Adaptive and Random Partition Software Testing

Junpeng Lv, Hai Hu, Kai-Yuan Cai, and Tsong Yueh Chen, *Member, IEEE*

***Abstract*—Random testing (RT) and subdomain testing are two major software testing strategies. Their simplicity makes them likely the most efficient testing strategies with respect to the time required for test case selection. However, the disadvantage of RT is its defect detection effectiveness. Adaptive testing (AT) is a feedback-based software testing strategy that has been shown to be more effective than RT and partition testing (PT). However, a major concern in the application of AT is its complexity and computational cost for test case selection. In this paper, we propose a hybrid approach that uses AT and random partition testing (RPT) in an alternating manner. The motivation for this approach is that both strategies are employed such that the underlying computational complexity of AT is reduced by introducing RPT into the testing process without affecting the defect detection effectiveness. A case study with seven real-life subject programs is presented. The experimental results demonstrate that this novel strategy considerably reduces the computational overhead of the original AT strategy but still outperforms the pure RT strategy and PT strategy in terms of the number of test cases used to detect and remove a given number of defects. In addition, a sensitivity analysis is conducted to validate the robustness of our strategy.**

***Index Terms*—Adaptive testing, random partition testing, random testing, software testing.**

I. Introduction

RANDOM testing (RT) and subdomain testing are two major software testing strategies. In traditional RT [1], the input domain of the software under test (SUT) includes a large number of distinct inputs. In each run, an input is selected as a test case and executed according to a uniform or nonuniform probability distribution. No partitioning is applied to the input domain. In contrast, in subdomain testing [2], the input domain of the SUT is partitioned into a number of disjoint or overlapping subdomains from each of which one or more test cases are generated. The resulting test suite is composed of a modest number of test cases that are exhaustively executed against the SUT. More specifically, if the subdomains are disjoint, the testing strategy is also referred to as partition testing (PT). A new testing strategy known as random partition testing (RPT) is proposed by combining RT and PT [3], [4]. Suppose that the input domain of the SUT is partitioned into m subdomains $D_1, D_2,\ldots, D_m$, with the ith subdomain D_i composed of d_i distinct test cases. The RPT strategy selects a test case in two steps. First, a subdomain is selected according to a uniform probability distribution, i.e., the probability that each subdomain is selected is $1/m$. Secondly, a test case is picked from the selected subdomain and executed according to a uniform probability distribution. Suppose that the jth subdomain is selected, then the probability that any test case in the jth subdomain is picked is $1/d_j$. If no partitioning is conducted on the input domain, then RPT will effectively become RT.

The simplicity of RT and PT makes them likely the most efficient testing strategies with respect to the cost of test case selection. However, the disadvantage of RT is its defect detection effectiveness. Much effort has been invested in improving the effectiveness of RT and RPT [3], [5]–[8]. A particular testing technique known as the adaptive testing (AT) strategy is proposed to utilize the testing history collected online to improve the testing effectiveness of RPT [9]. More specifically, the SUT is modeled using a controlled Markov chain (CMC), and the first step of RPT (i.e., selection of the subdomain) is achieved by solving the CMC model using successive approximation techniques to minimize the overall testing cost. Fig. 1 provides a pictorial overview of the AT strategy. Case studies on real-life software applications indicate that AT can improve the effectiveness of RT and RPT by reducing the number of test cases required to detect and remove a certain number of defects.

However, a major concern in the application of AT is the computational overhead incurred during the decision-making process. Both analytical and experimental analyses indicate that AT is more computationally complex than RT and RPT, and thus, a considerably longer time is required to select test cases for execution [10]. In practice, this effect is undesirable when the time budget for testers is limited. If the application of advanced testing techniques will result in the delay of the testing plan itself, the advantages of these techniques over RT will be reduced, if not completely offset. One possible approach for alleviating this problem is to reduce the computational complexity of the AT algorithm. A fixed memory approach is proposed to improve the original genetic algorithm-based AT-GA strategy, which limits the testing history under consideration to a fixed memory window [3]. However, in certain scenarios, the improved AT strategy still incurs an unreasonable overhead, and the use of a constant memory throughout the entire testing process does not appear to be feasible [11]. Balancing testing effectiveness and time efficiency remains a major challenge in the application of the AT approach and many advanced testing techniques.

In this paper, we address this challenge by combining the AT and RPT strategies. The new testing technique, which is known as the adaptive and random partition testing (ARPT)

Manuscript received April 10, 2012; revised October 30, 2013; accepted January 15, 2014. Date of publication May 7, 2014; date of current version November 13, 2014. This work was supported in part by an ARC LP grant and in part by the National Natural Science Foundation of China under Grant 61272164. This paper was recommended by Associate Editor A. Bargiela.

J. Lv, H. Hu, and K.-Y. Cai are with the School of Automation Science and Electrical Engineering, Beihang University, Beijing 100191, China (e-mail: realljp@ buaa.edu.cn; huhai.orion@gmail.com; kycai@buaa.edu.cn).

T. Y. Chen is with the School of Software and Electrical Engineering, Swinburne University of Technology, Hawthorn, VIC 3122, Australia (e-mail: tychen@swin.edu.au).

