Adaptive Partition Testing

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

Abstract—Random testing and partition testing are two major families of software testing techniques. They have been compared both theoretically and empirically in numerous studies for decades, and it has been widely acknowledged that they have their own advantages and disadvantages and that their innate characteristics are fairly complementary to each other. Some work has been conducted to develop advanced testing techniques through the integration of random testing and partition testing, attempting to preserve the advantages of both while minimizing their disadvantages. In this paper, we propose a new testing approach, *adaptive partition testing*, where test cases are randomly selected from some partition whose probability of being selected is adaptively adjusted along the testing process. We particularly develop two algorithms, *Markov-chain based adaptive partition testing* and *reward-punishment based adaptive partition testing*, to implement the proposed approach. The former algorithm makes use of Markov matrix to dynamically adjust the probability of a partition to be selected for conducting tests; while the latter is based on a reward and punishment mechanism. We conduct empirical studies to evaluate the performance of the proposed algorithms using ten faulty versions of three large-scale open source programs. Our experimental results show that, compared with two baseline techniques, namely Random Partition Testing (RPT) and Dynamic Random Testing (DRT), our algorithms deliver higher fault-detection effectiveness with lower test case selection overhead. It is demonstrated that the proposed adaptive partition testing is an effective testing approach, taking advantages of both random testing and partition testing.

Index Terms—Random testing, partition testing, adaptive partition testing.

1 Introduction

Software testing is a major approach to assessing and assuring the reliability of the software under development. In the past few decades, we have seen lots of testing techniques proposed and developed based on different intuitions and for serving various purposes. Among them, random testing (RT) [1] and partition testing (PT) [2] represent two fundamental families of testing techniques.

In RT, test cases are randomly selected from the input domain (which refers to the set of all possible inputs of the software under test). Traditionally, when the testing purpose is to detect software faults, the random test case selection is normally based on the uniform distribution, that is, each possible input has the same probability to be selected as a test case. On the other hand, if the testing purpose is the reliability assessment, RT follows a so-called operational profile, which refers to the probability distribution according to users' normal operations.

Different from RT, PT was originally proposed to generate test cases in a more "systematic" way, aiming at improving the effectiveness of fault detection. The input domain is first divided into disjoint partitions, and test cases are then selected from each and every partition. In PT, each partition is expected to have a certain degree of homogeneity, that is, inputs in the same partition should cause similar software execution behavior. In the ideal case, a partition should be homogeneous, that is, if one input is fault-revealing/non-fault-revealing, all other inputs in the same partition will be fault-revealing/non-fault-revealing too.

Since 1980s or even earlier, RT and PT have been compared with each other in terms of their fault detection effectiveness [3], [4], [5], [6]. It had been surprising to quite a few people that PT, considered as more systematic, does not outperform RT too much, and that in some circumstances, RT even has higher effectiveness than PT. Intuitively speaking, PT should be very effective if each partition is homogeneous. However, such a homogeneity cannot be guaranteed in practice, and hence PT may be ineffective. In contrast, RT selects test cases in a random manner, so it is possible that it misses some faults that could easily be revealed by PT; on the other hand, also due to its randomness, RT can sometimes reveal non-trivial faults that are difficult to be detected by PT.

Generally speaking, RT and PT are based on fundamentally different intuitions. Therefore, it is likely that they can be complementary to each other in detecting various types of faults. It is thus interesting to investigate the integration of them for developing new testing techniques. In fact, Cai et al. [7], [8] have proposed the so-called random partition testing (RPT). In RPT, the input domain is first divided into m partitions $c_1, c_2, \ldots, c_m$, and each c_i is allocated a probability p_i. A partition c_i is randomly selected according to the testing profile $\{p_1, p_2, \ldots, p_m\}$, where $p_1+p_2+\cdots+p_m = 1$. A concrete test case is then randomly selected from the chosen c_i.

Independent researchers from various areas have individually observed that the fault-revealing inputs tend to cluster into "continuous regions" [9], [10]. Based on this observation, Cai et al. [11] further proposed the so-called dynamic random testing (DRT) technique based on the the-

C.-A. Sun and H. Dai are with the School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing 100083, China. E-mail: casun@ustb.edu.cn.
H. Liu is with the College of Engineering and Science, Victoria University, Melbourne VIC 8001, Australia. E-mail: Huai.Liu@vu.edu.au.
T.Y. Chen is with the Department of Computer Science and Software Engineering, Swinburne University of Technology, Hawthorn VIC 3122, Australia. Email: tychen@swin.edu.au.
K.-Y. Cai is with the School of Automation Science and Electrical Engineering, Beihang University, Beijing 100191, China. E-mail: kycai@buaa.edu.cn

ory of software cybernetics [12]. Different from the original RPT, where the values of p_i are fixed and kept static along the testing process, DRT attempts to dynamically change the values of p_i: If a test case from a partition c_i reveals a fault, the corresponding p_i will be increased by a constant ϵ; otherwise, decreased by another constant δ. Although DRT is more effective than RPT, it also has its own problems, such as an appropriate assignment of values for (ϵ and δ), and the inefficiency in locating the partitions with higher fault-revealing probability.

In this paper, we propose a new testing approach, namely adaptive partition testing (APT), of integrating RT and PT. The key component of APT is a novel mechanism of adaptively changing the adjustments to p_i along the testing process. The major contributions of our study are as follows.

- We develop two algorithms for APT, namely Markov-chain based APT (MAPT) and reward-punishment based APT (RAPT), which, as their names manifest, implement the adaptive adjustment of p_i based on the Markov matrix and the reward and punishment mechanism, respectively.
- The performance of MAPT and RAPT is evaluated through a series of empirical studies on large-scale open source programs. It is shown that both MAPT and RAPT have significantly higher fault-detection effectiveness than RPT and DRT, while their test case selection overhead is lower.

The rest of the paper is organized as follows. In Section 2, we introduce the basic intuition of APT, and present the algorithms of MAPT and RAPT. In Section 3, we discuss the settings of empirical studies for evaluating MAPT and RAPT, the results of which are summarized in Section 4. The previous work closely related to APT is discussed in Section 5. Finally, we conclude the paper in Section 6.

2 Our Approach

2.1 Intuition

Let us first look at how DRT adjusts the value of p_i. Suppose that a test case from c_i ($i = 1, 2, \dots, m$) is selected and executed. If this test case reveals a fault, $\forall j = 1, 2, \dots, m$ and $j \neq i$, we set

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

and then set

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

Otherwise, that is, the test case does not reveal a fault, we set

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

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

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

As observed from Formulas 1 to 4, the adjustment of p_i is based on two preset constants ϵ and δ. Although some studies [13], [14], [15] have been conducted to give a guideline of optimizing ϵ and δ, it is almost impossible to have "golden" values of them across various situations. Thus, we are inspired to propose APT, which basically attempts to adjust the value of p_i adaptively to the online information of fault detection as well as the varying probability across different partitions. We develop two algorithms, namely MAPT and RAPT for implementing APT, as presented in the following two sections, respectively.

2.2 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 $p_{i,j} = Pr\{j|i\}$. If $i, j = 1, 2, \dots, m$, we can then construct a Markov matrix as follows:

$$P = \begin{pmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,m} \\ p_{2,1} & p_{2,2} & \cdots & p_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ p_{m,1} & p_{m,2} & \cdots & p_{m,m} \end{pmatrix} \tag{5}$$

Note that for any given i, $\sum_{j=1}^{m} p_{i,j} = 1$, that is, the sum of each row in the Markov matrix (Formula 5) is 1.

In MAPT, we consider each partition as a state in the Markov matrix. If a partition c_i is selected for conducting a test, the probability of selecting c_j for conducting the next test will be $p_{i,j}$. MAPT will adaptively adjust the value of each $p_{i,j}$ according to the following procedure.

