s})}{m-1} \\ p_{s,s} & \text{if } p_{s,s} \leq \dfrac{\tau \times (1-p_{s,s})}{m-1}, \end{cases} \tag{4}$$

The guidelines of setting the probability adjusting factors γ and τ are described in Section 4.8.

3.2.2 RAPT.* Similar to MAPT*, RAPT* considers four scenarios that may occur after completing a test. Formulas (7), (8), (9), and (12) are the expansion of RAPT* relative to RAPT, which can solve the problem of updating test profile when the source and follow-up test cases are not in the same partition.

Under the situation that *stc* and *ftc* cause an MR violation, RAPT* updates the test profile as follows. Let p_i and p'_i denote the original and updated probability for selecting a partition c_i, respectively. When *stc* and *ftc* belong to the same partition ($c_s = c_f$), we use Formulas 5 and 6 to update the selection probabilities of c_s and c_i ($i \neq s$). The hyperparameter ϵ is the probability adjusting factor, and its value can be changed adaptively based on the values of Rew_i, which is used to determine to what extent a partition c_i can be rewarded. If a test case in c_i reveals a fault, Rew_i will be incremented by 1, and test cases will be repeatedly selected from c_i until a nonfault-revealing test case is selected from c_i. Accordingly, the partitions with stronger fault revealing ability have a larger probability adjusting factor. If a test case selected from c_i does not reveal a fault, Rew_i will become 0.

$$p'_i = \begin{cases} p_i - \dfrac{(1 + lnRew_s) \times \epsilon}{m-1} & \text{if } p_i > \dfrac{(1 + lnRew_s) \times \epsilon}{m-1} \\ 0 & \text{if } p_i \leq \dfrac{(1 + lnRew_s) \times \epsilon}{m-1}, \end{cases} \tag{5}$$

$$p'_s = 1 - \sum_{i=0, i \neq s}^{m} p'_i. \tag{6}$$

When $c_s \neq c_f$, we use Formulas 7, 8, and 9 to update the selection probabilities of c_s, c_f, and c_i ($i \neq s, f$).

$$p'_i = \begin{cases} p_i - \dfrac{(1 + lnRew_s) \times \epsilon}{m-2} & \text{if } p_i > \dfrac{(1 + lnRew_s) \times \epsilon}{m-2} \\ 0 & \text{if } p_i \leq \dfrac{(1 + lnRew_s) \times \epsilon}{m-2}, \end{cases} \tag{7}$$

$$p'_s = p_s + \frac{1 - \sum_{i=0, i \neq s, i \neq f}^{m} p'_i - p_s - p_f}{2}, \tag{8}$$

$$p'_f = p_f + \frac{1 - \sum_{i=0, i \neq s, i \neq f}^{m} p'_i - p_s - p_f}{2}. \tag{9}$$

When the results of *stc* and *ftc* do not violate the MR, RAPT* employs the following procedure to update test profile. If *stc* and *ftc* belong to same partition ($c_s = c_f$), we use Formulas 10 and 11 to update test profile. The hyperparameter δ is the probability adjusting factor. Pun_i is used to determine to what extent a partition c_i can be punished. If a test case selected from c_i does not reveal a fault, Pun_i will be incremented by 1. If Pun_i reaches a preset upper bound value Bou_i, c_i will be regarded to have a very low failure rate, and its corresponding p_i will become 0. However, if a test case in c_i reveals a fault, Pun_i will become 0.

When $c_s \neq c_f$, Formula 10 is used to update the selection probability of c_s. Let $s = f$, Formula 10 can be used to update the selection probability of c_f. The selection probabilities of c_i $(i \neq s, f)$ are updated by Formula 12.

$$p'_s = \begin{cases} p_s - \delta & \text{if } p_s > \delta \text{ and } Pun_s \neq Bou_s \\ 0 & \text{if } p_s \leq \delta \text{ or } Pun_s = Bou_s, \end{cases} \tag{10}$$

$$p'_i = \begin{cases} p_i + \dfrac{\delta}{m-1} & \text{if } p_s > \delta \text{ and } Pun_s \neq Bou_s \\ p_i + \dfrac{p_s}{m-1} & \text{if } p_s \leq \delta \text{ or } Pun_s = Bou_s, \end{cases} \tag{11}$$

$$p'_i = p_i + \frac{(p_s - p'_s) + (p_f - p'_f)}{m-2}. \tag{12}$$

The detailed guidelines of setting the probability adjusting factors ϵ and δ are described in Section 4.8. Note that although MAPT* and RAPT* are proposed as part of the FDMT framework, they can be used as independent testing techniques. Since MAPT* and RAPT* can address the oracle problem through the integration with MT, they significantly boost the applicability of APT. Similar work can be conducted to enhance other software cybernetics based testing techniques that have the oracle problem.

Compared with the original MAPT and RAPT algorithms, MAPT* and RAPT* have much higher applicability, that is, they can be used in more testing scenarios.

(1) The previous MAPT and RAPT algorithms assume the presence of a test oracle; therefore, when the oracle problem is met, MAPT and RAPT are no longer applicable. As a result, MAPT and RAPT cannot be used as candidates for test case selection techniques in MT. However, MAPT* and RAPT* are designed based on the principle of MT, and naturally serve as good candidates for test case selection techniques in MT.

(2) After executing a test case, MAPT and RAPT update the test profile according to the execution result of a test case, that is, when MAPT and RAPT update test profile according to the following two situations: (a) The current test case detects a fault; and (b) The current test case does not detect a fault. However, MAPT* and RAPT* consider the more complicated execution behaviors of MT (in a test process, both the source and follow-up test cases are executed), and the strategies of updating the test profile are designed according to four scenarios discussed above.

3.3 Selection of Metamorphic Relations

As discussed in Section 3.1, in FDMT, once a source test case is selected (more specially, a partition is chosen), a subset of MRs will be available for selection. One simple way is the random MR selection strategy, which we term as RMRS, to be distinguished from RT for test case selection.

RMRS does not use extra information and directly selects an MR randomly from the subset of eligible MRs for a partition. As discussed above and demonstrated by previous studies [40, 59],

the performance of MT is highly dependent on the used MRs, and thus selecting effective MRs is a critical step in MT. Based on the previous observations that good MRs should cause different execution behaviors of source and follow-up test cases [3, 17], we introduce a so-called "*category-partition-based distance metric*" (CP-distance) that measures the distance between the source and follow-up test cases. Similar to that used in ART [6], the CP-distance is able to reflect the diversity in functionalities achieved by source and follow-up test cases. Based on the CP-distance, a *Diversity-Oriented strategy of MR selection* (DOMR) is proposed.

The formal definition of CP-distance is as follows. Suppose for a program, we have identified the set of categories $A = \{A_1, A_2, \ldots, A_g\}$, where g refers to the total number of categories. For each A_i, its choices are denotes by $P^{A_i} = \{p_1^{A_i}, p_2^{A_i}, \ldots, p_h^{A_i}\}$, where h denotes the number of choices for A_i. For an input x, its corresponding set of categories is denoted by $A(x) = \{A_1^x, A_2^x, \ldots, A_q^x\}$, where q ($q \leq g$) refers to the number of categories associated with x. Since categories are distinct and their choices are disjoint, input x actually refers to 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 of choices that contains elements in either $P(x)$ or $P(y)$ but not both, that is,

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

where "\" is the set difference operator. We then define

$$DA(x, y) = \left\{A_m | A_i \text{ if } p_j^{A_i} \in DP(x, y)\right\},$$

where $DA(x, y)$ is effectively the set of categories in which inputs x and y have different choices. The CP-distance 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 x and y have different choices.

Obviously, a greater value of CP-distance represents the more dissimilar executions between two test cases. After selecting a source test case stc_i and obtaining an *MR candidate set* $\mathcal{R} = \{r_1^{stc_i}, r_2^{stc_i}, \ldots, r_k^{stc_i}\}$ (k is the number of MRs whose source test case could be stc_i), DOMR constructs a set of candidate follow-up test cases $FC = \{ftc_{r_1^{stc_i}}, ftc_{r_2^{stc_i}}, \ldots, ftc_{r_k^{stc_i}}\}$ according to every MR in $\mathcal{R}$. Then, the distance $CP_{stc_i, ftc_{r_h^{stc_i}}}$ ($h \in \{1, 2, \ldots, k\}$) 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 if and only if it has the longest CP-distance among all candidate MRs, that is,

$$CP_{stc_i, ftc_{r_h^{stc_i}}} = max\{CP_{stc_i, ftc_{r_1^{stc_i}}}, CP_{stc_i, ftc_{r_2^{stc_i}}}, \ldots, CP_{stc_i, ftc_{r_k^{stc_i}}}\}.$$

Note that under the situation where there is more than one MR having the longest CP-distance, one of them is randomly selected.

When FDMT is practiced, it is assumed that MRs and partitions (including categories and choices) are available. Manual cost of FDMT is mainly related to the settings of parameters and test profiles, since the selection of source test cases and MRs are controlled by the strategy-based algorithms. Furthermore, CPM is widely adopted in practice, which provides effective partitioning schemes (in terms of the category-choice framework) required by FDMT and also facilitates the MR identification using METRIC. Considering these together, we believe that FDMT can be deemed promising to be widely applied in practice.

3.4 Illustration

We use ACMS as an example to illustrate the proposed FDMT. The specification of ACMS is described in Section 2.2. Given the specification, we employed CPM (described in Section 2.2) to partition the input domain. Based on identified categories, choices, and constraints among them, we derived

Table 2. Partitions of ACMS

Partitions	Test Frames	Example Test Cases
c_1	{$region_{domestic}$, *is student*$_{yes}$, *air class**, *weight of luggage**, *airfare**}	{*region* = domestic, *is student* = yes, *air class* = economy, *weight of luggage* = 100kg, *airfare* = 1,000CNY}
c_2	{$region_{domestic}$, *is student*$_{no}$, *air class**, *weight of luggage**, *airfare**}	{*region* = domestic, *is student* = no, *air class* = first, *weight of luggage* = 0kg, *airfare* = 1,500CNY}
c_3	{$region_{international}$, *is student*$_{yes}$, *air class**, *weight of luggage**, *airfare**}	{*region* = international, *is student* = yes, *air class* = business, *weight of luggage* = 0kg, *airfare* = 2,000CNY}
c_4	{$region_{international}$, *is student*$_{no}$, *air class**, *weight of luggage**, *airfare**}	{*region* = international, *is student* = no, *air class* = economy, *weight of luggage* = 50kg *airfare* = 1,800CNY}

a set of complete test frames, each of which corresponds to a partition (detailed partitions to be discussed in Section 4.6).