Digital Object Identifier 10.1109/TSMC.2014.2318019

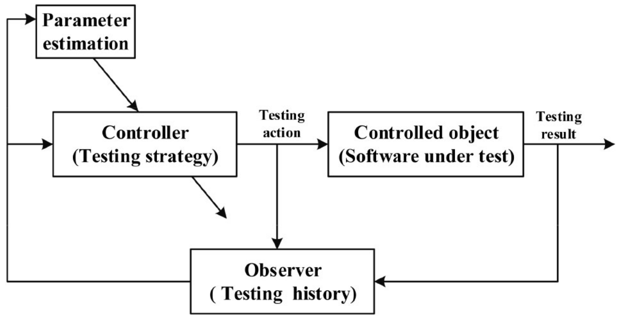

Parameter estimation
Controller (Testing strategy)
Testing action
Controlled object (Software under test)
Testing result
Observer (Testing history)

Fig. 1. Overview of adaptive testing.

strategy, is based on a simple concept: AT and RPT are utilized in an alternating manner to leverage the strength of each strategy. The advantage of this approach is two-fold. First, the use of AT in the testing process is expected to improve the defect detection effectiveness. Secondly, RPT is used to reduce the computational cost required to select the subdomain or test class. By carefully setting the switching point of the testing strategies, the testing process can be optimized, and a balance between testing effectiveness and time efficiency can be achieved. Two ARPT strategies are proposed and examined via experiments on eight versions of seven real-life software applications. Moreover, a sensitivity analysis is conducted using experiments to further investigate the robustness of the proposed ARPT strategies.

The remainder of this paper is organized as follows. In Section II, related studies are reviewed. In Section III, two ARPT strategies are proposed, and the detailed algorithms are described. A case study is presented and analyzed in Section IV. In Section V, a sensitivity analysis and experimental validation are presented. A general discussion is provided in Section VI, and conclusions and future research plans are summarized in Section VII.

II. Related Studies

This paper is related to several topics in software testing.

First, this paper is related to researches on RT and PT, and many research studies have been published on the performance of RT and PT. The simulation results and actual RT experiments conducted by Duran and Ntafos [1] suggest that RT may often be more cost effective than PT schemes and confirm the viability of RT as a useful validation tool. Frankl *et al.* [12] examined the relationship between debugging testing and operational testing using probabilistic analysis. Testing effectiveness was measured in terms of the delivered reliability of a program after testing, which was used to compare these two testing techniques and explore the favorable conditions for each of these techniques. Gutjahr [2] compared PT and RT under the assumption that program failure rates are not known with certainty before testing, and modeled the program failure rates using random variables. Weyuker and Jeng [13] investigated the conditions that affect the efficacy of PT and compared the fault detection capabilities of PT and RT.

Chen *et al.* [14] found that for the case of disjoint subdomains, if the number of test cases selected from each subdomain is proportional to its size, which is known as the proportional sampling strategy (PSS), the probability of revealing at least one failure using subdomain testing is not less than that of RT. Chen *et al*. [14] proposed the Basic Maximin Algorithm; the main idea of this algorithm is to initially allocate one test case to every subdomain to ensure coverage, and allocate additional test cases individually to the subdomain whose sampling rate is currently the lowest in each subsequent iteration. Based on the total ordering among the failure rates of (possibly overlapping) input subdomains, Morasca and Serra-Capizzano [15] introduced necessary and sufficient conditions for comparing the expected values of the numbers of failures caused by the applications of the testing techniques. Their work addresses and compares applications of random- and subdomain-based testing techniques in a unified manner. Moreover, "adaptive" testing techniques in which the test case selection at the *k*th input selection may depend on the results obtained in the previous (*k*-1) selections can also be assessed using the comparison approach.

Cai *et al.* [4] proposed RPT as a combination of RT and PT. Indeed, RPT is intimately related to traditional PT. The number of test cases selected from each subdomain is deterministic in PT but stochastic in RPT, which makes RPT a generalized form of PT.

The simplicity of RT and PT makes them likely the most efficient testing strategies with respect to the computation time required for test case selection. However, their disadvantages lie in defect detection effectiveness.

Thus, much effort has been invested in improving the effectiveness of RT, e.g., adaptive random testing (ART) [16]. ART is an enhanced form of RT that aims to distribute the test cases more evenly within the input space. Chen and Merkel [5] analytically examined the possible effect of failure patterns on software testing and obtained an upper bound on the potential effectiveness improvement by assuming additional knowledge of the regions. This theoretical analysis on the upper bound of RT was also intended to demonstrate the superiority of ART. Chan *et al.* [6] presented a novel adaptation of traditional RT known as restricted random testing (RRT), which offers a significant improvement over RT, as assessed by the F-measure. Shahbazi *et al.* [17] recently adopted centroidal Voronoi tessellations to enhance RT by improving the coverage of the input space. Pacheco *et al.* [18] proposed a technique referred to as feedback-directed RT, in which test sequences for the unit testing of object-oriented programs are incrementally generated using feedback obtained from the execution of the previously constructed sequences. An automatic testing tool known as DART was proposed by Godefroid *et al.* [19] to automatically generate the testing environment and test inputs. Moreover, DART can analyze the behavior of the SUT under RT and generate new test inputs to systematically direct the execution along alternative program paths.

Many studies have attempted to improve the effectiveness of PT. A particular testing technique known as the AT strategy [9] was proposed to utilize the testing history collected online to improve the testing effectiveness of RPT. Cai *et al.* [3] proposed an AT methodology using a fixed-memory feedback mechanism, i.e., AT-GA, to improve the defect detection effectiveness of traditional RPT. A series of case studies on the Space program was reported, and the experimental results demonstrated the advantage of AT in various testing scenarios. Hu *et al.* extended the AT approach for software component testing in [20]. An AT strategy that employed recursive least squares estimation (RLSE) was proposed to estimate the failure rates of the software components, and experiments on three subject programs were reported. Cai *et al.* also proposed the use of AT for optimal software reliability assessment in [21]. The AT strategy was also used to improve the parameter estimation process in that paper. In [22], the proposed AT-GM strategy was based on the general CMC model in which certain unrealistic assumptions in previous AT studies were discarded to widen the range of application. Cai *et al.* [7], [8] followed the idea of software cybernetics and proposed a simple yet effective technique of dynamic random testing (DRT).