Suppose that a test case from c_i is selected and executed. If this test case reveals a fault, $\forall j = 1, 2, \dots, m$ and $j \neq i$, we set

$$p'_{i,j} = \begin{cases} p_{i,j} - \dfrac{\gamma \times p_{i,i}}{m-1} & \text{if } p_{i,j} > \dfrac{\gamma \times p_{i,i}}{m-1} \\ p_{i,j} & \text{if } p_{i,j} \leq \dfrac{\gamma \times p_{i,i}}{m-1} \end{cases}, \tag{6}$$

and then we set

$$p'_{i,i} = 1 - \sum_{\substack{j=1 \\ j\neq i}}^{m} p'_{i,j}. \tag{7}$$

Otherwise, that is, the test case does not reveal a fault, $\forall j = 1, 2, \dots, m$ and $j \neq i$, we set

$$p'_{i,j} = \begin{cases} p_{i,j} + \dfrac{\tau \times p_{i,j}}{m-1} & \text{if } p_{i,i} > \dfrac{\tau \times (1-p_{i,i})}{m-1} \\ p_{i,j} & \text{if } p_{i,i} \leq \dfrac{\tau \times (1-p_{i,i})}{m-1} \end{cases}, \tag{8}$$

and then we have

$$p'_{i,i} = \begin{cases} p_{i,i} - \dfrac{\tau \times (1-p_{i,i})}{m-1} & \text{if } p_{i,i} > \dfrac{\tau \times (1-p_{i,i})}{m-1} \\ p_{i,i} & \text{if } p_{i,i} \leq \dfrac{\tau \times (1-p_{i,i})}{m-1} \end{cases}. \tag{9}$$

The details of MAPT is given in Algorithm 1. In MAPT, the first test case is selected from a partition that is randomly selected according to the initial probability profile $\{p_1, p_2, \ldots, p_m\}$ (Lines 5 and 6 in Algorithm 1). After the execution of a test case, the Markov matrix P will be updated through changing the values of $p_{i,j}$ (Line 13): If a fault is revealed, Formulas 6 and 7 will be used; otherwise, Formulas 8 and 9 will be used. The updated matrix will be used to guide the random selection of the next test case (Lines 8 and 9). Such a process is repeated until the termination condition is satisfied (refer to Line 3). The termination condition here can be either "testing resource has been exhausted", or "a certain number of test cases have been executed", or "a certain number of faults have been detected", etc. Note that after a fault is detected (Line 11), the testing process continues only when the termination condition is not satisfied (Line 3); otherwise, the testing process is stopped.

Algorithm 1 MAPT

Input: $\gamma, \tau, p_1, p_2, \ldots, p_m$
```
1:  Initialize Markov matrix P by setting p_{i,j} = p_j
2:  Set noTC = 0
3:  while termination condition is not satisfied
4:    if noTC = 0
5:      Select a partition c_i according to profile {p_1, p_2, ..., p_m}
6:      Select a test case t from c_i
7:    else
8:      Given that the previous test case is from c_i, select a
        partition c_j according to profile {p_{i,1}, p_{i,2}, ..., p_{i,m}}
9:      Select a test case t from c_j
10:   end_if
11:   Test the software using t
12:   Increment noTC by 1
13:   Update P based on the testing result according to
      Formulas 6 to 9
14: end_while
```

From Formulas 6 to 9, we can see that an update of P involves m simple calculations. In addition, the selection of c_i and the selection of a test case all need a small time. Simply speaking, the execution time for one iteration in MAPT mainly depends on the execution of a test case, in particular when program under test is a complex one. Therefore, the time complexity for MAPT to select n test cases is $O(m \cdot n)$.

2.3 RAPT

It is commonly observed that faults normally intend to cluster in some codes. This indicates that if a fault is detected by a test case in a partition c_i, other faults are likely to exist in the relevant codes and thus the execution of more tests from c_i will more likely lead to fault detection. Based on the reward and punishment mechanism, RAPT attempts to give a higher chances to selecting the fault-revealing 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 failure rate, and its corresponding p_i will become 0. Basically, the higher value Rew_i has, the larger p_i the partition c_i has. The p_i's adjustment mechanism of RAPT is as follows. Suppose that a test case from c_i is selected and executed. If this test case reveals a fault, $\forall j = 1, 2, \ldots, m$ and $j \neq i$, we set

$$p'_j = \begin{cases} p_j - \dfrac{(1+\ln Rew_i) \times \epsilon}{m-1} & \text{if } p_j \geq \dfrac{(1+\ln Rew_i) \times \epsilon}{m-1} \\ 0 & \text{if } p_j < \dfrac{(1+\ln Rew_i) \times \epsilon}{m-1} \end{cases}, \tag{10}$$

and then we have

$$p'_i = 1 - \sum_{\substack{j=1 \\ j \neq i}}^{m} p'_j. \tag{11}$$

Otherwise, that is, the test case does not reveal a fault, we set

$$p'_i = \begin{cases} p_i - \delta & \text{if } p_i \geq \delta \\ 0 & \text{if } p_i < \delta \text{ or } Pun_i = Bou_i \end{cases}, \tag{12}$$

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

$$p'_j = \begin{cases} p_j + \dfrac{\delta}{m-1} & \text{if } p_i \geq \delta \\ p_j + \dfrac{p_i}{m-1} & \text{if } p_i < \delta \text{ or } Pun_i = Bou_i \end{cases}. \tag{13}$$

The details of RAPT is given in Algorithm 2. Like MAPT, RAPT selects the first test case from a partition that is randomly selected according to the initial profile $\{p_1, p_2, \ldots, p_m\}$ (Lines 5 and 6 in Algorithm 2). If a test case reveals a fault, the same partition will be used for selecting the next test case until a non-fault-revealing test case is selected (refer to the while loop from Lines 7 to 12). Otherwise, the probability profile for p_i will be updated according to Formulas 10 to 13 (Lines 14 to 19). Note that the values of Rew_i and Pun_i are adaptively adjusted during the testing process. After a fault is detected (Line 11), the testing process continues only when the termination condition is not satisfied (Line 3); otherwise, the testing process is stopped.

Similar to MAPT, RAPT only requires a small time for updating the probability profile, selecting the partition, and selecting test cases. Hence, the time complexity of RAPT is also $O(m \cdot n)$ for selecting n test cases.

2.4 Example

We use an explanatory example to illustrate the proposed technique. A Baggage Billing system (BB) enables airline counter staff to decide whether passengers need to pay the

Algorithm 2 RAPT

Input: $\epsilon, \delta, p_1, p_2, \ldots, p_m, Bou_1, Bou_2, \ldots, Bou_m$

```
Initialize Rew_i = 0 and Pun_i = 0 for all i = 1, 2, ..., m
Set noTC = 0
while termination condition is not satisfied
  Select a partition c_i according to the profile {p_1, p_2, ..., p_m}
  Select a test case t from c_i
  Test the software using t
  while a fault is revealed by t
    Increment Rew_i by 1
    Set Pun_i = 0
    Select a test case t from c_i
    Test the software using t
  end_while
  Increment Pun_i by 1
  if Rew_i ≠ 0
    Update p_j (j = 1, 2, ..., m and j ≠ i) and p_i
    according to Formulas 10 and 11, respectively
    Set Rew_i = 0
  else
    Update p_i and p_j (j = 1, 2, ..., m and j ≠ i)
    according to Formulas 12 and 13, respectively
  end_if
end_while
```

TABLE 1
Categories and choices of BB

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

baggage fees and how much to pay. BB performs the calculation based on several factors such as aircraft cabin (*Economy*, *Business*, and *First Class*), flight region (*International* and *Domestic*), and baggage weight (*Above Limit* and *Below Limit*). If the baggage weight is below the limit, there is not an excess baggage charge; otherwise, passengers will be charged for overweight according to the baggage accounting standard.