For ease of illustration, we only considered the *region* category and *is student* category to determine the partitions. Table 2 shows the resulting partitions for ACMS, where *air class**, *weight of luggage**, and *airfare** indicate no classification on the categories of air class, weight of luggage, and airfare, respectively.

Let us first look at how FDMT works when MAPT* and DOMR are used to select source test cases and MRs, respectively. As detailed in Section 4.8, we set $\gamma = \tau = 0.1$ for MAPT*. Assume that the initial test profile is $TP = \{< c_1, 0.25 >, < c_2, 0.25 >, < c_3, 0.25 >, < c_4, 0.25 >\}$, and the initial Markov matrix $\mathcal{P}$ is as follows:

$$\begin{Bmatrix} 0.25 & 0.25 & 0.25 & 0.25 \\ 0.25 & 0.25 & 0.25 & 0.25 \\ 0.25 & 0.25 & 0.25 & 0.25 \\ 0.25 & 0.25 & 0.25 & 0.25 \end{Bmatrix}.$$

Suppose that at the first step of testing, the partition c_1 is selected according to TP, and then tc_1 = {*region* = domestic, *is student* = yes, *air class* = economy, *weight of luggage* = 100 kg, *airfare* = 1,000CNY} is randomly selected as the source test case. Further assume that there are two MRs related to partition c_1 (refer to Section 4.7 for more information about MRs for the subject programs in this study), which are described below:

— $MR^1_{c_1}$**:** Suppose passenger A is a student, who purchases economy class ticket on a domestic flight, and the weight of his/her carry-on luggage exceeds the weight of free luggage; passenger B is not a student, who purchases economy class ticket on a domestic flight, and the weight of his/her carry-on luggage does not exceed the weight of free luggage. Then, the luggage cost of passenger A is greater than that of B.

— $MR^2_{c_1}$**:** Suppose passenger A is a student, who purchases economy class ticket on a domestic flight, and the weight of his/her carry-on luggage exceeds the weight of free luggage; passenger B is not a student, who purchases first class ticket on a domestic flight, and the weight of his/her carry-on luggage does not exceed the weight of free luggage. Then, the luggage cost of passenger A is greater than that of B.

Suppose that based on the source test case tc_1, two follow-up test cases tc^1_1 and tc^2_1 can be obtained using $MR^1_{c_1}$ and $MR^2_{c_1}$, respectively: tc^1_1 = {*region* = domestic, *is student* = no,

air class = economy, *weight of luggage* = 0 kg, *airfare* = 1000CNY} and tc_1^2 = {*region* = domestic, *is student* = no, *air class* = first, *weight of luggage* = 0 kg, *airfare* = 1500CNY}. Based on the generated follow-up test cases, we can get their corresponding complete test frames (obviously, they both belong to c_2), and calculate CP-distance between the source test case and the follow-up test cases. In this example, $|DA(tc_1, tc_1^1)| = 2$, and $|DA(tc_1, tc_1^2)| = 3$. Since $|DA(tc_1, tc_1^2)| > |DA(tc_1, tc_1^1)|$, we select $MR_{c_1}^2$. Correspondingly, we use tc_1 and tc_1^2 to execute ACMS, and determine whether the test results violates $MR_{c_1}^2$. If the MR is violated, a fault is said to be detected. Then, according to the process of Scenario 2 in Section 3.2.1, $p_{1,j}$ and $p_{2,j}$ $(j = 1, 2, 3, 4)$ are updated using Formulas (1) and (2). The updated $\mathcal{P}$ is as follows:

$$\begin{Bmatrix} 0.2749 & 0.2417 & 0.2417 & 0.2417 \\ 0.2417 & 0.2749 & 0.2417 & 0.2417 \\ 0.2500 & 0.2500 & 0.2500 & 0.2500 \\ 0.2500 & 0.2500 & 0.2500 & 0.2500 \end{Bmatrix}.$$

Based on the updated $\mathcal{P}$, the next partition can be selected according to $p_{i,j}$. The above steps are repeated until the termination condition is satisfied.

Let us further illustrate the process of FDMT when applying RAPT* and DOMR to select source test cases and MRs, respectively. Parameters for RAPT* are set as $\epsilon = \delta = 0.1$ (details to be given in Section 4.8). Assume that the initial test profile is $TP = \{< c_1, 0.25 >, < c_2, 0.25 >, < c_3, 0.25 >, < c_4, 0.25 >\}$, and $Pun_i = Rew_i = 0$, where $i = 1, 2, 3, 4$. Further suppose that in the first round of testing, the partition c_1 is selected according to TP, tc_1 = {*region* = domestic, *is student* = yes, *air class* = economy, *weight of luggage* = 100 kg, *airfare* = 1000CNY} is then randomly selected as the source test case. Given the above two MRs, $MR_{c_1}^1$ and $MR_{c_1}^2$, DOMR is employed to select $MR_{c_1}^2$ as its corresponding follow-up test case tc_1^2 has larger distance to tc_1 than tc_1^1 for $MR_{c_1}^1$. If the execution results of tc_1 and tc_1^2 violate $MR_{c_1}^2$ (that is, a fault is detected), we set $Rew_1 = 1$ and $Rew_2 = 1$, and follow Scenario 2 to update $TP = \{< c_1, 0.35 >, < c_2, 0.35 >, < c_3, 0.15 >, < c_4, 0.15 >\}$ based on Formulas (7), (8), and (9).

4 EMPIRICAL STUDY

We conducted a series of empirical studies to evaluate the performance of FDMT.[1]

4.1 Research Questions

FDMT controls the execution process of MT with the main goal of improving the fault-detection efficiency of MT. Accordingly, our experiments were designed to measure FDMT from two critical perspectives of cost-effectiveness, namely the fault-detection effectiveness and the overall efficiency. A testing technique can be considered as highly cost-effective if it is effective in fault detection as well as efficient in terms of consumed resources and time. Correspondingly, the efficiency of MT is reflected and thus should be evaluated in two aspects: (1) whether FDMT can detect faults using fewer test cases than the traditional MT; and (2) whether FDMT can detect the same number of faults using a shorter time than the traditional MT. Specifically, we attempt to answer the following two research questions:

RQ1 Can the proposed FDMT approach use fewer test cases than MT to detect faults? Test cases are the major resources for testing. Many efforts will be made on generating test cases, and it may be time-consuming to execute them in many situations. In SWEBOK [8, Section 4.2.4], it is recommended to use the number of test cases required for detecting the

[1]The experimental materials (including the implementations, categories and choices, complete test frames, MRs, and used faults of subject programs) have been made available at https://fdmt2021.github.io/FDMT/.

Table 3. Six Programs as Experimental Objects

Source	Program Name	Implementation Language	LOC	Number of Faults	Number of MRs	Number of Partitions
Laboratory	ACMS	Java	128	3	142	40
	CUBS	Java	107	3	184	32
	ERS	Java	117	1	1,130	70
	MOS	Java	135	1	3,512	180
GNU	grep	C	10,068	8	12	550
Alibaba	FastJson	Java	125,192–149,544*	4	17	84

*Different faulty versions of FastJson have varying LOCs.

first fault as one major means to evaluate the testing effectiveness. Intuitively speaking, the fewer test cases a testing technique can use to detect software fault, the more effective the technique is.

In addition to comparing FDMT with MT, we are also interested in the impacts of different source test cases and MR selection strategies on FDMT's performance. Thus, we further investigate the following two sub-research questions:

— [**RQ1.1**] To what extent can MAPT* and RAPT* improve the effectiveness of MT?
— [**RQ1.2**] Is DOMR more effective than RMRS in the FDMT framework?

RQ2 Can the proposed FDMT approach take shorter testing time than MT to detect the same number of faults?

The overall testing time is a direct measurement for a testing technique's efficiency. A good testing method should be able to achieve its testing goal (e.g., detecting a certain number of faults) as quickly as possible.

Similar to RQ1, we also have two sub-research questions under RQ2:

— [**RQ2.1**] To what extent can MAPT* and RAPT* help reduce the testing time of MT?
— [**RQ2.2**] Can DOMR bring higher efficiency than RMRS?

Note that the number of test cases required for fault detection (for RQ1) is the fundamental measurement for the effectiveness of a testing technique; whereas the testing time (for RQ2) naturally reflects the overall testing efficiency, which is collectively affected by a wide variety of factors, such as the number of used test cases, the program execution time, and the operating environment.

4.2 Subject Programs

In order to evaluate the cost-effectiveness of the proposed method on different scales, various implementation languages, and a variety of fields, we chose three sets of subject programs: (1) four laboratory programs, namely ACMS, CUBS, ERS, and MOS, which were developed by our team according to various real-life business specifications; (2) the regular expression processor component of a large UNIX/LINUX utility application, namely GNU grep; and (3) a Java library developed by Alibaba, namely FastJson,[2] which can be used to convert Java Objects into their JSON representation, and convert a JSON string to an equivalent Java object. Table 3 summarizes the basic information of each subject program, including the program name, the implementation language, the lines of code, the number of faults used in our experiments, the number of identified MRs, and the number of partitions.

Note that the four laboratory programs have been used in previous studies related to MT and APT. For example, Sun et al. [64] used these programs to evaluate the effectiveness and

[2]This project is available at https://github.com/alibaba/fastjson.

efficiency of METRIC+, an advanced MR identification method developed based on METRIC [22]. Sun et al. [63] also used three of these programs (ACMS, CUBS, and ERS) to evaluate the fault-detecting efficiency of DRT in web services. grep has also been widely used in various studies for software testing [7, 68].

The four laboratory programs were developed in Java programming language. The basic function of each program is as follows:

(1) Aviation Consignment Management System (ACMS) aims to help airline companies check the allowance (weight) of free baggage, and the cost of additional baggage. The specification of ACMS is described in Section 2.2.
(2) 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.
(3) Expense Reimbursement System (ERS) assists the sales supervisor of a company in handling the following tasks: (i) calculating the cost of employees who use the cars based on their positions and the number of miles actually traveled; (ii) accepting the reimbursement requests that include airfare, hotel accommodation, food, and cell-phone expenses of employees.
(4) Meal Ordering System (MOS) helps catering providers to determine the quantity for every type of meal and other special requests (if any) that need to be prepared and loaded onto the aircraft. For each flight, MOS generates a **Meal Schedule Report** (**MSR**), containing the number of various types of meals (first class, business class, economy class, vegetarian, child, crew member, and pilot) and the number of flower bundles.