The AT strategy aims to improve the effectiveness of PT and RPT, whereas other studies, e.g., ART, primarily focus on the improvement of RT. Although many research studies have focused on comparisons between RT and PT, one important question is how to take into account the cost of partitioning the test suite in PT, because this cost is not involved in RT. For certain programs, creating partitions requires considerable effort, but this process may cost less for others. The comparison between RT and PT may be biased if the cost of partitioning is not considered. Thus, the same problem should be encountered in comparing AT with other advanced strategies based on RT.

The studies on AT strategies differ from previous studies on PT. Previous studies primarily focus on comparing RT with PT using different metrics [1], [2], [13]–[15] and on methods of implementing PT [13], [14] that are often offline. The AT is an online testing strategy that attempts to improve PT using feedback information. Although "adaptive" testing techniques were also investigated in [15], the focus was on quantitative assessment of these "adaptive" techniques but not on how to improve them.

In this paper, we focus on solving the problem of tradeoffs between testing effectiveness and efficiency to improve PT and RPT. These types of tradeoffs can also be encountered in evaluating the improvement over RT. For example, ART selects test cases from a limited number of candidates instead of the entire input space to reduce the computational overhead while still maintaining ARTs advantage over RT.

III. Adaptive and Random Partition Testing

Several viable options exist for combining AT and RPT in a testing process.

1) Use AT to choose a certain number of test cases and then use RPT to select a number of test cases before returning to AT.
2) Use AT to choose the first *m* test cases and then switch to RPT for the remaining tests.

Fig. 2 illustrates the testing processes for the two aforementioned options.

Fig. 2 shows two variants of the ARPT strategy, i.e., ARPT-1 and ARPT-2, which are described in Table I.

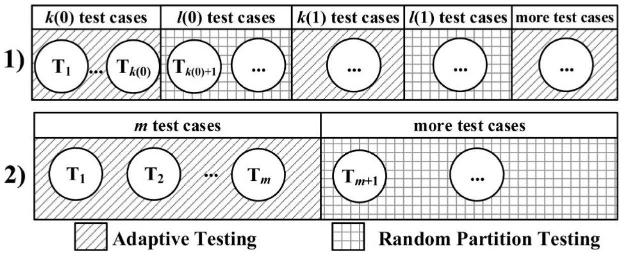


Fig. 2. Two ARPT options.

TABLE I
Two ARPT Strategies

ARPT-1: ARPT Approach #1	ARPT-2: ARPT Approach #2

initialize(t=0, n=0, $S_F(t)$=0 and
$k(0)=k_0$, $l(0)=l_0$);
while(*the stopping criterion is not satisfied*){
$n = n + 1$;
If ($n \le k(t)$) {
T_n = **adaptive**( );
}
If($k(t)<n\le k(t)+l(t)$){
T_n = **random partition**( );
}
execute(T_n);
If (failure == TRUE) {
removeDefect( );
$S_F(t)$=1;
}
If ($n == k(t)+l(t)$) {
($k(t+1)$,$l(t+1)$)=**CalcSteps**($k(t)$,$l(t)$,
$S_F(t)$);
$t = t+1$;$n = 0$; $S_F(t) = 0$;
}
}
	initialize(n=0 and m);
	while(*the stopping criterion is not satisfied*){
	If ($n < m$) {
	T_n = **adaptive**( );
	}
	else{
	T_n = **random partition**( );
	}
	execute(T_n);
	If (failure == TRUE) {
	removeDefect();
	}
	$n = n + 1$;
	}

In ARPT-1, several parameters require initialization, i.e., t, n, $S_F(t)$, $k(0)$, and $l(0)$. The parameter t denotes the "state" of the current testing process. In this paper, the term "state" denotes an AT segment and a RPT segment. For example, the ith state should be a testing process that contains an AT process with $k(i)$ test cases and a RPT process with $l(i)$ test cases. The parameter n is used to determine when to change the testing strategy. The signal $S_F(t)$ is retained to record whether any defect is detected. In this paper, the parameters $k(0)$ and $l(0)$ are initialized to k_0 and l_0, that is, these values are determined according to the testing scenarios. For ARPT-2, parameter m requires initialization in addition to the parameter n, as shown in Table I. The parameter m is the testing length for the AT segment because there is only one switching point in ARPT-2.

A. Adaptive Testing

AT plays an important role in ARPT strategies. In AT, the software testing process is treated as a control problem: the corresponding testing strategy is treated as a control policy or controller, and the SUT is treated as a controlled object with uncertainty. Furthermore, if an optimization (testing) goal is explicitly given, then the test case selection problem is translated into an optimal control problem. In an optimal control problem, a dynamic system (controlled object) is defined whose behavior follows its own "laws of motion" (e.g., Markov state transition laws) and may be influenced or regulated by suitable choices of certain system variables known as control or action variables.

The controls that can be applied at any time are chosen according to the control policies or laws that are derived from histories of the system and the given performance criterion or performance index. The performance criterion measures or evaluates the system responses to the applied control policies. The control policies (or controller) and controlled system constitute a closed-loop feedback system, as shown in Fig. 1. Then, the optimal control problem is translated into the problem of how to determine a control policy that optimizes (i.e., either minimizes or maximizes) the performance criterion. Therefore, we must identify three components to solve an optimal control problem: a model for the controlled object, a set of admissible control policies, and a performance index (or objective function).