Given the above specification, we employ the category-partition method (CPM) [16] to partition the input domain (More details on the exercise of CPM will be discussed in Section 3.4.1). As a consequence, we identify categories and associated choices of BB, as shown in Table 1. With these categories/choices and constraints among choices, we further derive a set of *complete test frames* and each of them corresponds to a partition. Note that the number of partitions may vary with the granularity level. To ease the illustration, we derive partitions without considering the baggage weight category. Table 2 shows the resulting partitions for BB, where $weight_*$ indicates no classification on the baggage weight.

We now apply MAPT and RAPT to testing BB. First, we set the parameters of MAPT and RAPT (More details on the parameter settings will be discussed in Section 3.4). As an illustration, we simply set $\gamma = \epsilon = 0.1$, and $\tau = \delta = 0.01$. As for $Bou_i (i = 1, \ldots, 6)$ of RAPT, their values should be set according to the testing adequacy. As a technical guideline,

TABLE 2
Partitions of BB

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

if the program has not been adequately tested (i.e. it is easy to detect faults), we set $Bou_i = 10\% \times k_i$, where k_i refers to the number of test cases belonging to partition c_i; Otherwise, we set $Bou_i = 70\% \times k_i$ or even bigger. For this example, we set $Bou_i = 2$ and assume that only partition c_1 has faults.

We record the execution information of each test case. Given a partition c_i and its test case t_k, the execution information is represented by a 3-tuple $\langle c_i, t_k, flag \rangle$, where $flag$ can be either $true$ or $false$ indicating whether t_k detects a fault or not, respectively. Assume the execution information of the first nine test cases during a test is as follow: $\langle c_1, t_1, true \rangle$, $\langle c_1, t_2, true \rangle$, $\langle c_1, t_3, false \rangle$, $\langle c_2, t_4, false \rangle$, $\langle c_3, t_5, false \rangle$, $\langle c_6, t_6, false \rangle$, $\langle c_4, t_7, false \rangle$, $\langle c_5, t_8, false \rangle$, and $\langle c_2, t_9, false \rangle$.

Let us illustrate how the probability profiles are updated with these nine test cases in MAPT and RAPT. In the context of MAPT, the steps of Algorithm 1 are followed. Assume an equal initial profile $p = \{ \langle c_1, 0.1667 \rangle, \langle c_2, 0.1667 \rangle, \langle c_3, 0.1667 \rangle, \langle c_4, 0.1667 \rangle, \langle c_5, 0.1667 \rangle, \langle c_6, 0.1667 \rangle \}$, and the initial Markov matrix as follows:

$$P = \begin{pmatrix} 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 \\ 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 \\ 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 \\ 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 \\ 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 \\ 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 & 0.1667 \end{pmatrix} \tag{14}$$

Now consider the process of updating Markov matrix. Assume we first select a test case in partition c_1 according to p, then $P_{1,j}$ $(j = 1...6)$ is updated using Formulas 6 and 7, since a fault is revealed. Next, c_1 is selected according to the updated $P_{1,j}$ (the first line of P), which refers to the transition probabilities from c_1 to c_j. In this way, P is updated once after each case execution. After the ninth test case is executed, $P_{6,j}$ (the 6th line of P) is updated accordingly and the current Markov matrix is as follows:

$$P = \begin{pmatrix} 0.1984 & 0.1605 & 0.1605 & 0.1605 & 0.1605 & 0.1605 \\ 0.1673 & 0.1634 & 0.1673 & 0.1673 & 0.1673 & 0.1673 \\ 0.1670 & 0.1670 & 0.1650 & 0.1670 & 0.1670 & 0.1670 \\ 0.1670 & 0.1670 & 0.1670 & 0.1650 & 0.1670 & 0.1670 \\ 0.1670 & 0.1670 & 0.1670 & 0.1670 & 0.1650 & 0.1670 \\ 0.1670 & 0.1670 & 0.1670 & 0.1670 & 0.1670 & 0.1650 \end{pmatrix} \tag{15}$$

Let us look at RAPT where the steps of Algorithm 2 are followed. Again, we have an equal initial profile $p = \{\langle c_1, 0.1667 \rangle, \langle c_2, 0.1667 \rangle, \langle c_3, 0.1667 \rangle, \langle c_4, 0.1667 \rangle, \langle c_5, 0.1667 \rangle, \langle c_6, 0.1667 \rangle\}$, and set $Pun_i = 0$ and $Rew_i = 0$ where $i \in \{1, \ldots, 6\}$.

Recall the execution information of the first nine test cases. Since t_1 in c_1 reveals a fault, we set $Rew_1 = Rew_1+1 = 1$ and $Pun_1 = 0$, and the profile p is updated using Formulas 10 and 11. Similarly, t_2 in c_1 reveals a fault, we set $Rew_1 = Rew_1 + 1 = 2$, and p is updated using Formulas 10 and 11. Since t_3 in c_1 does not reveal a fault, we set $Pun_1 = Pun_1 + 1 = 1$ and $Rew_1 = 0$, and update p using Formulas 12 and 13. Accordingly, we have $p = \{\langle c_1, 0.3360\rangle, \langle c_2, 0.1328\rangle, \langle c_3, 0.1328\rangle, \langle c_4, 0.1328\rangle, \langle c_5, 0.1328\rangle, \langle c_6, 0.1328\rangle\}$. In this way, partition c_i is selected according to p which is updated once after each test case execution. After the ninth test case is executed, Pun_2 reaches the preset bound value Bou_2 (because t_4 and t_9 in c_2 do not reveal a fault), and we have $p = \{\langle c_1, 0.3721\rangle, \langle c_2, 0\rangle, \langle c_3, 0.1569\rangle, \langle c_4, 0.1569\rangle, \langle c_5, 0.1569\rangle, \langle c_6, 0.1569\rangle\}$.

3 Empirical Studies

We have conducted a series of empirical studies to evaluate the performance of MAPT and RAPT. The design of experiments is described in this section.

3.1 Research questions

In our experiments, we focus on the following two research questions:

RQ1 How effective are MAPT and RAPT in detecting software faults?
The fault-detection effectiveness is one key criterion to evaluate the performance of a testing technique. In this study, we examined how effectively MAPT and RAPT detected distinct faults in real-life programs.

RQ2 What is the actual test case selection overhead for each of the two APT algorithms?
In Section 2, we have theoretically demonstrated that both MAPT and RAPT only require linear time for generating test cases. We also wish to empirically evaluate the test case selection overhead of them in detecting software faults.

3.2 Object programs

In our study, we selected real-life programs from the software artifact infrastructure repository (SIR) [17] as the objects of the experiments. Among object programs in SIR, we only included those that are written in C, whose sizes are larger than 5K LOC in order to make the evaluation as closer as possible to the real software testing, and that are associated with test suites generated using partition testing techniques. Accordingly, we have selected `grep`, `gzip`, and `make` as object programs. Furthermore, these selected object programs have different versions, which manifests practical scenarios of software development. Table 3 summarizes the basic information of the object programs, giving the name of object program (*Program*), the size of the object program (*LOC*), the number of test cases in the associated test repository (*Size of test suite*), the concerned versions of the object program (*Version ID*), and the number of faults included for evaluation (*Number of faults*).