We used version 2.5.1a of the grep program [28]. 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 widely used in UNIX system, providing an opportunity to demonstrate the real-world relevance of our approach.
(2) grep program and its input format are of great complexity, but still manageable as a target for automated test case selection.
(3) grep program has been widely used in the research of testing [6, 26], which means that we can easily access existing faulty versions, test cases, and MRs.

The inputs of grep were categorized into three components: (*i*) options, which consist of a list of commands to modify the searching process; (*ii*) pattern, which is the regular expression to be searched for; and (*iii*) files, which refer to the input files to be searched. The scope of functionality of this program is very large, which makes it impractical to construct infrastructure for thoroughly testing all its functionalities. Therefore, we restricted our focus to the regular expression analyzer of grep.

FastJson is a Java library that can be used to convert Java Objects into their JSON representation (known as serialization). It can also be used to convert a JSON string to an equivalent Java object (known as deserialization). We chose FastJson in our study for several reasons:

(1) FastJson is more complex than other programs and has been widely used in real-word projects, which help justify the practicality of our approach.
(2) FastJson is an open source project; thus we can easily obtain its faulty versions.

Due to the large scale of FastJson, we restricted our focus to the deserialization part. Two version sets of FastJson are available, namely FastJson 1.1.* and FastJson 1.2.*. Since there

are no faults related to deserialization reported for FastJson 1.1.*, we focused on FastJson 1.2.1 -- FastJson 1.2.48 to collect its faulty versions, as detailed in Section 4.3.

These three sets of programs present complementary strengths and weaknesses as experiment objects. The laboratory programs implement simple functions and their interfaces are easy to understand. The test engineers can easily select source test cases, and identify MRs for testing them. However, these programs are small and there are a limited number of faulty versions available for each program. The grep program is much larger for which a reasonable amount of mutation faults could be generated. FastJson is also a very large program, and its associated faults can be obtained from GitHub. In the next section, we present the detail of how we created/selected the faults for each object program.

4.3 Faults

We employed the tool MuJava [45] to conduct mutation analysis for four laboratory programs, generating a total of 792 mutants (210, 187, 180, and 215 mutants for ACMS, CUBS, ERS, and MOS, respectively). Each mutant was obtained by changing the original program's syntax (using one of the applicable mutation operators provided by MuJava). The "equivalent" mutants (those always showing the same execution behaviors as the base program) were first filtered by the following two steps: (1) Randomly generate 10,000 test cases to execute each mutant of all laboratory programs (the total time overhead was 49.812s); (2) The members of research team (including two master students and two doctoral students) independently analyzed each still-alive mutant. If they had different analysis results for the same mutant, a discussion would be held to reach a consensus on whether a mutant is equivalent to the base program. In total, we identified 22 equivalent mutants, including 3 of ACMS and 19 of CUBS, respectively. We also found that there were lots of mutants that were very easy to kill. Such mutants were filtered through the following procedure. Complete test frames were constructed based on the identified categories and choices (the process of which is explained in Section 4.5). Then, test cases were randomly selected from each complete test frame. If a mutant could be killed by more than 10% of the random test cases, we considered them as easy-to-kill and removed them from our study (the time overhead of this process was 12.192s). Finally, we filtered 204, 165, 179, and 214 mutants, and obtained three, three, one, and one faults for ACMS, CUBS, ERS, and MOST, respectively (refer to the fifth column of Table 3).

Barus et al. [6] have provided 20 faulty versions for grep, including one real fault and 19 artificial faults. Also provided was a test suite that contains 101,193 test cases. We removed the faulty versions that could be killed by more than 10% of the original test cases, and finally obtained eight faults.

All information about FastJson's faults is available at https://github.com/alibaba/fastjson. We first collected all the issues related to deserialization of FastJson. For those issues that are recognized as faults by the developers, we then examined their descriptions to determine whether a fault is related to the deserialization using either reasoning by semantics (i.e., the description concerns deserialization) or reasoning by cases (with deserialization examples), then removed the faults that caused the runtime errors, and finally got four real hard-to-kill faults, which appear in FastJson_1.2.31, FastJson_1.2.36, FastJson_1.2.40, and FastJson_1.2.48. Note that in each round of testing on every subject program, we removed the fault once it was detected by a testing technique.

4.4 Variables

4.4.1 Independent Variables. The independent variable in our experiments relates to the different techniques under investigation. Table 4 summarizes the six techniques we selected for our study. By nature, we chose the proposed FDMT approach, which has four variants, namely

Table 4. Independent Variable—Techniques under Study

Technique	Source Test Case Selection Strategy	MR Selection Strategy
MR-FDMT	MAPT*	RMRS
MD-FDMT	MAPT*	DOMR
RR-FDMT	RAPT*	RMRS
RD-FDMT	RAPT*	DOMR
MT-ART	ARTSum	RMRS
MT-RT	RT	RMRS

MR-FDMT, MD-FDMT, RR-FDMT, RD-FDMT, constructed through the combination of two source test case selection strategies (MAPT* and RAPT*) and two MR selection strategies (RMRS and DOMR).

RT is a fundamental technique for source test case generation. In this study, we employed the strategy of RT plus RMRS (denoted by MT-RT), as one baseline for the comparison. There also exist other techniques to generate source test cases. For example, Barus et al. [7] investigated the impact of source test cases on the effectiveness of MT. Their experimental results show that ART outperforms RT on enhancing the effectiveness of MT. In our experiments, we employed one of the latest linear-order ART algorithms—ARTSum [6] (which was designed for programs with non-numeric inputs) to generate the source test cases for MT. In a word, we used the strategy of ARTsum plus RMRS, denoted by MT-ART, as another baseline technique.

4.4.2 Dependent Variables. The choice of a metric for comparing the performance of testing techniques is a non-trivial task.

In this study, we consider two kinds of testing scenarios: (1) when a fault is detected for the first time, that is, no prior fault detection; and (2) when a fault is detected after some prior fault detection(s). For the former, the F-measure[3] is used, while for the latter, F2-measure is used. F-measure refers to the expected number of test cases required to detect the first fault. F2-measure is defined as the expected number of additional test cases required to detect the second fault after revealing the first fault. More specifically, the F-measure reflects the performance of our approach without the prior fault detection. Furthermore, the F2-measure can be generalized to evaluate the performance after some fault(s) have already been detected (that is, the capability of detecting the $(i+1)$th fault in the context of the detection of ith faults). In other words, the measure on the highest number of faults can be reflected by the generalization of F2-measure. In addition, the F2-measure is more appropriate than other metrics for the highest number of faults, such as the mutation score of the given MRs and test suites, for the evaluation of "dynamic" testing techniques, such as APT [62] and FDMT.

Due to the randomness in our techniques, we determined the values of F-measure and F2-measure through the following procedure. We first define *F-count* as the number of test cases needed to detect the first fault in a specific test run, and *F2-count* as the number of additional test cases needed to detect the second fault after the detection of the first fault.

[3]Note that the F-measure used in this study is different from F-score used in the fields of statistical analysis and machine learning.

The F-measure/F2-measure can be calculated as the mean value of $F\text{-}count/F2\text{-}count$ over multiple test runs:

$$\text{F-measure} = \overline{F\text{-}count}, \tag{13}$$

and

$$\text{F2-measure} = \overline{F2\text{-}count}. \tag{14}$$

The F-measure is particularly appropriate for measuring the fault-detection effectiveness of FDMT, as it intuitively indicates the speed of fault detection. The value of F2-measure particularly reflects FDMT's capability of detecting multiple faults with the aid of historical testing information. Intuitively speaking, the smaller F-measure/F2-measure is, the better a testing technique is.

An obvious metric for RQ2 is the time required to detect faults. Corresponding to the F-measure and F2-measure, we used the metrics F-time and F2-time, which refer to the expected time for detecting the first and second faults, respectively. This study focuses on improving the cost-effectiveness of MT from the perspective of dynamically selecting MRs and test cases. In order to guarantee a fair comparison between FDMT and traditional MT, we need to ensure they have the same cost for MR generation. Apparently, different MR generation techniques incur different costs, and there are many factors determining which generation technique should be adopted for a particular system. Since the METRIC approach is applicable in various systems (due to the generality of CPM), we used it to generate MRs for this study. Accordingly, the starting point of FDMT is that MRs and that categories and choices (main artifacts of CPM) are already available. Note that the efforts for MR generation are inevitable for any MT approach; correspondingly, the relevant time cost is applicable to both FDMT and traditional MT. In other words, when we compare the testing time for FDMT and traditional MT to evaluate the performance improvement, the time required for such a common procedure of MR generation (including the effort of performing CPM and METRIC) would be removed and thus not necessary to be taken into account. F-time/F2-time reflects the overall time taken for testing, which is mainly composed of the following three components:

(1) `Test frame selection time` that refers to how long it takes to select the complete test frame (or partition) according to a test profile.
(2) `Test case creation time` that consists of the time for creating source test case from the selected complete test frame as well as constructing follow-up test case based on the selected MR.
(3) `Test case execution time` that represents the time required for executing both source and follow-up test cases.

Obviously, a smaller value of F-time/F2-time indicates a higher efficiency.

4.5 Categories and Choices

The categories and choices for the four laboratory programs and `FastJson` were identified by the authors, while the categories and choices for `grep` were provided in previous studies [6]. The identification of appropriate categories and choices was basically a subjective process, so different individuals (members in our laboratory) have been involved to cross-check the identified categories and choices to guarantee the generality.

For `ACMS`, `CUMS`, `ERS`, and `MOS`, we have identified five, four, five and seven categories, each of which had two to five choices.

For `grep`, Barus et al. [6] have made use of the existing testing information available in the software infrastructure repository [26] to identify categories and choices. Furthermore, they divided the categories into two groups, namely the independent and the dependent categories. The

independent categories refer to those that can form a valid test case on their own, whereas the dependent categories need the presence of other categories to form valid test cases. There were a total of 14 categories for grep, and the number of choices for each category ranges from two to twelve. Detailed categories and choices for grep can be found in previous studies [6].