For the software testing problem of interest, let
$Y_t = j$, if the SUT contains j defects at time t

$$j = 0,\ 1,\ 2, \ldots, N;\ t = 0,\ 1,\ 2, \ldots$$

$$Z_t = \begin{cases} 1, & \text{if the action taken at time } t \text{ detects a defect} \\ 0, & \text{if the action taken at time } t \text{ detects no defect.} \end{cases}$$

In this paper, a defect is defined as a certain error in the software, e.g., missing variable definitions before use, that results in unexpected or unacceptable external software behavior in certain situations, i.e., software failure. In this paper, an action refers to selection and execution of a test case from the input domain of the SUT. However, the term "action" can be interpreted more widely. It can refer to selection of a software run or an equivalence class of test cases for a specific testing technique. It can also refer to selection of a software testing technique, e.g., boundary testing, path testing, or RT. An action can even mean selection of a tester. Different actions have various defect detection capabilities, and the software demonstrates different degrees of testability under different actions. A control policy (testing strategy) specifies which action should be taken at any specified time.

Thus, a simple model of the software testing process is built on the following assumptions.

1) The software initially ($t = 0$) contains N defects.
2) Each action detects at most one defect.
3) If a defect is detected, then it is removed immediately, and no new defects are introduced, i.e., $Y_t = j$ and $Z_t = 1$ mean $Y_{t+1} = j - 1$.
4) If no defect is detected, then the number of remaining software defects remains unchanged, i.e., $Y_t = j$ and $Z_t = 0$ mean $Y_{t+1} = j$.
5) $Y_t = 0$ is an absorbing state and a target state.
6) At any time there are always m admissible actions, and the action set is $A = \{1, 2, \ldots, m\}$. Sometimes, a test case will make certain changes in the testing environment or the subject program to block other test cases from being selected and executed. For example, in a flight control system, a test can be defined as one maneuver, e.g., rudder rotation. In this case, one test can actually affect the following actions. If the rudder is positively rotated and it reaches the extreme position, it is forbidden to execute some maneuvers, e.g., positive rotation. Thus, positive rotation of the rudder is not admissible for the next test.
7) Z_t only depends on the software state Y_t and the action A_t taken at time t, i.e.,
$$\begin{aligned} &\mathrm{P}_r\left\{Z_t = 1 \middle| Y_t = j, A_t = i\right\} = j\theta_i \\ &\mathrm{P}_r\left\{Z_t = 0 \middle| Y_t = j, A_t = i\right\} = 1 - j\theta_i; \\ &\qquad\qquad\qquad\qquad j = 0,\ 1,\ 2, \ldots, N \\ &\mathrm{P}_r\left\{Z_t = 1 \middle| Y_t = 0, A_t = i\right\} = 0 \\ &\mathrm{P}_r\left\{Z_t = 0 \middle| Y_t = 0, A_t = i\right\} = 1;\ t = 0,\ 1,\ 2, \ldots \end{aligned}$$
where θ_i can be interpreted as the probability of a defect being detected by action i.
8) Action A_t taken at time t incurs a cost of $W_{Y_t}(A_t)$ regardless of whether it detects a defect; this cost could be the resource or time required to execute the action at time t. In practice, this value can be determined by the testers according to the actual testing cost
$$W_{Y_t}(A_t = i) = \begin{cases} w_j(i), & \text{if } Y_t = j \neq 0 \\ 0, & \text{if } Y_t = 0. \end{cases}$$
9) The cost of removing a detected defect is ignored.

In Assumption 3, every detected defect is removed immediately. This assumption may not be realistic in certain scenarios with imperfect debugging [23], i.e., when a failure is observed, the corresponding defects cannot be completely removed, and new defects might be introduced. Assumption 3 aims to provide a complete description of the testing process such that a CMC could be used to model this process. Incorporating imperfect debugging into the AT framework can be investigated in future studies.

We are interested in how many actions or tests are required to remove all N defects from the SUT. Suppose that the first $\tau-1$ actions (at times 0, 1, ..., $\tau-2$) detect (and remove) N-1 defects and the τth action (at time $\tau-1$) detects the last defect. Therefore, we have $Y_{\tau-1} = 1$ and $Y_\tau = 0$. Thus, τ is referred to as the first-passage time to state 0 ($Y_\tau = 0$). In other words, τ denotes the number of actions that are performed to remove all N defects. Furthermore, let ω denote the testing strategy that is adopted in the process of software testing. Thus, ω specifies how test cases are selected individually during software testing. ω is not specified a priori and should be determined with respect to a given testing goal [e.g., (1) below]. Different testing strategies should lead to different values of τ. Therefore, τ should be treated as a random variable. Consequently, the total cost incurred during the testing process is a random variable as well. The expected total cost (from the first action at time 0 to the last action at time $\tau-1$) is denoted as

$$J_\omega(N) = E_\omega \sum_{t=0}^{\tau-1} W_{Y_t}(A_t). \tag{1}$$

$W_{Y_t}(A_t)$ is the cost incurred by action A_t taken at time t (according to Assumption 8), and τ is the total number of actions required to detect and remove all the N defects from the SUT. Therefore, $\sum_{t=0}^{\tau-1} W_{Y_t}(A_t)$ is the total cost incurred in a single testing process that detects and removes all N defects.

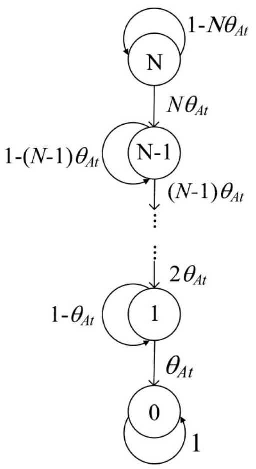


Fig. 3. Software state transition under test.