TABLE 3
Object programs

Program	LOC	Size of test suite	Version ID	Number of faults
`grep`	10,068	470	V1	2
			V2	2
			V3	2
			V4	1
`gzip`	5,680	214	V1	3
			V2	1
			V4	2
			V5	3
`make`	35,545	793	V1	2
			V2	1

As to the selection of faults, we only include those that are hard to detect for evaluation; in more detail, we checked all faults in the repository and excluded those faults whose detections require less then 20 randomly selected test cases, except that the detection of the fault in V2 of ”gzip” requires about 15 randomly selected test cases. Note that the fault in V3 of `gzip` cannot be revealed by any test cases in the associated test suite, hence it was not included in our experiments. In summary, the empirical studies were conducted on three object programs with ten faulty versions and 19 distinct faults, namely 3 faults for `make`, 7 faults for `grep`, and 9 faults for `gzip`.

3.3 Variables

3.3.1 Independent variables

The independent variable in our study is the testing technique. The two APT algorithms, MAPT and RAPT, are the independent variables. In addition, we selected RPT and DRT as the baseline techniques for comparison.

3.3.2 Dependent variables

In this study, we consider two testing scenarios: (i) a fault is detected, the testing process is paused and debugging is started, and (ii) when a fault is detected, the testing process continues without a pause. The dependent variable for RQ1 is the metric for evaluating the fault-detection effectiveness. There exist quite a few effectiveness metrics, such as the P-measure (the probability of at least one fault being detected by a test suite), the E-measure (the expected number of faults being detected by a test suite), and the F-measure (the expected number of test cases to detect the first fault). Among them, the F-measure is the most appropriate metric for measuring the fault-detection effectiveness of adaptive testing techniques in the first testing scenario, such as MAPT, RAPT, and DRT, in which the generation of new test cases is adaptive to the previous testing process. In our study, we use F_{method} to represent the F-measure of a testing method, where $method$ can be MAPT, RAPT, DRT, and RPT.

As shown in Algorithms 1 and 2, the testing process may not be terminated after the detection of the first fault. In addition, the fault detection information can lead to different probability profile adjustment mechanisms. Therefore, it is also important to see what would happen after the first fault is revealed. In our study, we introduce a new metric F2-measure, which refers to how many additional test cases are required to reveal the second fault after the detection of

TABLE 4
Number of partitions of each object program

Program	Coarse	Medium	Fine
grep	3	9	13
gzip	4	6	12
make	4	8	16

the first fault in the second testing scenario. Note that F2-measure can be further generalized to measure the number of test cases required for detecting the $i+1^{th}$ fault in the context of having the i^{th} fault being detected in an iterative way. Similarly, we use $F2_{method}$ to represent the F2-measure of a testing method.

For RQ2, an obvious metric is the time required on detecting faults. In this study, corresponding to the F-measure and F2-measure, we used the T-measure and T2-measure to measure the time for detecting the first fault and the extra time for detecting the second fault, respectively. Similarly, T_{method} and $T2_{method}$ are used to denote these time metrics.

For each of the above four metrics, a smaller value intuitively implies a better performance.

3.4 Experimental settings

3.4.1 Partitioning

Partition testing is a mainstream family of software testing technique, which can be realized in different ways, such as *Intuitive Similarity*, *Equivalent Paths*, *Risk-Based*, and *Specified As Equivalent* (two test values are equivalent if the specification says that the program handles them in the same way). In our study, we made use of the category-partition method (CPM) [16], a typical PT technique based on *Specified As Equivalent*, to conduct the partitioning. In CPM, a functional requirement is decomposed into a set of categories, which characterize input parameters and critical environmental conditions. Each category is further divided into disjoint partitions, namely choices, which refer to the values or value ranges that the category can take. CPM constructs test frames, each of which is a valid combination of choices. A test case can be generated by allocating concrete value to each choice in a test frame. To investigate the performance of our techniques under various scenarios, we applied the following three levels of granularity in partitioning:

- **Coarse**: Select only one category, and partition the input domain according to its choices.
- **Medium**: Select two categories that have fewer choices annotated with [single] or [error], and partition the input domain according to the combinations of their choices.
- **Fine**: Consider all categories, and partition the input domain according the combinations of their choices.

Note that the reason why we preferred not to select those categories containing lots of choices annotated with [single] or [error] is that this kind of choices cannot be combined with choices of other categories. The numbers of partitions on each granularity of every object program are reported in Table 4.

3.4.2 Initial probability profile

In the experiments, we made use of two types of initial probability profiles, namely *equal* and *proportional*. In the equal initial probability profile, $p_1 = p_2 = \cdots = p_m = \dfrac{1}{m}$. In the proportional initial probability profile, $p_i = \dfrac{k_i}{k}$, where k is the total number of test cases in the test suite, and k_i is number of the test cases inside c_i.

3.4.3 Parameters

Previous studies [13], [14], [15] have given some guidelines on how to set ϵ and δ for DRT. We followed these studies to set $\epsilon = 0.05$ and calculate the proper δ according to the following formula (which is extracted from [15]):

$$\frac{1}{\theta_M} - 1 < \frac{\epsilon}{\delta} < \frac{1}{\theta_\Delta - 1} - 1, \tag{16}$$

where θ_M and θ_Δ are the largest and the second largest failure rates inside each partition, respectively. Note that the value of δ was different for different scenarios. Hereby, a scenario refers to an instance of one particular faulty version with a certain granularity level and a specific initial probability profile (for example, "grep v1" with "coarse granularity" and "proportional probability profile").

To have a fair comparison, we set RAPT to have the same values of ϵ and δ as DRT.

To set the values of γ and τ for MAPT, 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 dramatic or very marginal. Both situations might result in the "unfair" (either too big or too small) award/punishment to certain partitions. After several rounds of trials, we concluded that $\gamma = \tau = 0.1$ were the fair settings. 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.

3.5 Experimental environment

Our experiments were conducted on a virtual machine running the Ubuntu 11.06 64-bit operating system. In this system, there were two CPUs and a memory of 2GB. The test scripts were generated using Java and bash shell. In the experiments, we repeatedly run the testing using each technique 20 times with different random seeds to guarantee the statistically reliable mean values of the metrics (F-measure, F2-measure, T-measure, and T2-measure).

3.6 Threat to validity

Although we have carefully designed the empirical studies, some factors that might have influenced our results are summarized as follows.

3.6.1 Internal validity

The threat to internal validity is related to the implementations of the testing techniques, which involved a moderate amount of programming work. The code was also cross-checked by different individuals. We are confident that all techniques were correctly implemented.

3.6.2 External validity

One obvious threat to external validity is that we only considered three object programs. However, they are real-life programs with fairly large size, and they have been popularly used in many studies. In addition, 19 distinct faults were used to evaluate the performance. Though we have tried to improve the generality by applying different granularities in partitioning and two types of initial profiles, we cannot say whether or not similar results would be observed on other types of programs. In addition, the identification of categories and choices in CPM is a subjective process, which may affect the external validity of our study. A comprehensive evaluation should include different kinds of software, the cost of developing such a comprehensive benchmark prohibits us from doing so.

3.6.3 Construct validity

The metrics used in our study are simple in concept and straightforward to apply. They are also the commonly used metrics by the community. Hence, the threat to construct validity is little.

3.6.4 Conclusion validity

We have run a sufficiently large number of trials to guarantee the statistical reliability of our experimental results. In addition, as to be discussed in Section 4, statistical tests were conducted to verify the significance of our results.

4 EXPERIMENTAL RESULTS

4.1 RQ1: Fault-detection effectiveness

The detailed experimental results of F-measure and F2-measure are given in Tables A1 to A3 in the Appendix. The distributions of F-measure and F2-measure on each object program are displayed by the boxplots in Figures 1 and 2. In the boxplot, the upper and lower bounds of the box represent the third and first quartiles of a metric, respectively, and the middle line denotes the median value. The upper and lower whiskers respectively indicate the largest and smallest data within the range of $\pm 1.5 \times IQR$, where IQR is the interquartile range. The outliers outside IQR are denoted by hollow circles. The solid circle represents the mean value of a metric.