To define the categories and choices of FastJson, the specification and user documentation were first examined. However, they were either not discovered or did not provide significant detail to construct appropriate categories and choices. Therefore, we also made use of the existing best practices, the existing set of test cases, and the source code recorded in the Alibaba repository.[4] As mentioned above, we restricted our experimentation to testing the deserialization of FastJson. As a consequence, the categories and choices were only related to this portion of FastJson. Deserialization is the process of translating a stored format (for example, a Json file or memory buffer) into an object state, which contains the values of all the variables in a program. Therefore, we considered the base member variables and their combinations. Totally, we identified eleven categories for FastJson, each of which contained two to 28 choices.

4.6 Partitioning and Initial Test Profiles

Our test case selection strategies (MAPT* and RAPT*) were established on the concept of partitioning (or complete test frames in CPM). The following gives the detail how we partitioned the input domain for each subject program.

For ACMS, CUBS, ERS, and MOS, we considered all categories and then partitioned their input domain according to all possible combinations of choices (that is, the complete test frames), resulting in 40, 32, 70, and 180 partitions, respectively. The other two programs, grep and FastJson, have two types of categories, namely the independent categories and dependent categories. The former refers to the categories that exist in any complete test frame no matter whether other categories exist or not; while the existence of the latter is dependent on whether some independent categories and associated choices appear in a complete test frame. In our experiments, we partitioned the input domains for grep and FastJson only based on the independent categories. Accordingly, we constructed 550 and 84 partitions for grep and FastJson, respectively.

Based on the partitioning, the four FDMT techniques (more specifically, MAPT* and RAPT*) make use of feedback information to adjust the test profile. Then, the updated test profile will guide the selection of the next source test case and MR. Before implementing these techniques, a concrete test profile should be initialized for each program. In the original study of APT [62], we have compared the two types of initial test profiles, namely the uniform distribution and the proportional profile, in terms of their impacts on the performance metrics (F-measure/F2-measure and F-time/F2-time). The comparison results showed that there was no significant difference between them. In our experiment, we used a uniform probability distribution for the initial test profile. In other words, the initial test profile for each program is $\{< c_1, \frac{1}{m} >, < c_2, \frac{1}{m} >, \ldots, < c_m, \frac{1}{m} >\}$, where c_i is the ith partition and the probability of selecting c_i in the initial test profile is $p_i = \frac{1}{m}$ ($1 \leq i \leq m$ and m refers to the number of partitions).

The number of partitions for each object program is given in the seventh column of Table 3.

4.7 Identification of MRs for Subject Programs

In our study, for ACMS, CUBS, ERS, and MOS, we made use of the METRIC technique [22] to identify MRs, which is based on the concepts of categories and choices [51]. In METRIC, only two distinct complete test frames are considered by the tester to construct an MR. In general, METRIC takes the following steps to identify MRs:

[4]The best practices and test suite are available at: https://github.com/alibaba/fastjson/wiki.

Table 5. Theoretical Values of δ

Program	θ_M	θ_Δ	δ
ACMS	1.16E-2	3.87E-3	3.91E-4
CUBS	6.32E-2	4.02E-2	2.74E-3
ERS	1.50E-2	1.42E-2	7.40E-4
MOS	6.50E-3	4.33E-3	2.72E-4
grep	9.41E-1	8.82E-1	4.20E-1
FastJson	2.87E-2	2.84E-2	2.94E-2

Step 1. Select two relevant and distinct complete test frames as a candidate pair.
Step 2. Determine whether or not the selected candidate pair is useful for identifying an MR, and if yes, then provide the corresponding MR description.
Step 3. 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 guideline, we identified 142, 184, 1130, and 3,512 MRs for ACM, CUBS, ERS and MOS, respectively (refer to the sixth column of Table 3). Note that the MRs generated by METRIC are meaningful in practice, as demonstrated by previous empirical studies. This is mainly due to the basic notion of METRIC: It identifies MRs by selecting two relevant and distinct complete test frames as a candidate pair, which normally correspond to specific scenarios of the specification of SUT.

METRIC normally can generate a huge amount of MRs even for a small number of categories and choices. The programs grep and FastJson have a relatively large number of complex categories and choices, which could result in too many MRs if METRIC were used. In our study, to control the scale of experiments, we obtained MRs of grep and FastJson manually by analyzing their necessary properties. Specifically, for grep, Barus et al. [7] has identified 12 MRs, which were re-used in this study. The development of the MRs for grep were restricted to the available test cases of grep. In particular, these MRs are only associated with the regular expression parameter of grep's input. The details of these 12 MRs can be found in the previous study [7]. We derived MRs for FastJson based on the generated test cases, best practices, and all faults reported in Alibaba repository. In total, 17 MRs were identified particularly for testing the deserialization of FastJson.

4.8 Parameters for MAPT* and RAPT*

Previous studies [39, 42, 62, 70] have given some guidelines on how to set ε and δ of the reward-punishment mechanism used in RAPT*. We followed these studies to set $\varepsilon = 0.05$ and calculated the proper δ according to the following formula [39]:

$$\frac{1}{\theta_M} - 1 < \frac{\varepsilon}{\delta} < \frac{1}{\theta_\Delta} - 1, \tag{15}$$

where θ_M and θ_Δ are the largest and the second largest failure rates amongst all partitions, respectively. Note that the value of θ_Δ varied with versions of FastJson. According to Formula 15, the theoretical values of δ in each program are shown in Table 5.

All faults in our experiments are non-trivial and thus not easy to be killed. Thus, the value of Bou_i could not be too small. We set $Bou_i = 70\% \times k_i$, where k_i is the number of test cases selected from c_i.

To set the values of γ and τ for MAPT*, we followed the previous study [62], in which we conducted a series of preliminary experiments. We observed that γ and τ could neither be too large nor too small, which imply that the change in probability profile would be either very

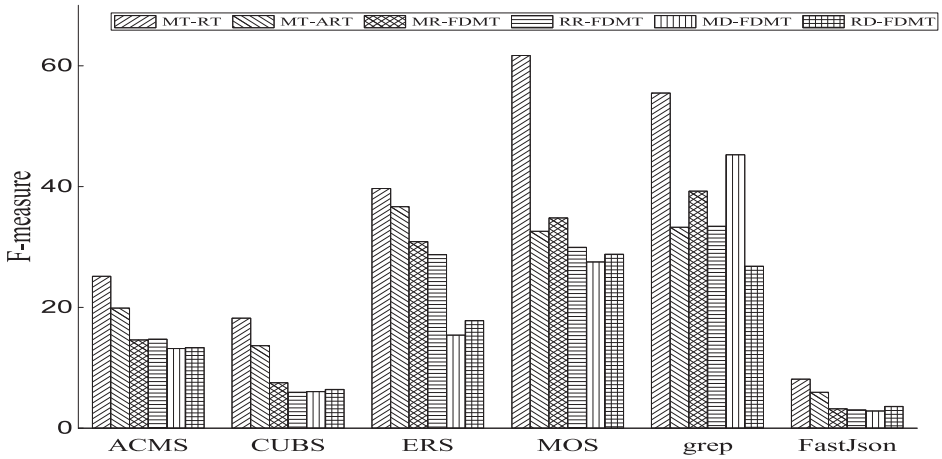


Fig. 2. F-measures on ACMS, CUBS, ERS, MOS, grep, and FastJson.

dramatic or very marginal. Both situations might result in the "unfair" (either too big or too small) award/punishment to certain partitions. Previous experimental results [62] showed that $\gamma = \tau = 0.1$ were fair settings, which were also used in this study.

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 4 GB. The test scripts were generated using Java, bash shell, and Python. In the experiments, we repeatedly ran the testing using each technique 30 times (as suggested in a previous study [2]) with different random seeds to guarantee the statistically reliable mean values of the metrics.

5 EXPERIMENTAL RESULTS

5.1 RQ1: Fault-Detection Effectiveness

Figures 2 and 3 depict the experimental results of F-measure and F2-measure, respectively. Note that since ERS and MOS each has only one fault, they were excluded from the experiments on F2-measure. It is clearly shown that on all subject programs, all four FDMT techniques (MD-FDMT, MR-FDMT, RD-FDMT, and RR-FDMT) and MT-ART used fewer test cases than MT-RT to detect the first and second faults, as indicated by their smaller values of F-measure and F2-measure. The results indicate that FDMT and MT-ART did improve the fault-detection effectiveness of MT.

In comparison with MT-ART, among the four proposed FDMT techniques, RR-FDMT and RD-FDMT had lower mean value of F-measure than MT-ART for all subject programs. MT-ART had lower mean value of F-measure than MR-FDMT for MOS and grep. Only for grep, did MT-ART have lower mean value of F-measure than MD-FDMT. In terms of F2-measure, all four FDMT techniques performed better than MT-ART for testing ACMS, and CUBS, and three of four FDMT techniques (MR-FDMT, MD-FDMT, and RD-FDMT) performed better than MT-ART for testing FastJson. However, MT-ART had lower mean value of F2-measure than four FDMT techniques for testing grep.

For answering sub-research question RQ1.1, we further compared the performance of the two source test case selection strategies, MAPT* and RAPT*. Out of 12 comparison scenarios for F-measure (that is, "MD-FDMT vs. RD-FDMT" and "MR-FDMT vs. RR-FDMT" on six programs),

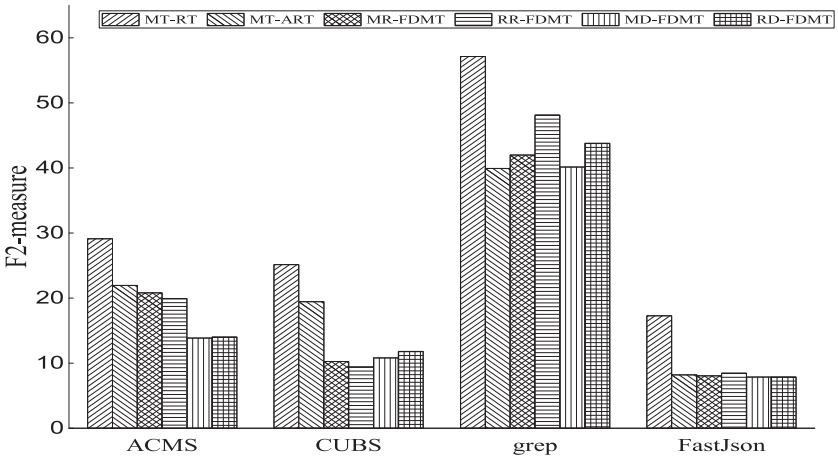


Fig. 3. F2-measures on ACMS, CUBS, grep, and FastJson.