The expression $E_\omega \sum_{t=0}^{\tau-1} W_{Y_t}(A_t)$ represents the mathematical expectation of $\sum_{t=0}^{\tau-1} W_{Y_t}(A_t)$ under the testing strategy ω. To simplify the mathematical notation, this quantity is denoted by $J_\omega(N)$, as shown in (1). A natural objective is to determine the testing strategy ω that minimizes $J_\omega(N)$. Such a testing strategy removes all N defects with the least expected total cost. A special case is that for every action i, there exists

$$W_{Y_t}(A_t = i) = \begin{cases} 1, & \text{if } Y_t \neq 0 \\ 0, & \text{if } Y_t = 0. \end{cases}$$

Then, the corresponding optimal testing strategy detects and removes all N defects with the minimum number of actions (tests), i.e., $\sum_{t=0}^{\tau-1} W_{Y_t}(A_t) = \tau$.

Assumptions 1–7 define a model for the controlled object component shown in Fig. 1. This model describes how many distinct inputs (actions) are acceptable to the SUT and how the SUT can possibly behave for distinct inputs.

The above assumptions and (1) define a simple CMC as shown in Fig. 3. In addition, based on the theory of CMC, the optimal action A_t that should be taken at time t in state $Y_t = j$, $j \epsilon \{N, N-1, \ldots, 2\}$ can be determined by the following recursive formula given in [9]:

$$v_{n+1}(j) = \min_{1\le i\le m} \left\{ W_{Y_t}(i) + \sum_{k\neq 0} q_{jk}(i)\, v_n(k) \right\}, n = 0, 1, 2, \ldots$$

where $q_{jk}(i)$ denotes the probability that action i transforms the software from state j to state k, and $\{v_0(k); k = 1, 2 \ldots, N\}$ are arbitrary. Let $\lim_{n\to 8} v_{n+1}(j) = v(j)$. We have the following equations:

$$\begin{aligned} v(1) &= \min\nolimits_{1\le i\le m} \left\{ \frac{W_1(i)}{\theta_i} \right\} \\ v(j) &= \min\nolimits_{1\le i\le m} \left\{ \frac{W_j(i)}{j\theta_i} + v(j-1) \right\}; \quad j = 2, 3, \ldots N. \end{aligned} \quad (2)$$

Equation (2) implies that in state $Y_t \neq 0$, the global optimal action is the one that minimizes $\frac{W_{Y_t}(i)}{\theta_i}$ with respect to i. In this case, $A_t = arg\ \min_{1\le k\le m}\{\frac{W_{Y_t}(i)}{\theta_i}\}$. Thus, the controller component shown in Fig. 1 can be determined. A Markov process can increase the computational complexity. Thus, several approaches are proposed to solve this problem, e.g., computerized AT [24], [25]. In this paper, the CMC is only required to provide the solution for the optimal action, and it does not incur an additional cost in the decision-making process beyond this stage.

In [9], the unknown parameters $\theta_1, \theta_2, \ldots, \theta_m$ are involved in implementing the optimal testing strategy. Thus, these unknown parameters must be estimated online. This leads to an adaptive software testing strategy (i.e., the optimal testing strategy with an online parameter estimation scheme), as shown in Fig. 1. The estimates of the unknown parameters are updated using the history of testing data and are treated as the true values of these parameters in decision-making. This process illustrates the certainty-equivalence principle in adaptive control [26].

We set $z_{-1} = 0$. Let $\{z_t, t = 0, 1, 2, \ldots\}$ be realization of $\{Z_t, t = 0, 1, 2, \ldots\}$. Furthermore, let $\{a_t, t = 0, 1, 2, \ldots\}$ be realization of $\{A_t, t = 0, 1, 2, \ldots\}$, i.e., at time t, action a_t is taken and it generates an observation $Z_t = z_t$. Therefore, $(z_0, z_1, \ldots, z_t; a_0, a_1, \ldots, a_t) = H_t$ represents the entire history of the testing results and constitutes the observer component shown in Fig. 1. Furthermore, let

$$r_t = \left(N - \sum_{j=-1}^{t-1} z_j \right) \theta_{a_t} \quad (3)$$

where r_t represents the probability that action a_t detects a defect.

In this model, the testing result of a test case from a given action at time t is a random variable that follows a binomial distribution determined by theoretical prediction (3). Thus, for each action, the accumulated deviation/error between (3) and the corresponding testing results should be minimized. According to Assumption 6, there are always m admissible actions such that the sum of deviations for these m actions should also be minimized. Thus, given $z_0, z_1, \ldots, z_t$ and $a_0, a_1, \ldots, a_t$, we can use the least squares method to perform parameter estimation. Therefore, the parameter estimation component shown in Fig. 1 can be defined. Let

$$L(z_0, z_1, \ldots, z_t; a_0, a_1, \ldots, a_t) = \sum_{j=0}^{t} (z_j - r_j)^2. \quad (4)$$

The $m + 1$ parameters $N, \theta_1, \theta_2, \ldots, \theta_m$ are estimated by minimizing the function $L(z_0, z_1, \ldots, z_t; a_0, a_1, \ldots, a_t)$, and the resulting estimates, denoted as $N^{(t+1)}, {\theta_1}^{(t+1)}, \ldots, {\theta_m}^{(t+1)}$, are functions of the entire history of the testing data up to the current instant of time, i.e.,

$$\begin{aligned} N^{(t+1)} &= N(z_0, z_1, \ldots, z_t; a_0, a_1, \ldots, a_t), \\ {\theta_i}^{(t+1)} &= \theta_i(z_0, z_1, \ldots, z_t; a_0, a_1, \ldots, a_t); \ i = 1, 2, \ldots m. \end{aligned} \quad (5)$$