From Figures 1 and 2, we can observe that in general, RAPT was the best performer in terms of F-measure and F2-measure, followed by MAPT, DRT, and RPT in descending order. We further conducted hypothesis testing to verify the statistical significance of this observation. We used the Holm-Bonferroni method [18] to determine which pair of testing techniques have significant difference in terms of each metric. Across the whole study, for each pair of testing techniques, denoted by technique a and technique b, the null hypothesis (H_0) was that a and b had the similar performance in terms of one metric. All the null hypotheses were ordered by their corresponding p-values, from lowest to largest; in other words, for null hypotheses H_0^1, H_0^2, $\ldots$, we had $p_1 \leq p_2 \leq \ldots$. For the given confidence level $\alpha = 0.05$, we found the minimal index h such that $p_h > \dfrac{\alpha}{N+1-h}$ (where N is the total number of null hypotheses). Then, we rejected H_0^1, H_0^2, $\ldots$, H_0^{h-1}, that is, we regarded the pair of techniques involved in each of these hypotheses to have statistically significant difference in terms of a certain metric. On the other hand, we considered the pair of techniques involves in each of H_0^h, H_0^{h+1}, $\ldots$ not to have significant difference with respect to one metric, as these hypotheses were accepted. The statistical testing results are shown in Tables 5 and 6. Each cell in the tables gives the number of scenarios where the technique on the top row performed better than that on the left column. If the difference is significant, the number will be displayed in bold. For example, the "59" in the top right corner in Table 5 indicates that out of 60 scenarios (10 faulty versions $\times$ 3 granularities $\times$ 2 initial profiles), RAPT had smaller F-measure than RPT for 59 scenarios. Correspondingly, the "1" in the bottom left corner in Table 5 means that the F-measure of RPT was smaller than that of RAPT for only one scenario. The corresponding null hypothesis was that RAPT and RPT had similar performance in terms of F-measure. Since this hypothesis was rejected, we could say that the fault-detection capabilities of RAPT and RPT were significantly different in terms of F-measure. This statistically significance difference is represented by the bold font of "59" and "1", which further indicates that RAPT was significantly better than RPT. Note that although two numbers ("59" and "1") are given, they are corresponding to one null hypothesis — each pair of numbers just further telling us the number of scenarios where one technique was better than the other technique.

TABLE 5
Number of scenarios where the technique on top row has smaller F-measure than that on left column out of 60 scenarios

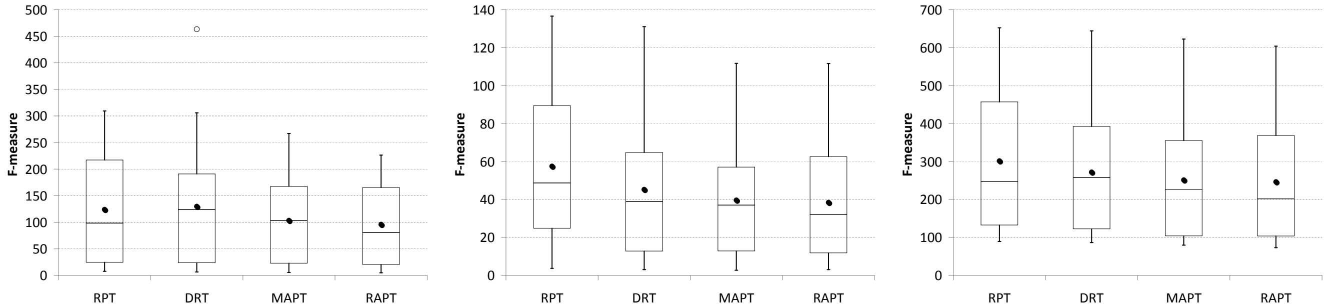
	RPT	DRT	MAPT	RAPT
RPT	—	**49**	**59**	**59**
DRT	**11**	—	**56**	**55**
MAPT	**1**	**4**	—	**40**
RAPT	**1**	**5**	**20**	—

TABLE 6
Number of scenarios where the technique on top row has smaller F2-measure than that on left column out of 60 scenarios

	RPT	DRT	MAPT	RAPT
RPT	—	**31**	**36**	**36**
DRT	**11**	—	**33**	**38**
MAPT	**6**	**9**	—	**33**
RAPT	**6**	**4**	**9**	—

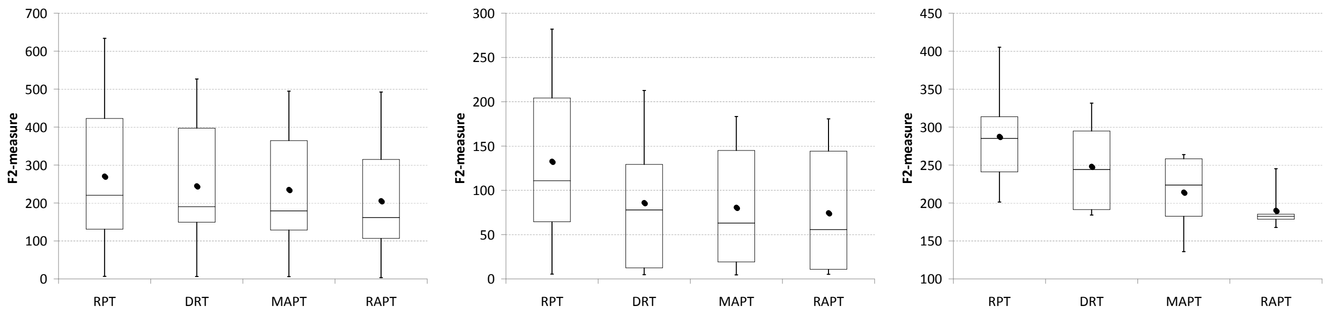
All the entries in Tables 5 and 6 are in italic font. Such observations imply that the effectiveness difference between each pair of testing techniques was always statistically significant. In summary, RAPT was significantly better than MAPT, DRT, and RPT, in terms of both F-measure and F2-measure; similarly, MART was significantly better than DRT and RPT, while DRT was significantly better than RPT.

In additional to the hypothesis testing, we also calculated the effect size to measure the magnitude of the performance difference between each pair of techniques. Because neither the F-measure nor F2-measure is normally distributed, it is not suitable to implement the conventional calculation of effect size based on the mean value and standard deviation of these measures. Instead, we make use of the "performance improvement ratio". For example, on V1 of `grep`



(a) grep (b) gzip (c) make

Fig. 1. Boxplots of F-measures on each object program



(a) grep (b) gzip (c) make

Fig. 2. Boxplots of F2-measures on each object program

TABLE 7
Effect size for the F-measure difference

Pair of techniques	Effect size
DRT vs. RPT	0.6249
MAPT vs. RPT	1.3500
RAPT vs. RPT	1.5234
MAPT vs. DRT	0.5749
RAPT vs. DRT	0.8134
RAPT vs. MAPT	0.3258

TABLE 8
Effect size for the F2-measure difference

Pair of techniques	Effect size
DRT vs. RPT	0.6221
MAPT vs. RPT	0.8956
RAPT vs. RPT	1.1822
MAPT vs. DRT	0.1889
RAPT vs. DRT	0.4769
RAPT vs. MAPT	0.3119