MAPT* had better performance in six cases, and RAPT* in other six. MAPT* outperformed RAPT* in five out of eight cases (two comparison pairs on four programs) when comparing their F2-measures. Briefly speaking, the impacts of MAPT* and RAPT* on FDMT's fault-detection effectiveness were indistinguishable.

For RQ1.2, in the comparison between the two MR selection strategies (RMRS and DOMR), we can observe that DOMR outperformed RMRS in nine out of 12 comparison scenarios (that is, "MD-FDMT vs. MR-FDMT" and "RD-FDMT vs. RR-FDMT" on six programs) in terms of F-measure. DOMR helped deliver smaller F2-measure than RMRS in six out of eight cases for F2-measure. In other words, DOMR seemed better than RMRS with respect to F-measure and F2-measure.

We further evaluated the magnitude of the performance difference between each pair of techniques by calculating Cohen's D, a popular effect size measure. Note that neither F-measure nor F2-measure follows normal distribution, so we could not directly use their absolute values to calculate the effect size. In our study, we first calculated the so-called "performance improvement ratio" [62], which is defined as the improvement ratio of F-measure/F2-measure of one technique over the other. For example, the F-measures of MD-FDMT and MT-RT on ACMS were 13.2 and 25.1, respectively. The performance improvement ratio of MD-FDMT over MT-RT can then be calculated as $\frac{25.1-13.2}{25.1} = 0.4741$. The effect size for each pair of techniques was worked out based on these ratios according to a standard formula $\frac{\mu_1-\mu_2}{\sigma}$ [33], where μ_1 and μ_2 refer to the average values of two datasets, and σ denotes the standard deviation of the aggregation of these two sets. Tables 6 and 7 give the effect size results for F-measure and F2-measure, respectively. In each pair of techniques (first column of the tables), the former one outperformed the latter one in terms of the average F-measure/F2-measure.

Based on Tables 6 and 7, we further evaluated the extent of performance difference among the six techniques under study according to some rules of thumb for effect size [57]. It is clearly shown that the performance improvements of all four FDMT techniques over MT-RT were "very large" (effect size > 1.2) in terms of both F-measure and F2-measure, and MT-ART also performed better than MT-RT to a "very large" extent.

The effect size results also confirm our answer to RQ1.1, that is, it is difficult to distinguish the impacts of MAPT* and RAPT* (the two source test case selection strategies) on FDMT's performance. On F-measure, RD-FDMT was better than MD-FDMT to a "very small" extent ($0.2 \geq$ effect size > 0.01), and RR-FDMT was better than MR-FDMT to a "small" extent. However, MD-FDMT

Table 6. Effect Size Showing the Magnitude of F-measure's Difference between Two Techniques (the Former Technique Had Smaller Mean Value of F-measure than the Latter)

Pair of Techniques	Effect Size	Difference
MT-ART vs. MT-RT	1.6771	Very Large
MR-FDMT vs. MT-RT	1.8142	Very Large
RR-FDMT vs. MT-RT	1.8549	Very Large
MD-FDMT vs. MT-RT	1.8274	Very Large
RD-FDMT vs. MT-RT	1.9807	Very Large
MR-FDMT vs. MT-ART	0.9629	Large
RR-FDMT vs. MT-ART	1.2194	Very Large
MD-FDMT vs. MT-ART	1.2735	Very Large
RD-FDMT vs. MT-ART	1.6088	Very Large
RD-FDMT vs. MD-FDMT	0.1872	Very Small
RD-FDMT vs. MR-FDMT	0.9779	Large
RD-FDMT vs. RR-FDMT	0.5631	Medium
MD-FDMT vs. RR-FDMT	0.2517	Small
MD-FDMT vs. MR-FDMT	0.5996	Medium
RR-FDMT vs. MR-FDMT	0.4046	Small

Table 7. Effect Size Showing the Magnitude of F2-measure's Difference between Two Techniques (the Former Technique Had Smaller Mean Value of F2-measure than the Latter)

Pair of Techniques	Effect Size	Difference
MT-ART vs. MT-RT	1.7759	Very Large
MR-FDMT vs. MT-RT	1.7962	Very Large
RR-FDMT vs. MT-RT	1.6912	Very Large
MD-FDMT vs. MT-RT	1.9058	Very Large
RD-FDMT vs. MT-RT	1.8567	Very Large
MR-FDMT vs. MT-ART	0.6688	Medium
RR-FDMT vs. MT-ART	0.4935	Small
MD-FDMT vs. MT-ART	1.1454	Large
RD-FDMT vs. MT-ART	0.9395	Large
MD-FDMT vs. RD-FDMT	0.2253	Small
RD-FDMT vs. MR-FDMT	0.2747	Small
RD-FDMT vs. RR-FDMT	0.3443	Small
MD-FDMT vs. RR-FDMT	0.5311	Medium
MD-FDMT vs. MR-FDMT	0.4929	Small
MR-FDMT vs. RR-FDMT	0.1014	Very Small

outperformed RD-FDMT in terms of F2-measure to a "small" extent, and the outperformance of MR-FDMT over RR-FDMT on F2-measure was "very small".

Also reinforced is the answer to RQ1.2 that DOMR was superior to RMRS in helping FDMT achieve smaller F-measure and F2-measure. The extent how RD-FDMT outperformed RR-FDMT was "medium" ($0.8 \geq$ effect size > 0.5) and "small" in terms of F-measure and F2-measure, respectively. The outperformance of MD-FDMT over MR-FDMT was "medium" and "small" on F-measure and F2-measure, respectively. The superior performance of DOMR can also be confirmed by the

remaining two pairs of techniques (MD-FDMT vs. RR-FDMT and RD-FDMT vs. MR-FDMT)—No matter which source test case selection strategies were used, DOMR generally performed better than RMRS, with the difference ranging from "small" to "large".

Answer to RQ1: All four FDMT techniques improved the fault-detection effectiveness of MT-RT to a very large extent, as demonstrated by their much smaller F-measure and F2-measure. FDMT generally outperformed MT-ART in terms of both F-measure and F2-measure. For RQ1.1, the two source test case selection strategies, MAPT* and RAPT*, had indistinguishable impacts on FDMT's performance. For RQ1.2, with respect to MR selection techniques, DOMR outperformed RMRS.

5.2 RQ2: Overall Testing Time

The results of F-time and F2-time are depicted in Figures 4 and 5, respectively. Different from the observation made on F-measure and F2-measure, MT-RT did not always have the longest F-time or F2-time. All four FDMT techniques had shorter F-time than MT-RT on five out of six programs. In terms of F2-time, MR-FDMT was better than MT-RT on three out of four programs; while the other three FDMT techniques each outperformed MT-RT on two out of four programs. MT-ART had a longer F-time than all four FDMT techniques for four out of six subject programs and a longer F2-time for two out of four programs.

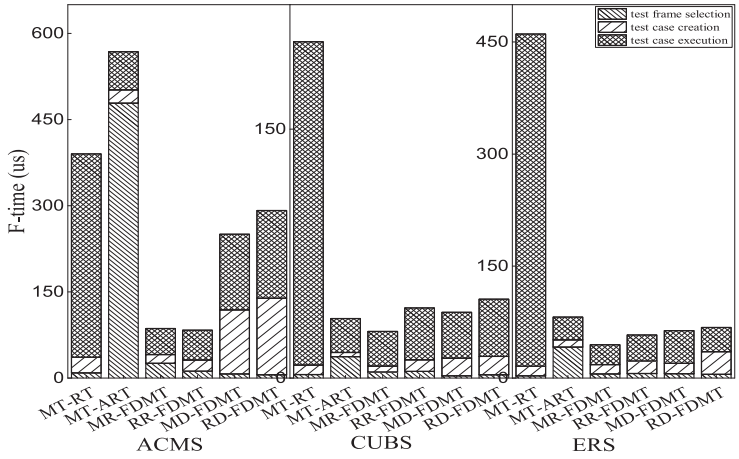
Comparing the two source test case selection strategies (RQ2.1), MAPT* seemed marginally superior to RAPT* in reducing the overall testing time. MAPT* (used in either MD-FDMT or MR-FDMT) performed better than RAPT* (either RD-FDMT or RR-FDMT) in nine out of twelve comparison scenarios for F-time; and the former outperformed the latter in six out of eight comparison scenarios for F2-time. With respect to the MR selection strategies (RQ2.2), it is hard to distinguish the performance of RMRS (used in either MR-FDMT or RR-FDMT) and DOMR (either MD-FDMT or RD-FDMT) in terms of F-time and F2-time.

Another interesting observation made in Figures 4 and 5 is the significant difference among the three components of overall testing time. Across all subject programs, all FDMT techniques and MT-RT took the shortest time for complete test frame selection, which was negligible as compared with that for test case creation and execution. Note that the feedback-directed selection of complete test frames (which mainly consists of dynamic adjustment of test profiles and the selection of partitions based on test profiles) is the core process in MAPT* and RAPT* and thus their major difference from RT. The very short complete test frame selection time implies that the extra computation overhead incurred by MAPT* and RAPT* would not affect the overall time performance too much.

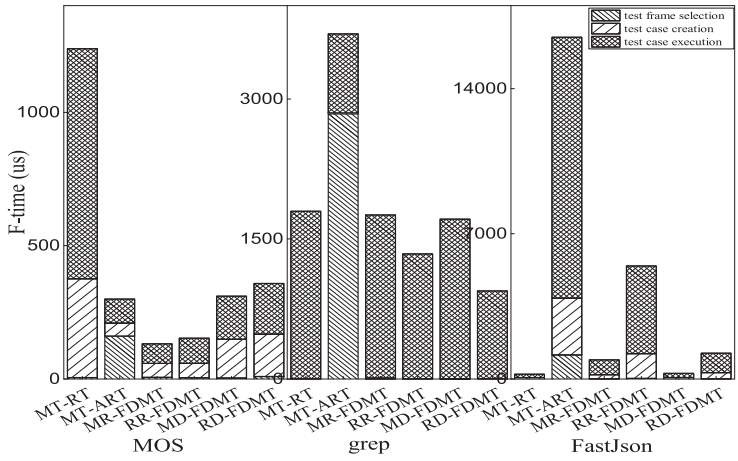
We also calculated the effect size for measuring the magnitude of difference between each pair of techniques on F-time and F2-time, as summarized in Tables 8 and 9, respectively. As observed from Tables 8 and 9, the performance improvements of the four FDMT techniques over MT-ART varied from "small" to "medium" and from "very small" to "medium" in terms of F-time and F2-time, respectively. MT-RT performed better than MT-ART to a "medium" extent on both F-time and F2-time.