A method that is commonly used to obtain these estimates is to differentiate $L(z_0, z_1, \ldots, z_t; a_0, a_1, \ldots, a_t)$ with respect to each parameter of interest and obtain a system of nonlinear equations. However, $L(z_0, z_1, \ldots, z_t; a_0, a_1, \ldots, a_t)$ is not always differentiable, and sometimes the resulting system of nonlinear equations may not have a solution. A more general approach to parameter estimation is to minimize

$L(z_0, z_1, \ldots, z_t; a_0, a_1, \ldots, a_t)$ directly using the genetic algorithm. However, for large-scale software systems, the total number of actions (tests) that should be applied is extremely large. Thus, using the complete history of testing data to perform online parameter estimation induces a severe computational complexity problem. One possible solution is to use the testing data generated by the latest l actions (tests) for the parameter estimation. In this case, (4) and (5) are transformed into (6) and (7) as follows:

$$L\left(z_{t_0}, z_{t_0+1}, \ldots, z_t; a_{t_0}, a_{t_0+1}, \ldots, a_t\right) = \sum_{j=t_0}^{t} (z_j - r_j)^2 \quad (6)$$

$$\begin{aligned} N^{(t+1)} &= N\left(z_{t_0}, z_{t_0+1}, \ldots, z_t; a_{t_0}, a_{t_0+1}, \ldots, a_t\right) \\ \theta_i{}^{(t+1)} &= \theta_i\left(z_{t_0}, z_{t_0+1}, \ldots, z_t; a_{t_0}, a_{t_0+1}, \ldots, a_t\right); \quad (7) \\ & \quad i = 1, 2, \ldots m \end{aligned}$$

where $t_0 = min\{0, t - l + 1\}$.

Based on the above four components shown in Fig. 1, we obtain the following AT strategy.

1) Initialize parameters. Set $z_{-1} = 0$, $N^{(0)} = N_0$, $Y_0 = N^{(0)}$, $\theta_i{}^{(0)} = \theta_{0i}$, $i = 1, 2, \ldots, m$, $t = 0$, and $A_0 = arg \min_{1\le k\le m}\{\frac{W_{Y_0}(k)}{\theta_k{}^{(0)}}\}$, where A_0 denotes the first action performed for software testing.
2) Set l to a given value, which is typically determined based on the knowledge of the subject programs. Let $t_0 = min\{0, t-l+1\}$. The testing data collected at times $\{t_0, t_0 + 1, \ldots, t\}$ are used to update the estimates of the parameters at time t +1 in step 3.
3) Estimate the parameters by minimizing $L\ (z_{t_0}, z_{t_0+1}, \ldots, z_t;\ \ a_{t_0}, a_{t_0+1}, \ldots, a_t)$ according to (6) and obtain

$$\begin{aligned} N^{(t+1)} &= N\left(z_{t_0}, z_{t_0+1}, \ldots, z_t; a_{t_0}, a_{t_0+1}, \ldots, a_t\right) \\ \theta_i{}^{(t+1)} &= \theta_i\left(z_{t_0}, z_{t_0+1}, \ldots, z_t; a_{t_0}, a_{t_0+1}, \ldots, a_t\right) \end{aligned}$$

$i = 1, 2, \ldots m$. If θ_i does not appear in $L(z_{t_0}, z_{t_0+1}, \ldots, z_t; a_{t_0}, a_{t_0+1}, \ldots, a_t)$, or action i was not taken in the latest l actions, then pick a value from the unity interval [0,1] at random in accordance with the uniform distribution and assign this value to $\theta_i{}^{(t+1)}$ (this step specifies the parameter estimation scheme).
4) Determine the optimal action

$$A_{t+1} = arg \min_{1\le k\le m}\{\frac{W_{Y_{t+1}}(k)}{\theta_k{}^{(t+1)}}\} .$$

(This step specifies the optimal controller or testing strategy.)
5) Observe the testing result $Z_{t+1} = z_{t+1}$ activated by the action A_{t+1}.
6) Update the current software state that represents the number of remaining defects

$$Y_{t+1} = N^{(t+1)} - \sum_{j=-1}^{t+1} z_j.$$

7) Set $t = t + 1$.
8) If any given testing stopping criterion is satisfied, then stop testing; otherwise, go to step 3.

B. Parameter Setting

In the use of ARPT strategies, a key point is to determine when to switch to the other strategy, and a heuristic method is proposed for ARPT-1 to solve this problem. The underlying concept is that the length of a testing segment at state t [i.e., $k(t)$ or $l(t)$] should be adjusted based on the testing history in which defects may or may not be found. More specifically, if any defect is detected in the jth state, the current segment lengths [i.e., $k(j)$ and $l(j)$] are "suitable" and should be kept unchanged; otherwise, longer testing segments are preferred.

In this paper, an example of the heuristic method proposed for ARPT-1 is implemented. First, the initial testing segment lengths for AT and RPT in ARPT-1 are set, and the testing process begins. After each RPT segment, we check whether any defects are observed in the latest AT and RPT segments. If any defects are detected, the testing segment lengths for both AT and RPT should be kept unchanged; otherwise, the new lengths are set to twice the old lengths, i.e., the lengths increase exponentially. The detailed algorithm is described in the following steps.