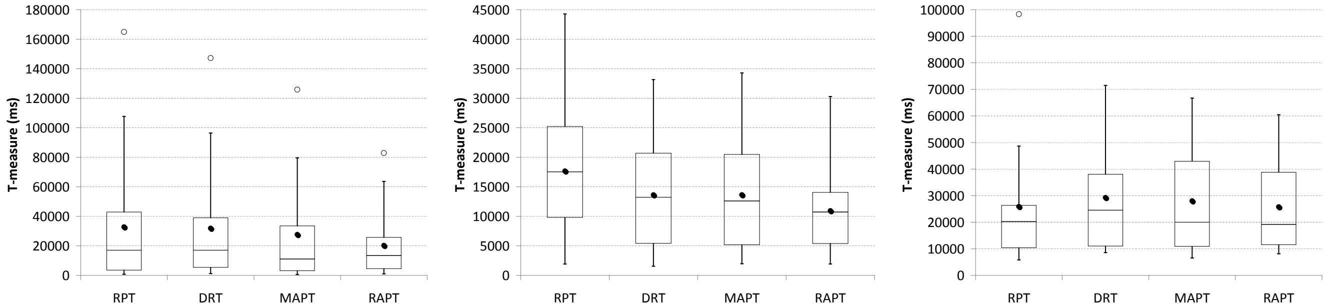
with coarse granularity and equal profile, the F-measures of RAPT and RPT are 159.10 and 231.70, so the performance improvement ratio of RAPT over RPT in this scenario is $\frac{231.70-159.10}{231.70} = 0.3133$. A standard formula $\frac{|\mu_1-\mu_2|}{\sigma}$ [19] was used to calculate the effect size, where μ_1 and μ_2 are the mean values of two sets of data, and σ is the standard deviation of both data sets. Tables 7 and 8 summarize the effect size that shows the magnitude of the difference between two techniques in terms of F-measure and F2-measure, respectively.

Based on the data given in Tables 7 and 8, we can further measure how large/small the performance difference is according to some rules of thumb [20]. In terms of F-measure, we can consider the difference between MAPT/RAPT and RPT as "very large" (effect size > 1.2); the difference between RAPT and DRT was "large" (effect size > 0.8); the difference was "medium" for DRT vs. RPT and MAPT vs. DRT (effect size > 0.5); whereas RAPT and MAPT only had "small" difference (effect size > 0.2). For F2-measure, there are two pairs of techniques with large difference (RAPT vs. RPT and MAPT vs. RPT), one pair with medium difference (DRT vs. RPT), two pairs with small difference (RAPT vs. DRT and RAPT vs. MAPT), and one with only "very small" differnce (MAPT vs. DRT, where effect size > 0.01).

4.2 RQ2: Selection overhead

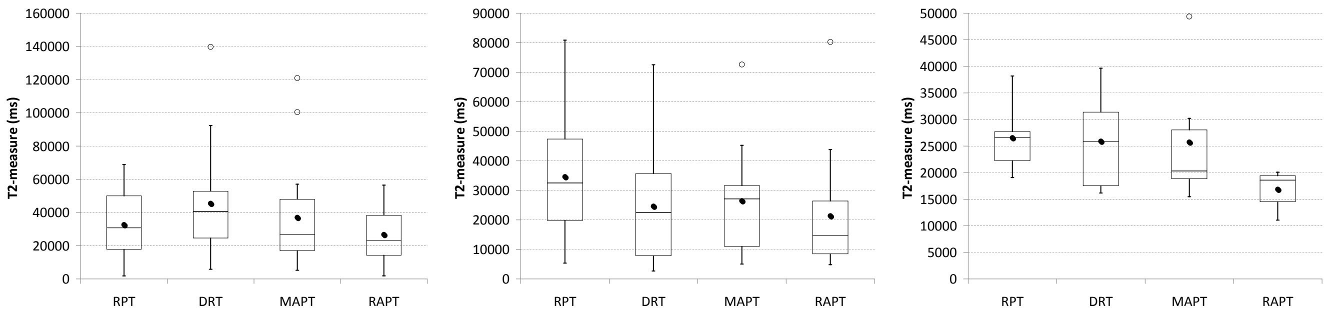
The detailed experimental results of T-measure and T2-measure are given in Tables A4 to A6 in the Appendix, and their distributions on each object program are plotted in Figures 3 and 4. From Figures 3 and 4, we can observe that in general, RAPT had the best performance, while MAPT just marginally outperformed DRT and RPT in terms of T-measure and T2-measure. Note that RAPT consistently delivered the best performance for all object programs in terms of both T-measure and T2-measure, while the other testing techniques fluctuate in performance according to object programs. For instance, DRT outperformed RPT for `gzip` in terms of both T-measure and T2-measure, while the latter outperformed the former in terms of T-measure for `make`.

We also conducted the hypothesis testing to determine which pair of testing techniques have significant difference



(a) grep (b) gzip (c) make

Fig. 3. Boxplots of T-measures on each object program



(a) grep (b) gzip (c) make

Fig. 4. Boxplots of T2-measures on each object program

TABLE 9
Number of scenarios where the technique on top row has smaller T-measure than that on left column out of 60 scenarios

	RPT	DRT	MAPT	RAPT
RPT	—	30	36	**41**
DRT	30	—	36	**41**
MAPT	24	24	—	**39**
RAPT	**19**	**19**	**21**	—

TABLE 10
Number of scenarios where the technique on top row has smaller T2-measure than that on left column out of 60 scenarios

	RPT	DRT	MAPT	RAPT
RPT	—	24	**30**	**33**
DRT	18	—	22	**29**
MAPT	**12**	20	—	**32**
RAPT	**9**	**13**	**10**	—

TABLE 11
Effect size for the T-measure difference

Pair of techniques	Effect size
DRT vs. RPT	0.2140
MAPT vs. RPT	0.2911
RAPT vs. RPT	0.4715
MAPT vs. DRT	0.3351
RAPT vs. DRT	0.4522
RAPT vs. MAPT	0.1793

TABLE 12
Effect size for the T2-measure difference

Pair of techniques	Effect size
DRT vs. RPT	0.1306
MAPT vs. RPT	0.2685
RAPT vs. RPT	0.6961
MAPT vs. DRT	0.1368
RAPT vs. DRT	0.5205
RAPT vs. MAPT	0.4640

in terms of T-measure and T2-measure, as shown in Tables 9 and 10.

From Table 9, we can observe that six entries ("30" & "30" for DRT vs. RPT, "36" & "24" for MAPT vs RPT, and "36" & "24" for MAPT vs DRT) are not in the bold font. These observations imply that in terms of T-measure, the performances of DRT and RPT were not distinguishable, and MAPT only marginally outperformed DRT and RPT. On the other hand, the performance of RAPT was significantly better than that of the other three techniques. Similar observation was made on the results of T2-measure, except that MAPT had significantly better T2-measure than RPT (whereas the T-measure difference between these two was marginal). In other words, the additional computation introduced in RAPT and MAPT for updating probability profiles is compensated with the saving of test executions.

Similar to the F-measure and F2-measure, we also calculated the effect size to evaluate the magnitude of the performance difference between each pair of techniques in terms of T-meaure and T2-measure, as summarized in Table 11 and 12, respectively.

With respect to T-measure, one pair of techniques (RAPT vs. MAPT) had very small difference, while other five pairs had small difference. In terms of T2-measure, two pairs (DRT vs. RPT and MAPT vs. DRT) had very small difference,

TABLE 13
Number of scenarios where the granularity level on top row is associated with smaller measure than that on left column

Comparison for F-measure

	Coarse	Medium	Fine
Coarse	—	36	32
Medium	44	—	**28**
Fine	48	**52**	—

Comparison for F2-measure

	Coarse	Medium	Fine
Coarse	—	25	19
Medium	31	—	**17**
Fine	37	**39**	—

Comparison for T-measure

	Coarse	Medium	Fine
Coarse	—	32	**27**
Medium	48	—	33
Fine	**53**	47	—

Comparison for T2-measure

	Coarse	Medium	Fine
Coarse	—	29	21
Medium	27	—	19
Fine	35	37	—

two pairs (MAPT vs. RPT and RAPT vs. MAPT) had small difference, and two pairs (RAPT vs. RPT and RAPT vs. DRT) had medium difference.