For further answering RQ2.1, MD-FDMT was better than RD-FDMT in both F-time and F2-time, with "small" difference. MR-FDMT outperformed RR-FDMT in both F-time and F2-time to a "medium" extent. These comparison results confirmed the above observation that MAPT* might be better than RAPT* in terms of helping reducing FDMT's overall testing time, but the difference was marginal.

For RQ2.2, DOMR marginally outperformed RMRS with regard to F-time and F2-time. The FDMT techniques with DOMR (MD-FDMT and RD-FDMT) were superior to those with RMRS (MR-FDMT



(a) F-time on ACMS, CUBS, ERS

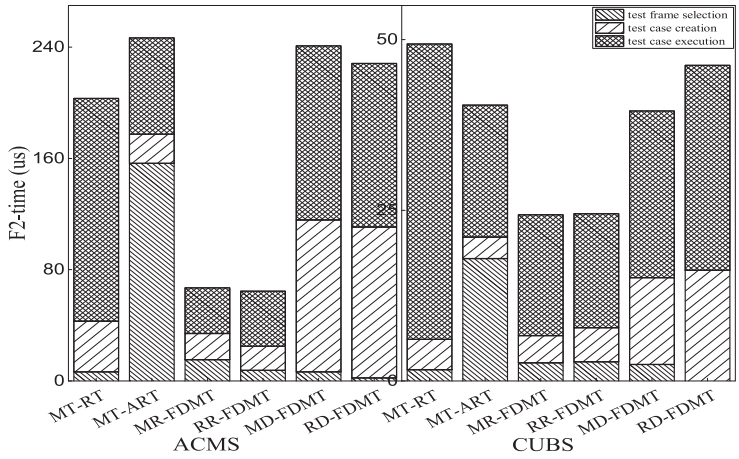


(b) F-time on MOS, grep, and FastJson

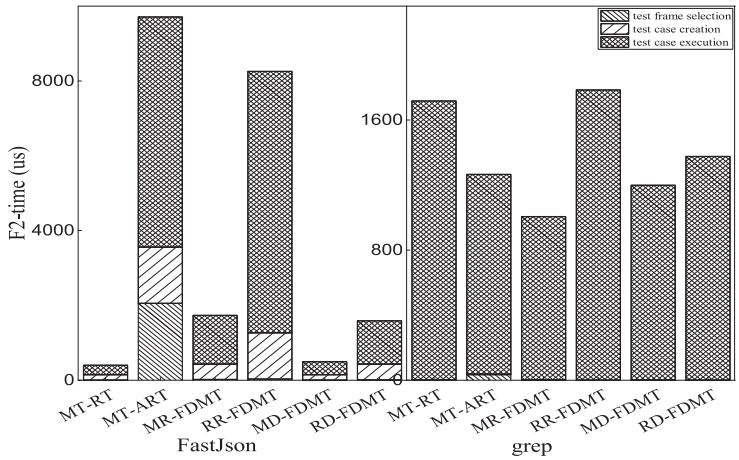
Fig. 4. Testing results of F-time.

and RR-FDMT) from a "small" to "medium" extent in three comparison cases (MD-FDMT vs. MR-FDMT, MD-FDMT vs. RR-FDMT, and RD-FDMT vs. RR-FDMT) for F-time. The outperformance of the former over the latter in F2-time varied from "very small" to "medium". Considering both the comparison results on strategies for selecting source test cases and MRs, we observed that MD-FDMT (the one using MAPT* and DOMR) might be the best FDMT technique for reducing the overall testing time.

Answer to RQ2: Among all four FDMT techniques, MD-FDMT was the only one that could generally outperform MT-RT in terms of both F-time and F2-time. In general, all FDMT techniques were better than MT-ART in terms of the overall testing time. For RQ2.1, MAPT* appeared to be superior to RAPT* in helping FDMT achieve a good performance in time. For RQ2.2, DOMR was better than RMRS in terms of testing efficiency, but the difference was marginal.



(a) F2-time on ACMS, CUBS



(b) F2-time on grep, and FastJson

Fig. 5. Testing results of F2-time.

5.3 Threats To Validity

The threat to internal validity is mainly related to the implementations of the testing techniques, which involved a moderate amount of programming work. Our code has been cross-checked by different individuals, and we are confident that all techniques were correctly implemented.

One possible threat to external validity is about the subject programs and seeded faults under evaluation. Although the four laboratory programs are not very complex, they do implement real-life business scenarios of diverse application domains. Furthermore, we chose two large-scale programs (grep and FastJson). Although we endeavored to improve the generalizability of our findings by selecting subject programs from a variety of domains, we anticipate that the evaluation results may vary slightly with different types of programs.

The metrics used in our study are simple in concept and straightforward to apply, so there should be little threat to construct validity.

Table 8. Effect Size Showing the Magnitude of F-time's Difference between Two Techniques (the Former Technique Had Smaller Mean Value of F-time than the Latter)

Pair of Techniques	Effect Size	Difference
MT-RT vs. MT-ART	0.5944	Medium
MR-FDMT vs. MT-RT	0.0834	Very Small
MT-RT vs. RR-FDMT	0.4984	Small
MD-FDMT vs. MT-RT	1.2436	Very Large
MT-RT vs. RD-FDMT	0.1614	Very Small
MR-FDMT vs. MT-ART	0.5975	Medium
RR-FDMT vs. MT-ART	0.4200	Small
MD-FDMT vs. MT-ART	0.6151	Medium
RD-FDMT vs. MT-ART	0.5830	Medium
MR-FDMT vs. RR-FDMT	0.5045	Medium
MD-FDMT vs. MR-FDMT	0.3459	Small
MR-FDMT vs. RD-FDMT	0.1789	Very Small
MD-FDMT vs. RR-FDMT	0.5630	Medium
RD-FDMT vs. RR-FDMT	0.4567	Small
MD-FDMT vs. RD-FDMT	0.4750	Small

Table 9. Effect Size Showing the Magnitude of F2-time's Difference between Two Techniques (the Former Technique Had Smaller Mean Value of F2-time than the Latter)

Pair of Techniques	Effect Size	Difference
MT-RT vs. MT-ART	0.7472	Medium
MT-RT vs. MR-FDMT	0.3582	Small
MT-RT vs. RR-FDMT	0.7050	Medium
MD-FDMT vs. MT-RT	0.1176	Very Small
MT-RT vs. RD-FDMT	0.4338	Small
MR-FDMT vs. MT-ART	0.6901	Medium
RR-FDMT vs. MT-ART	0.1209	Very Small
MD-FDMT vs. MT-ART	0.7493	Medium
RD-FDMT vs. MT-ART	0.6872	Medium
MR-FDMT vs. RR-FDMT	0.6359	Medium
MD-FDMT vs. MR-FDMT	0.3704	Small
MR-FDMT vs. RD-FDMT	0.0249	Very Small
MD-FDMT vs. RR-FDMT	0.7074	Medium
RD-FDMT vs. RR-FDMT	0.6325	Medium
MD-FDMT vs. RD-FDMT	0.4458	Small

With regard to the threat to conclusion validity, as reported for empirical studies in the field of software engineering [2], at least 30 trials are necessary to ensure the statistical significance of results. Accordingly, we have run a sufficient number of trials to ensure the reliability of our experimental results. Furthermore, we calculated the effect size to measure the magnitude of difference between each pair of techniques on every metric.

6 RELATED WORK

As justified by extensive studies [18, 59], MT is not only able to address the oracle problem, but also capable of detecting a variety of faults. MRs, the core component of MT, have attracted lots of attentions from the research community. One major research area on MR acquisition is how to construct them systematically. Zhang et al. [72] developed a search-based approach to automatic inference of polynomial MRs for a SUT, where a set of parameters is used to represent polynomial MRs, and the problem of inferring MRs is turned into a problem of searching for suitable values of the parameters. Then, particle swarm optimization is used to solve the search problem. Ayerdi et al. [4] proposed a genetic algorithm based approach for generating MRs based on some "input patterns". Sun et al. [65] proposed a data-mutation directed MR acquisition methodology, where data mutation is employed to construct input relations and the generic mapping rule associated with each mutation operator is used to determine output relations. Chen et al. [22] developed a more general specification-based method, namely METRIC, based on the concepts of categories and choices in CPM [51]. Also provided was a tool called MR-GENerator for identifying MRs. Sun et al. [64] extended METRIC to METRIC+, which introduced the output categories and choices into MT and thus provided a more comprehensive and systematic framework for the MR identification. Unlike the above studies that identify MRs from scratch, some researchers [41, 55] proposed the construction of MRs based on existing ones, which might have been derived in an arbitrary and ad hoc way. FDMT also makes use of the concepts of categories and choices, but the purpose is to dynamically select the next MR to be used from a set of already identified MRs rather than the MR identification.

Given that there could be a huge amount of MRs for a system, it is a challenging job to select the most appropriate MRs, especially those efficient in fault detection. In a pioneer study, Chen et al. [19] recommended that MRs be selected such that each one should have different sensitivity to various fault types. In a case study [17], it was conjectured that good MRs should be identified with regard to the algorithm that the program follows. Mayer and Guderlei [49] confirmed the usefulness of white-box considerations in the MR identification. Asrafi et al. [3] analyzed the relationship between MT's fault-detection efficacy and code coverage, and pointed out that the MRs causing different execution behaviors (measured by the cumulative code coverage of source and follow-up test cases) have high probability to detect faults. Just et al. [36] compared the MRs derived from the component level and those from the system level, and observed that the former was more effective than the later in fault detection. However, all these techniques required the execution of all MT test cases constructed from every MR before deciding which MRs should be selected. By contrast, DOMR in FDMT attempts to dynamically choose the most appropriate MR for the next testing step, and thus avoids the high overhead of MT.

In addition to MRs, source test cases are another factor that influences the performance of MT. Lots of previous studies have used RT to generate source test cases [59]. Chen et al. [19] compared the effects of source test cases generated by special value testing and RT on the effectiveness of MT, and found that MT can be used as a complementary test method to special value testing. Dong et al. [27] proposed a path–combination–based MT method that first generates symbolic inputs for all executable paths and mines relationships among these symbolic inputs and their outputs. MRs could then be constructed on the basis of these relationships, and actual test cases would be created corresponding to the symbolic inputs. Barus et al. [7] employed ART [21] instead of RT to increase the diversity of source test cases. Different from all these studies, FDMT, for the first time in the context of MT, utilizes the historic testing data, especially those from the test execution process, as the feedback information to guide the source test case selection. Specifically, the feedback information is leveraged to adjust the test profile, which, in turn, decides the probabilities of different partitions to be selected for creating the next test case.