1) Initialize $t = 0$, $n = 0$, $k(0) = k_0$, $l(0) = l_0$, and $S_F(i) = 0$, $i = 0, 1, 2 \ldots$.
2) If $n = k\,(t) + l\,(t)$, then let

$$k\,(t+1) = \begin{cases} 2 * k(t), & \text{if } \mathrm{S}_F\,(t) = 0 \\ k(t), & \text{if } \mathrm{S}_F\,(t) = 1 \end{cases}$$

$$l\,(t+1) = \begin{cases} 2 * l(t), & \text{if } \mathrm{S}_F\,(t) = 0 \\ l(t), & \text{if } \mathrm{S}_F\,(t) = 1 \end{cases}$$

$t = t + 1$, and $n = 0$.
3) Set $n = n + 1$.
4) If $n \le k\,(t)$, conduct AT; if $k\,(t) < n \le l\,(t)$, conduct RPT.
5) If any defect is detected, $S_F(t) = 1$.
6) If any given stopping criterion is satisfied, then go to step 7; otherwise, go to step 2.
7) Stop the testing process.

For ARPT-2, there is no generalized solution for setting the parameter m. This parameter is defined empirically based on the practitioner's knowledge and experience with the subject program, e.g., estimating the AT performance for the corresponding program, and this is a drawback of the proposed approach.

Compared with ARPT-1, ARPT-2 appears to be more subjective because m is the only parameter used to determine the performance of ARPT-2, and this parameter is highly dependent on the practitioner's experience. However, in this section, a heuristic method is proposed for parameter setting in ARPT-1. Will these two approaches work? How should these parameters be set in real-life software testing, and how will they affect the performance of ARPT? These problems will be addressed experimentally in the following section.

IV. Case Study

A set of experiments is presented in this section. The purpose of these experiments is to investigate the performance of ARPT, i.e., whether ARPT can use fewer test cases than RT and RPT to detect all defects, while simultaneously incurring a lower computational overhead than AT.

TABLE II
SUBJECT PROGRAMS

Subject Programs	LOC	# of Defects	# of Test Cases	# of Classes	From
Space-36	9,564	36	13,466	4	European Space Agency
Space-21	9,564	21	13,466	4	European Space Agency
tcas	173	34	1,606	5	Siemens
replace	564	30	5,717	5	Siemens
sed	14,427	6	21,506	2	GNU
gzip	5,680	6	8,653	7	GNU
flex	10,459	16	18,561	22	GNU
SESD	3,559	26	5,938	17	Self-developed

Eight versions of seven different real-life software applications serve as the subject programs in the experiments. These subject programs are chosen from a variety of open-source software projects, including the Space program from the European Space Agency, the Siemens suite assembled by Siemens Corporate Research, UNIX utilities developed by GNU, and a source code analyzer developed for a domestic corporation by former graduate students in our research group.

A. Subject Programs and Experiment Configuration

Table II provides detailed information on each subject program, including the scale, the number of defects, the size and partition of the test suite, and the origin.

The Space program consists of 9564 lines of code (LOC) (6218 executable), and functions as an interpreter for an array definition language (ADL). In previous studies, the Space program also served as the SUT in various experiments [9], [18]. Thirty-six defects are injected into the Space program, which is subsequently subjected to testing. Additional details on the defect injection for the Space program can be found in [3]. Considering that certain defects injected into the Space program can be easily detected (i.e., typically with fewer than 10 test cases), this might have a considerable impact on the experiments and results. To avoid this problem, another version of the Space program with 21 injected defects is also examined as an individual subject program in the experiments, and 15 defects that can be easily detected are removed in this version. The test suite used in this paper includes 13 466 distinct test cases (ADL files) and is partitioned into four disjoint classes at random.

The items *tcas* and *replace* are chosen from the Siemens programs and are widely used as subject programs for studies on fault localization techniques [27]. *tcas* is an aircraft collision avoidance system, and *replace* is a module that performs pattern matching and substitution. The researchers at Siemens aimed to study the fault detection effectiveness of different coverage criteria and thus created faulty versions of the programs by manually seeding those programs with defects, typically by modifying a single line of code in the program. Based on these faulty versions, we injected 34 and 30 defects into *tcas* and *replace*, respectively. Moreover, the test suites for *tcas* and *replace* were composed of 1606 and 5717 test cases, respectively, and both of them were randomly partitioned into five disjoint classes to serve as the test suite.

flex, *sed*, and *gzip* are commonly used Unix command line utilities developed by GNU. Several previously released versions were obtained from the Software-artifact Infrastructure Repository (http://sir.unl.edu/portal/index.html) together with their test suites and injected defects. *flex* is a lexical analyzer generator and it serves as a tool for generating programs that perform pattern matching on text. *sed* is used to filter text, i.e., it takes text input, performs an operation (or a set of operations) on the text, and outputs the modified text. *gzip* (GNU zip) is a compression utility designed as a replacement for *compress*. Compared with *compress*, *gzip* is free from patented algorithms and exhibits superior compression performance. Defects are injected into these subject programs according to the bug history, and code is deleted from, inserted into, or modified among the versions. The test suites that are generated based on the test specification language (TSL) for *flex*, *sed*, and *gzip* are composed of 21 506, 8653, and 18 561 test cases, respectively, and are partitioned into 2, 7, and 22 disjoint classes according to the functionalities that the test cases are designed to test, respectively.

The *SESD* is a C/C++ code analysis program developed for a domestic software corporation; it takes C/C++ source code as its input and generates syntax and a structural analysis report of the source code as its output. The *SESD* program is composed of 3559 LOC with 3179 executable lines. The *SESD* program is first implemented by one programmer and subsequently subjected to independent testing. Consequently, 26 defects are found and documented. The input to the *SESD* program should be any C/C++ source file. A total of 5938 test cases, i.e., C++ source code files, are collected and randomly partitioned into 17 disjoint classes to serve as the test suite.