4.3 Further discussion

In our experiments, we have used three levels of granularity for partitioning as well as two types of initial probability profiles. We conducted further investigations to see whether there exist some strong correlations between the granularity/profile and the performance of APT.

4.3.1 Effect of granularity in partitioning

We compared each pair of granularity levels and conducted the hypothesis testing on the comparison, which is widely used to verify whether the performance difference between the testing techniques under study is statistically significant. The comparison results are summarized in Table 13. In most cases, the effects of different granularity level on APT's performance were not significantly different. In other words, we could not observe a strong correlation between the granularity level in partitioning and the performance of APT.

4.3.2 Effect of initial probability profile

We also compared the equal and proportional initial probability profiles in terms of their associated performance metrics. The comparison results are summarized in Table 14. It was shown that the proportional initial probability profile was significantly better than the equal one in terms of F2-measure, but the ranking was reversed when considering the T-measure. With respect to F-measure and T2-measure, there was no significant difference between these two types of initial probability profiles. In a word, we could not observe a strong correlation between the probability profile type and APT's performance.

TABLE 14
Number of scenarios where the initial profile type on top row is associated with smaller measure than that on left column

Comparison for F-measure

	Equal	Proportional
Equal	—	67
Proportional	53	—

Comparison for F2-measure

	Equal	Proportional
Equal	—	**28**
Proportional	**56**	—

Comparison for T-measure

	Equal	Proportional
Equal	—	**91**
Proportional	**29**	—

Comparison for T2-measure

	Equal	Proportional
Equal	—	51
Proportional	33	—

4.4 Summary

Through the evaluation, we have the following observations:

- RAPT demonstrates the best performance for all the three object programs with the four metrics. On the other hand, RAPT's outperformance varies with object programs as well as metrics. For instance, RAPT significantly outperformed RPT in terms of F2-measure and T2-measure for `make`, while such outperformance became slight in terms of in terms of F-measure and T-measure.
- MAPT demonstrates a better performance than RPT and DRT in terms of F1-measure and F2-measure for all the three object programs, while such outperformance was only marginal in terms of T-measure. An exceptional case is that MAPT was slightly worse than DRT in terms of T2-measure for `gzip`.

In summary, RAPT was consistently the best testing technique across all four metrics. MAPT was significantly better than DRT and RPT in terms of F-measure and F2-measure, but its outperformance was only marginal in terms of T-measure. This further indicates that among the proposed APT techniques, RAPT should be used.

5 Related Work

In this section, we will discuss the related work from the following four perspectives: (1) the integration of RT and PT, (2) the hypothesis of similarity among neighboring inputs, (3) the dynamic adjustment of test profile, and (4) the different treatments of fault-revealing and non-fault-revealing test cases.

It is not a new idea to integrate RT and PT. RPT [7], [8] may be the most straightforward technique for the integration, which selects a partition according to a pre-defined probability profile and selects test case from the partition. Cai [12] also proposed adaptive testing (AT) based on the concept of software cybernetics. Lv et al. [21] proposed an efficient AT strategy named Adaptive Testing with Gradient Descent method (AT-GD). Although AT outperformed both RT and PT in terms of fault-detection effectiveness, it requires very long execution time in reality. To address this efficiency problem, Lv et al. [22] proposed a hybrid approach that uses AT and random partition testing (RP-T) in an alternating manner. Furthermore, Cai et al. [11] proposed DRT, where the probability profile is dynamically adjusted according to the testing information. DRT has been applied into others fields such as Web services [23]. Since DRT makes use of some parameters for the probability profile adjustment, some studies [13], [14], [15] have been conducted to investigate the appropriate values for these parameters. However, the setting of these parameters may vary with different types of software. Compared with these techniques, our APT approach applies the novel mechanisms, such as MAPT and RAPT, to adaptively adjust the probability profile according to a variety of factors, which, in turn, are dynamically updated along the testing process.

The proposal of APT is inspired by the common observation that fault-revealing inputs tend to cluster into continuous regions inside the input domain [24], [9], [10], [25], [26]. Adaptive random testing (ART) [27], a family of enhance RT techniques, is also based on this observation. The basic notion of ART is "even spread": Because of the similarity of neighboring inputs, it is better to select a test case that is far away from the previous non-fault-revealing test cases. Compared with APT, ART mainly focuses on enhancing the effectiveness of RT by evenly spreading test cases across the input domain; in other words, it does not consider PT. Recently, CPM was used to help improve the applicability of ART into programs with non-numeric inputs [28] — the concepts of categories and choices were applied to provide a metric for measuring the dissimilarity between non-numeric test cases. APT and ART are based on the same hypothesis of similar behavior for neighboring inputs, but they attempt to improve the testing effectiveness from different perspectives.

There exist other studies that also involve the dynamic adjustment of test profile. Chen et al. [29], for example, proposed that the test profile should be dynamically revised according to the distribution of previously executed test cases. Similar work was further conducted by Liu et al. [30], where dynamic test profiles were designed to simulate various ART algorithms. The profiles investigated in these studies are actually the probability distribution of concrete test cases, while the profile in APT is the distribution for partitions.

One key issue of APT (and DRT) is that the profile will be adjusted in almost opposite ways depending on whether or not a test case reveals a fault. Such an arrangement is understandable because fault-revealing and non-fault-revealing test cases shall deliver quite different information. However, this issue was not considered in some techniques. For instance, quasi-random testing [31], [32] attempts to achieve the even spread of test cases by leveraging the features of low discrepancy and low dispersion of quasi-random sequences. Its test case selection process is not affected by the fault detection information at all. Liu et al. [33] studied the influence of fault-revealing test cases on the performance of ART, and found that if the fault-revealing test case was completely "forgotten" when selecting the next test case, ART would have better effectiveness in detecting multiple faults. Apparently, such a forgetting strategy is somewhat naive and not as fine-grained as the profile adjustment mechanisms used in MAPT and RAPT. In addition, Zhou et al. [34] proposed a technique to select test cases based on the online fault-detection information: The fault-revealing test cases will have high probability to be selected for the testing. This technique actually adjusted the selection of concrete test cases, while the adjustment mechanisms in our APT approach are conducted on the selection of partitions instead.

6 Conclusion

RT and PT are two fundamental families of software testing techniques. Each of them has its own merit and weakness, and sometimes they can be complementary to each other. Some studies have been conducted to integrate RT and PT, resulting in some advanced testing techniques, such as RPT and DRT. In this paper, we proposed a new approach to integrating RT and PT, namely APT. In APT, a test profile is maintained and adaptively updated based on some novel mechanisms, including the Markov matrix (MAPT) and the reward and punishment strategy (RAPT). Empirical studies have been conducted to evaluate the performance of MAPT and RAPT using three large-sized real-life programs associated with ten faulty versions and 19 distinct faults. It has shown that both RAPT and MAPT can use significantly fewer test cases than DRT and RPT. We have also observed that RAPT has significantly lower overhead than DRT and RPT, while MAPT's overhead is only marginally lower than that of DRT and RPT. Furthermore, RAPT has consistently shown the best performance for all three object programs across all four metrics. This further indicates that among the proposed APT techniques, RAPT should be used

Our experiments involved three levels of granularity and two types of initial probability profile. Our investigations showed that there was no strong correlation between the granularity/profile and the performance. It is thus necessary to further study the proper partitioning scheme(s) with appropriate granularity level for APT and relevant techniques. It is also of importance to investigate how to design a "good" probability profile aiming at optimizing the testing effectiveness.