More recently, Spieker and Gotlieb [61] proposed a new method to adaptively select MRs that are deemed to have high fault-detection potentials. They developed a so-called "**adaptive metamorphic testing**" (**AMT**) algorithm based on reinforcement learning with contextual bandits, which aims at dynamically adjusting the selection of MRs along the testing process. AMT has the similar purpose to that of our DOMR strategy (that is, the dynamic selection of appropriate MRs, unlike the "static" identification of "good" MRs in many other studies), but based on a different intuition about how to judge which MRs are more likely to be effective in fault detection. In addition, AMT did not consider the source test case selection process, which is an essential part of our FDMT approach. Besides these differences at the intuition level, AMT also differs from FDMT at the operation level, which makes it extremely difficult to fairly compare these two techniques. FDMT assumes the availability of test partitions and MRs, which are all generated based on the concepts of CPM in this study. One critical step of AMT is to extract context features to support the implementation of reinforcement learning. A fair empirical comparison between FDMT and AMT would require the extraction of proper context features for general-purpose systems (such as subject programs in our experiments), which itself is a challenging job and deserves a substantial separate study.

The idea of feedback has been widely used to generate or select test cases in the field of software testing. Pacheco et al. proposed feedback-directed RT technique [52, 53], which improves random test generation by incorporating feedback obtained from execution behaviors of generated test cases. Chen et al. [21] proposed ART to make use of the feedback information about the distributions of already executed test cases over the input domain to generate the next test case. Grechanik et al. [29] proposed a feedback-directed learning testing technique to find more performance problems, which first learns rules from execution traces of applications and then uses the rules to automatically select test input data. Segall and Brill [58] proposed a feedback-driven combinatorial test technique to cope with the unpredictability and uncontrollability of Cloud environments, which first collects the monitoring information from the Cloud and then drives or adjusts the subsequent test scenarios. Feedback has also been leveraged to improve fuzz testing techniques. For instance, FairFuzz first executes a small number of test cases to identify branches that are difficult to cover, and then uses the branches as feedback to guide seed mutation [38]. PerfFuzz first collects the feedback information such as code coverage as well as values associated with the program components of interest, and then determines whether the executed test cases can be added to the seed queue based on the collected feedback information [37].

In this study, the use of feedback information to control the testing procedure aligns with software cybernetics [11, 13, 69]. In fact, the source test case selection strategies used in FDMT (MAPT* and RAPT*) were derived from APT [62], which was also developed on the basis of software cybernetics. One pioneer study in this field is adaptive testing [10], where the Markov chain controller is used to optimize software testing as a control problem. Cai et al. [12] improved adaptive testing through "fixed-memory feedback for on-line parameter estimations". Lv et al. [43] further proposed an efficient adaptive testing technique with the gradient descent method, which is particularly useful for the estimation of software reliability. Similar to APT, DRT [39] also adjusts the test profile along the testing process. All these techniques have the assumption that the test oracle exists and hence they use the test execution results as the feedback information. However, their applicability would be hindered by the oracle problem. Integrating MRs into the testing, FDMT helps address the oracle problem for APT, and thus potentially for many other software cybernetics techniques.

7 CONCLUSION

MT is a testing method that can effectively alleviate the oracle problem as well as create new test cases complementary to those generated by other traditional techniques. Although MT has

consistently shown a high effectiveness in detecting a variety of real-life software bugs, there is still much room to improve its cost-effectiveness. In this article, we proposed a new approach called FDMT, which makes use of the historical information of test executions to dynamically select the source test cases and MRs with higher fault-detection potentials during the test process. A comprehensive framework was designed to implement FDMT, where the whole testing process can be regarded as a feedback control system. The core part of the framework is a feedback loop, which consists of the basic MT components, SUT, the historical testing data, and the testing strategies, where the testing history is utilized in selecting the next source test case and appropriate MRs. Based on software cybernetics, algorithms were designed for the selection of appropriate source test cases and MRs, leading to four concrete FDMT techniques, namely MR-FDMT, RR-FDMT, MD-FDMT, and RD-FDMT.

Empirical studies were conducted to evaluate and compare FDMT with the traditional MT on six programs, including four laboratory ones, one GNU application, and an industrial program. The experimental results showed that all four FDMT techniques could outperform MT in terms of the number of test cases, thus improving the fault-detection effectiveness. MD-FDMT was generally superior to MT, with respect to the time required for detecting various faults, implying higher efficiency. In other words, FDMT significantly enhanced the cost-effectiveness of MT. Among all four FDMT techniques, MD-FDMT generally delivered the highest performance improvement over traditional MT. Moreover, the source test case selection strategies used in FDMT (i.e., MAPT* and RAPT*) can be considered as an improvement of the original APT techniques (MAPT and RAPT, respectively), in light of their capabilities of addressing the oracle problem.

For our future work, we will develop new strategies for selecting source test cases and MRs on the basis of the general FDMT framework, aiming at optimizing MT's cost-effectiveness. It is necessary to conduct larger-scale empirical studies with more real-life faults to evaluate the performance of all strategies for FDMT. It is also promising to integrate other advanced techniques, such as AMT [61], into the framework. In addition, there exist other software cybernetics techniques like APT, which are confronted with the challenge of oracle problem. Hence, another promising research direction is to improve the applicability of these techniques using our FDMT framework.

REFERENCES

[1] John Ahlgren, Maria Eugenia Berezin, Kinga Bojarczuk, Elena Dulskyte, Inna Dvortsova, Johann George, Natalija Gucevska, Mark Harman, Maria Lomeli, Erik Meijer, Silvia Sapora, and Justin Spahr. 2021. Testing web enabled simulation at scale using metamorphic testing. In *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP'21)*. 140–149.

[2] Andrea Arcuri and Lionel Briand. 2011. 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)*. 1–10.

[3] Mahmuda Asrafi, Huai Liu, and Fei-Ching Kuo. 2011. On testing effectiveness of metamorphic relations: A case study. In *Proceedings of the 15th International Conference on Secure Software Integration and Reliability Improvement (SSIRI'11)*. 147–156.

[4] Jon Ayerdi, Valerio Terragni, Aitor Arrieta, Paolo Tonella, Goiuria Sagardui, and Maite Arratibel. 2021. Generating metamorphic relations for cyber-physical systems with genetic programming: An industrial case study. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'21)*. 1264–1274.

[5] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525.

[6] Arlinta Christy Barus, Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Robert Merkel, and Gregg Rothermel. 2016. A cost-effective random testing method for programs with non-numeric inputs. *IEEE Transactions on Computers* 65, 12 (2016), 3509–3523.

[7] Arlinta Christy Barus, Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, and Heinz W. Schmidt. 2016. 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)*. 5–11.

[8] Pierre Bourque and Richard E. Fairley (Eds.). 2014. *SWEBOK: Guide to the Software Engineering Body of Knowledge* (version 3.0 ed.). IEEE Computer Society, Los Alamitos, CA.
[9] Susan S. Brilliant, John C. Knight, and P. E. Ammann. 1990. On the performance of software testing using multiple versions. In *Proceedings of the 20th International Symposium on Fault-Tolerant Computing (FTCS'90)*. 408–415.
[10] Kai-Yuan Cai. 2002. Optimal software testing and adaptive software testing in the context of software cybernetics. *Information and Software Technology* 44, 14 (2002), 841–855.
[11] Kai-Yuan Cai, Tsong Yueh Chen, and T. H. Tse. 2002. Towards research on software cybernetics. In *Proceedings of the 7th IEEE International Symposium on High Assurance Systems Engineering (HSE'02)*. 240–241.
[12] Kai-Yuan Cai, Bo Gu, Hai Hu, and Yong-Chao Li. 2007. Adaptive software testing with fixed-memory feedback. *Journal of Systems and Software* 80, 8 (2007), 1328–1348.
[13] João W. Cangussu, Kai-Yuan Cai, Scott D. Miller, and Aditya P. Mathur. 2007. Software cybernetics. *Wiley Encyclopedia of Computer Science and Engineering* (2007). https://doi.org/10.1002/9780470050118.ecse707
[14] Alvin Chan, Lei Ma, Felix Juefei-Xu, Yew-Soon Ong, Xiaofei Xie, Minhui Xue, and Yang Liu. 2021. Breaking neural reasoning architectures with metamorphic relation-based adversarial examples. *IEEE Transactions on Neural Networks and Learning Systems* (2021). https://doi.org/10.1109/TNNLS.2021.3072166
[15] Tsong Yueh Chen, Shing C. Cheung, and Shiu Ming Yiu. 1998. *Metamorphic Testing: A New Approach for Generating Next Test Cases*. Technical Report. Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, Tech. Rep. HKUST-CS98-01.
[16] Tsong Yueh Chen, Joshua W. K. Ho, Huai Liu, and Xiaoyuan Xie. 2009. An innovative approach for testing bioinformatics programs using metamorphic testing. *BMC Bioinformatics* 10, 1 (2009), 24–32.
[17] Tsong Yueh Chen, D. H. Huang, T. H. Tse, and Zhi Quan Zhou. 2004. Case studies on the selection of useful relations in metamorphic testing. In *Proceedings of the 4th IberoAmerican Symposium on Software Engineering and Knowledge Engineer-ing (JIISIC'04)*. 569–583.
[18] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic testing: A review of challenges and opportunities. *ACM Computing Surveys* 51, 1 (2018), 4:1–4:27.
[19] Tsong Yueh Chen, Fei-Ching Kuo, Ying Liu, and Antony Tang. 2004. 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)*. 128–134.
[20] Tsong Yueh Chen, Fei-Ching Kuo, Wenjuan Ma, Willy Susilo, Dave Towey, Jeffrey Voas, and Zhi Quan Zhou. 2016. Metamorphic testing for cybersecurity. *Computer* 49, 6 (2016), 48–55.
[21] Tsong Yueh Chen, Hing Leung, and Ieng Kei Mak. 2004. Adaptive random testing. In *Proceedings of the 9th Asian Computing Science Conference (ASIAN'04)*. 320–329.
[22] Tsong Yueh Chen, Pak-Lok Poon, and Xiaoyuan Xie. 2016. METRIC: Metamorphic relation identification based on the category-choice framework. *Journal of Systems and Software* 116 (2016), 177–190.
[23] Tsong Yueh Chen and T. H. Tse. 2021. New visions on metamorphic testing after a quarter of a century of inception. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'21)*. 1487–1490.
[24] Tsong Yueh Chen and Yuen-Tak Yu. 1994. On the relationship between partition and random testing. *IEEE Transactions on Software Engineering* 20, 12 (1994), 977–980.
[25] David Coppit and Jennifer M. Haddox-Schatz. 2005. On the use of specification-based assertions as test oracles. In *Proceedings of the 29th IEEE/NASA Software Engineering Workshop (SEW'05)*. 305–314.
[26] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering* 10, 4 (2005), 405–435.
[27] Guowei Dong, Tao Guo, and Puhan Zhang. 2013. Security assurance with program path analysis and metamorphic testing. In *Proceedings of the 4th IEEE International Conference on Software Engineering and Service Science (ICSESS'13)*. 193–197.
[28] GNU Project. 2006. GNU grep. (2006). Retrieved from http://www.gnu.org/software/grep.
[29] Mark Grechanik, Chen Fu, and Qing Xie. 2012. Automatically finding performance problems with feedback-directed learning software testing. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*. 156–166.
[30] Richard Hamlet. 2002. Random testing. In *Proceedings of the Encyclopedia of Software Engineering*. John Wiley & Sons, Inc.
[31] Pinjia He, Clara Meister, and Zhendong Su. 2020. Structure-invariant testing for machine translation. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE'20)*. 961–973.
[32] Xiao He, Xingwei Wang, Jia Shi, and Yi Liu. 2020. Testing high performance numerical simulation programs: Experience, lessons learned, and open issues. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'20)*. 502–515.