One piece of future work is about the appropriate settings of the parameters (such as ϵ, δ, γ, and τ) in APT. On one hand, the settings we used in this study are based on the previous experience and/or empirical estimation. It is worthwhile to investigate the relationship between different settings and the performance of APT, and thus hopefully work out a guideline on how to set these parameters. On the other hand, no strong correlation between the initial probability profile and the performance was found, which should be further investigated. Our empirical studies compared APT with two baseline techniques that integrate RT and PT. It is also interesting to further compare APT with other testing techniques, such as ART. Last but not the least, it is interesting to integrate MAPT and RAPT as well as other relevant techniques, with the purpose of further enhancing the performance of APT.

ACKNOWLEDGMENTS

This research is supported by the Beijing Natural Science Foundation (Grant No. 4162040), the National Natural Science Foundation of China (Grant No. 61370061), the Aeronautical Science Foundation of China (Grant No. 2016ZD74004), and the Fundamental Research Funds for the Central Universities (Grant No. FRF-GF-17-B29).

REFERENCES

[1] R. Hamlet, "Random testing," in *Encyclopedia of Software Engineering*. John Wiley & Sons, Inc., 2002.

[2] E. J. Weyuker and B. Jeng, "Analyzing partition testing strategies," *IEEE Transactions on Software Engineering*, vol. 17, no. 7, pp. 703–711, 1991.

[3] D. Hamlet and R. Taylor, "Partition testing does not inspire confidence," *IEEE Transactions on Software Engineering*, vol. 16, no. 12, pp. 1402–1411, 1990.

[4] T. Y. Chen and Y. T. Yu, "On the relationship between partition and random testing," *IEEE Transactions on Software Engineering*, vol. 20, no. 12, pp. 977–980, 1994.

[5] ——, "On the expected number of failures detected by subdomain testing and random testing," *IEEE Transactions on Software Engineering*, vol. 22, no. 2, pp. 109–119, 1996.

[6] W. J. Gutjahr, "Partition testing vs. random testing: The influence of uncertainty," *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 661–674, 1999.

[7] K.-Y. Cai, T. Jing, and C.-G. Bai, "Partition testing with dynamic partitioning," in *Proceedings of the 29th Annual International Conference on Computer Software and Applications Conference Workshops (COMPSACW'05)*, 2005, pp. 113–116.

[8] K.-Y. Cai, B. Gu, H. Hu, and Y.-C. Li, "Adaptive software testing with fixed-memory feedback," *Journal of Systems and Software*, vol. 80, no. 8, pp. 1328–1348, 2007.

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

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

[11] K.-Y. Cai, H. Hu, and F. Ye, "Random testing with dynamically updated test profile," in *Proceedings of the 20th International Symposium on Software Reliability Engineering (ISSRE'09)*, 2009, Fast abstract 198.

[12] K.-Y. Cai, "Optimal software testing and adaptive software testing in the context of software cybernetics," *Information and Software Technology*, vol. 44, no. 14, pp. 841–855, 2002.

[13] J. Lv, H. Hu, and K.-Y. Cai, "A sufficient condition for parameters estimation in dynamic random testing," in *Proceedings of the IEEE 35th Annual Computer Software and Applications Conference Workshops (COMPSACW'11)*, 2011, pp. 19–24.

[14] Z. Yang, B. Yin, J. Lv, K.-Y. Cai, S. S. Yau, and J. Yu, "Dynamic random testing with parameter adjustment," in *Proceedings of the IEEE 38th International Computer Software and Applications Conference Workshops (COMPSACW'14)*, 2014, pp. 37–42.

[15] Y. Li, B. B. Yin, J. Lv, and K.-Y. Cai, "Approach for test profile optimization in dynamic random testing," in *Proceedings of the IEEE 39th Annual Computer Software and Applications Conference (COMPSAC'15)*, vol. 3, 2015, pp. 466–471.

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

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

[18] S. Holm, "A simple sequentially rejective multiple test procedure," *Scandinavian Journal of Statistics*, vol. 6, pp. 65–70, 1979.

[19] L. V. Hedges and I. Olkin, *Statistical Methods for Meta-Analysis*. Academic Press, 1985.

[20] S. Sawilowsky, "New effect size rules of thumb," *Journal of Modern Applied Statistical Methods*, vol. 8, no. 2, pp. 467–474, 2009.

[21] J. Lv, B.-B. Yin, and K.-Y. Cai, "On the asymptotic behavior of adaptive testing strategy for software reliability assessment," *IEEE Transactions on Software Engineering*, vol. 40, no. 4, pp. 396–412, 2014.

[22] J. Lv, H. Hu, K.-Y. Cai, and T. Y. Chen, "Adaptive and random partition software testing," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 44, no. 12, pp. 1649–1664, 2014.

[23] C.-A. Sun, G. Wang, K.-Y. Cai, and T. Y. Chen, "Towards dynamic random testing for web services," in *Proceedings of the IEEE 36th Annual Computer Software and Applications Conference (COMPSAC'12)*, 2012, pp. 164–169.

[24] L. J. White and E. I. Cohen, "A domain strategy for computer program testing," *IEEE Transactions on Software Engineering*, vol. 6, no. 3, pp. 247–257, 1980.

[25] P. G. Bishop, "The variation of software survival time for different operational input profiles (or why you can wait a long time for a big bug to fail)," in *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing (FTCS'91)*, 1993, pp. 98–107.

[26] A. G. Koru, K. E. Emam, D. Zhang, H. Liu, and D. Mathew, "Theory of relative defect proneness," *Empirical Software Engineering*, vol. 13, no. 5, pp. 473–498, 2008.

[27] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. H. Tse, "Adaptive random testing: The ART of test case diversity," *Journal of Systems and Software*, vol. 83, no. 1, pp. 60–66, 2010.

[28] A. C. Barus, T. Y. Chen, F.-C. Kuo, H. Liu, R. Merkel, and G. Rothermel, "A cost-effective random testing method for programs with non-numeric inputs," *IEEE Transactions on Computers*, vol. 62, no. 12, pp. 3508–3529, 2016.

[29] T. Y. Chen, F.-C. Kuo, and H. Liu, "Application of a failure driven test profile in random testing," *IEEE Transactions on Reliability*, vol. 58, no. 1, pp. 179–192, 2009.

[30] H. Liu, X. Xie, J. Yang, Y. Lu, and T. Y. Chen, "Adaptive random testing through test profiles," *Software: Practice and Experience*, vol. 41, no. 10, pp. 1131–1154, 2011.

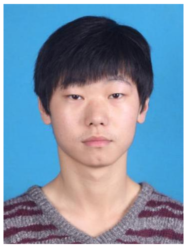
[31] T. Y. Chen and R. Merkel, "Quasi-random testing," *IEEE Transactions on Reliability*, vol. 56, no. 3, pp. 562–568, 2007.

[32] H. Liu and T. Y. Chen, "Randomized quasi-random testing," *IEEE Transactions on Computers*, vol. 65, no. 6, pp. 1896–1909, 2016.

[33] H. Liu, F.-C. Kuo, and T. Y. Chen, "Comparison of adaptive random testing and random testing under various testing and debugging scenarios," *Software: Practice and Experience*, vol. 42, no. 8, pp. 1055–1074, 2012.

[34] Z. Q. Zhou, A. Sinaga, L. Zhao, W. Susilo, and K.-Y. Cai, "Improving software testing cost-effectiveness through dynamic partitioning," in *Proceedings of the 9th International Conference on Quality Software (QSIC'09)*, 2009, pp. 249–258.

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



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

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

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

Kai-Yuan Cai received the BS, MS, and PhD degrees from Beihang University (Beijing University of Aeronautics and Astronautics), Beijing, China, in 1984, 1987, and 1991, respectively. He has been a full professor at Beihang University since 1995. He is a Cheung Kong Scholar (chair professor), jointly appointed by the Ministry of Education of China and the Li Ka Shing Foundation of Hong Kong in 1999. His main research interests include software testing, software reliability, reliable flight control, and software cybernatics.