[33] Larry Hedges and Ingram Olkin. 2014. *Statistical Methods for Meta-Analysis*. Academic Press.
[34] John Hughes. 2019. How to specify it!. In *Proceedings of the 20th International Symposium on Trends in Functional Programming (TFP'19) (Lecture Notes in Computer Science)*, Vol. 12053. 58–83.
[35] Zhan-Wei Hui, Song Huang, Tsong Yueh Chen, Man F. Lau, and Sebastian Ng. 2017. Identifying failed test cases through metamorphic testing. In *Proceedings of the 28th International Symposium on Software Reliability Engineering Workshops (ISSREW'17)*. 90–91.
[36] René Just and Franz Schweiggert. 2010. Automating software tests with partial oracles in integrated environments. In *Proceedings of the 5th Workshop on Automation of Software Test (AST'10)*. 91–94.
[37] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18)*. 254–265.
[38] Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 475–485.
[39] Ye Li, Bei-Bei Yin, Junpeng Lv, and Kai-Yuan Cai. 2015. Approach for test profile optimization in dynamic random testing. In *Proceedings of the 39th International Computer Software and Applications Conference (COMPSAC'15)*, Vol. 3. 466–471.
[40] Huai Liu, Fei-Ching Kuo, Dave Towey, and Tsong Yueh Chen. 2014. How effectively does metamorphic testing alleviate the oracle problem? *IEEE Transactions on Software Engineering* 40, 1 (2014), 4–22.
[41] Huai Liu, Xuan Liu, and Tsong Yueh Chen. 2012. A new method for constructing metamorphic relations. In *Proceedings of the 12th International Conference on Quality Software (QSIC'12)*. 59–68.
[42] Junpeng Lv, Hai Hu, and Kai-Yuan Cai. 2011. 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)*. 19–24.
[43] Junpeng Lv, Bei-Bei Yin, and Kai-Yuan Cai. 2014. On the asymptotic behavior of adaptive testing strategy for software reliability assessment. *IEEE Transactions on Software Engineering* 40, 4 (2014), 396–412.
[44] Pingchuan Ma, Shuai Wang, and Jin Liu. 2020. Metamorphic testing and certified mitigation of fairness violations in NLP models. In *Proceedings of the 29th International Joint Conference on Artificial Intelligence (IJCAI'20)*. 458–465.
[45] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. 2005. MuJava: An automated class mutation system. *Software Testing, Verification and Reliability* 15, 2 (2005), 97–133.
[46] Phu X. Mai, Fabrizio Pastore, Arda Goknil, and Lionel C. Briand. 2020. Metamorphic security testing for web systems. In *Proceedings of the 13th IEEE International Conference on Software Testing, Validation and Verification (ICST'20)*. 186–197.
[47] Muhammad Numair Mansur, Maria Christakis, and Valentin Wüstholz. 2021. Metamorphic testing of datalog engines. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'21)*. 639–650.
[48] Dusica Marijan, Arnaud Gotlieb, and Mohit Kumar Ahuja. 2019. Challenges of testing machine learning based systems. In *Proceedings of the 1st IEEE International Conference On Artificial Intelligence Testing (AITest'19)*. 101–102.
[49] Johannes Mayer and Ralph Guderlei. 2006. An empirical study on the selection of good metamorphic relations. In *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)*. 475–484.
[50] Andrew McNutt, Gordon Kindlmann, and Michael Correll. 2020. Surfacing visualization mirages. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI'20)*. 1–16.
[51] Thomas J. Ostrand and Marc J. Balcer. 1988. The category-partition method for specifying and generating fuctional tests. *Communications of the ACM* 31, 6 (1988), 676–686.
[52] Carlos Pacheco, Shuvendu K Lahiri, and Thomas Ball. 2008. Finding errors in. net with feedback-directed random testing. In *Proceedings of the 7th International Symposium on Software Testing and Analysis (ISSTA'08)*. 87–96.
[53] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*. 75–84.
[54] Krishna Patel and Robert M Hierons. 2018. A mapping study on testing non-testable systems. *Software Quality Journal* 26, 4 (2018), 1373–1413.
[55] Kun Qiu, Zheng Zheng, Tsong Yueh Chen, and Pak-Lok Poon. 2022. Theoretical and empirical analyses of the effectiveness of metamorphic relation composition. *IEEE Transactions on Software Engineering* 48, 3 (2022), 1001–1007.
[56] Marco Túlio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. 2020. Beyond accuracy: Behavioral testing of NLP models with checkList. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics (ACL'20)*. 4902–4912.
[57] Shlomo S. Sawilowsky. 2009. New effect size rules of thumb. *Journal of Modern Applied Statistical Methods* 8, 2 (2009), 467–474.

[58] Itai Segall and Rachel Tzoref-Brill. 2015. Feedback-driven combinatorial test design and execution. In *Proceedings of the 8th ACM International Systems and Storage Conference*. 1–6.
[59] Sergio Segura, Gordon Fraser, Ana B. Sanchez, and Antonio Ruiz-Cortés. 2016. A survey on metamorphic testing. *IEEE Transactions on Software Engineering* 42, 9 (2016), 805–824.
[60] Sergio Segura, José A. Parejo, Javier Troya, and Antonio Ruiz-Cortés. 2018. Metamorphic testing of RESTful web APIs. *IEEE Transactions on Software Engineering* 44, 11 (2018), 1083–1099.
[61] Helge Spieker and Arnaud Gotlieb. 2020. Adaptive metamorphic testing with contextual bandits. *Journal of Systems and Software* 165 (2020), 110574:1–110574:14.
[62] Chang-ai Sun, Hepeng Dai, Huai Liu, Tsong Yueh Chen, and Kai-Yuan Cai. 2018. Adaptive partition testing. *IEEE Transactions on Computers* 68, 2 (2018), 157–169.
[63] Chang-ai Sun, Hepeng Dai, Guan Wang, Dave Towey, Tsong Yueh Chen, and Kai-Yuan Cai. 2022. Dynamic random testing of web services: A methodology and evaluation. *IEEE Transactions on Services Computing* 15, 2 (2022), 736–751.
[64] Chang-ai Sun, An Fu, Pak-Lok Poon, Xiaoyuan Xie, Huai Liu, and Tsong Yueh Chen. 2021. METRIC+: A metamorphic relation identification technique based on input plus output domains. *IEEE Transactions on Software Engineering* 47, 9 (2021), 1765–1786.
[65] Chang-ai Sun, Yiqiang Liu, Zuoyi Wang, and W. K. Chan. 2016. μMT: A data mutation directed metamorphic relation acquisition methodology. In *Proceedings of 2016 IEEE/ACM the 1st International Workshop on Metamorphic Testing (MET'16), Co-Located with the 38th International Conference on Software Engineering (ICSE'16)*. 12–18.
[66] Chang-ai Sun, Zuoyi Wang, and Guan Wang. 2014. A property-based testing framework for encryption programs. *Frontiers of Computer Science* 8, 3 (2014), 478–489.
[67] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering (ICSE'18)*. 303–314.
[68] Xiaoyuan Xie, W. Eric Wong, Tsong Yueh Chen, and Baowen Xu. 2013. Metamorphic slice: An application in spectrum-based fault localization. *Information and Software Technology* 55, 5 (2013), 866–879.
[69] Hongji Yang, Feng Chen, and Suleiman Aliyu. 2017. Modern software cybernetics: New trends. *Journal of System and Software* 124 (2017), 157–169.
[70] Zijiang Yang, Beibei Yin, Junpeng Lv, Kai-Yuan Cai, Stephen S. Yau, and Jia Yu. 2014. Dynamic random testing with parameter adjustment. In *Proceedings of the 38th IEEE Annual International Computer Software and Applications Conference Workshops (COMPSACW'14)*. 37–42.
[71] Yuanyuan Yuan, Shuai Wang, Mingyue Jiang, and Tsong Yueh Chen. 2021. Perception matters: Detecting perception failures of VQA models using metamorphic testing. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR'21)*. 16908–16917.
[72] Jie Zhang, Junjie Chen, Dan Hao, Yingfei Xiong, Bing Xie, Lu Zhang, and Hong Mei. 2014. Search-based inference of polynomial metamorphic relations. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE'14)*. 701–712.
[73] Jie Ming Zhang, Mark Harman, Lei Ma, and Yang Liu. 2022. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering* 48, 1 (2022), 1–36.
[74] Lei Zhang, Bei-Bei Yin, Junpeng Lv, Kai-Yuan Cai, Stephen S. Yau, and Jia Yu. 2014. A history-based dynamic random software testing. In *Proceedings of the 38th International Computer Software and Applications Conference Workshops (COMPSACW'14)*. 31–36.

Received 6 August 2021; revised 25 February 2022; accepted 22 April 2022