We choose the above eight subject programs out of the numerous candidates offered by the Software-artifact Infrastructure Repository because we intend to examine the effectiveness of the ARPT strategies via experiments on subject programs with different properties, e.g., different scales, functionalities, and numbers of defects and test cases. More specifically, we choose the Siemens programs to represent small programs, i.e., programs with less than 1000 LOC. *gzip*, *SESD*, and the Space programs are chosen to represent "moderate-sized" programs, i.e., programs with 3000 to 10 000 LOC. *sed* and *flex* are large programs with more than 10 000 LOC. Meanwhile, the number of defects also varies, i.e., *sed* and *gzip* contain fewer than 10 injected defects, whereas the remaining programs have no fewer than 16 defects.

Moreover, the partitioning schemes for the test suites are different. For *SESD*, Space, and the Siemens programs, their test suites are partitioned into classes at random because we have no prior information on the functionality or coverage of each test case within the test suites obtained together with these subject programs. However, the test suites of *sed*, *flex*, and *gzip* are partitioned according to the functionalities, i.e., each test suite is partitioned based on the parameters and/or input files that the corresponding subject program takes as inputs.

For the subject programs, we develop a testing platform to conduct the testing process automatically. This platform selects test cases using a specific testing strategy and subsequently executes the test cases. The platform also uses an automated test "oracle" to detect failures and remove the detected defects, which means comparing the faulty version with the correct version to determine whether a failure occurs and ultimately removing the detected defect. In this paper, the selected test case is executed with replacement because all the defects are integrated into one faulty version. In doing so, we can imitate the interaction between defects in practice, e.g., "masking." "Masking" means that certain other defects cannot be detected before a particular defect is removed. Thus, retaining the executed test cases can provide the possibility of detecting masked defects. For defect removal, we retain the execution trace for each test case to record every defect invoked by that test case. If more than one defect is invoked by a failure-revealing test case, we randomly select one defect out of all of the defects invoked by that test case for removal, which is intended to imitate imperfect debugging in practice. The remaining invoked defects can be considered to be latent defects that should be detected in the upcoming testing stage, which is common in practice. If multiple defects are invoked by a test case, any of these defects can be detected during debugging. In practice, the number of defects that can be localized is affected by the debugging effectiveness, which can vary for different teams. Thus, the approach to imitating imperfect debugging is not unique, and its effect on the experimental results will be discussed in the following paragraphs. The testing process is automatically conducted by our platform until all the defects are removed.

All the experiments in this paper are run on four DELL PRECISION T1500 workstations with a 64-bit Windows 7 operating system. Each workstation is equipped with an i7-870 CPU that has four cores running at 2.93 GHz and contains 4 GB of memory.

B. Testing Strategies and Performance Measurements

In this set of experiments, six different testing strategies are examined: RT, RPT, PSS, ARPT-1, ARPT-2, and an AT strategy (AT-GA-200).

1) RT: The *Random Testing* strategy randomly selects test cases from the entire test suite, which is not partitioned. In this paper, RT only provides a baseline for performance comparison.

2) RPT: The *Random Partition Testing* strategy first randomly selects a test case class, i.e., each test case class has an equal probability of being selected. For example, if a test suite is partitioned into n classes, then every class has a selection probability of $1/n$. Then, a test case is randomly chosen from the selected class and executed.

3) PSS: The original *PSS* aims to allocate test cases proportional to the sizes of the subdomains. However, it is difficult to determine the number of required test cases prior to testing because the experiments aim to detect and remove all the defects. In this paper, we take the idea of the Basic Maximin Algorithm proposed by Chen *et al.* [14], which is described below.

TABLE III
MANN–WHITNEY U TEST p-VALUES

Subject Programs	Sample X Sample Y	RT	RPT	PSS	AT-GA-200
Space-36	ARPT-1	0.048	0.017	0.007	0.997
	ARPT-2-1100	0.565	0.249	0.195	0.155
	AT-GA-200	0.037	0.010	0.003	-
Space-21	ARPT-1	0.048	0.026	0.016	0.828
	ARPT-2-1000	0.228	0.235	0.108	0.186
	AT-GA-200	0.007	0.009	0.001	-
tcas	ARPT-1	0.000	0.000	0.021	0.001
	ARPT-2-150	0.000	0.000	0.036	0.032
	AT-GA-200	0.000	0.000	0.000	-
replace	ARPT-1	0.000	0.028	0.008	0.700
	ARPT-2-300	0.000	0.193	0.053	0.374
	AT-GA-200	0.000	0.044	0.013	-
sed	ARPT-1	0.019	0.048	0.026	0.696
	ARPT-2-100	0.002	0.502	0.002	0.086
	AT-GA-200	0.037	0.009	0.047	-
gzip	ARPT-1	0.036	0.045	0.011	0.058
	ARPT-2-100	0.000	0.000	0.000	0.884
	AT-GA-200	0.000	0.000	0.000	-
flex	ARPT-1	0.017	0.011	0.001	0.851
	ARPT-2-300	0.989	0.910	0.245	0.009
	AT-GA-200	0.008	0.006	0.000	-
SESD	ARPT-1	0.010	0.049	0.001	0.718
	ARPT-2-300	0.189	0.409	0.107	0.194
	AT-GA-200	0.004	0.022	0.000	-

a) The Basic Maximin Algorithm:

1) Set $n_i = 1$ and $\sigma_i = 1/d_i$ for $i = 1, 2, \ldots, k$.
2) Set $q = n - k$.
3) While $q > 0$, repeat the following steps.
    a) Find j that $\sigma_j = \min \sigma_i$.
    b) Set $n_j = n_j + 1$.
    c) Set $\sigma_j = \sigma_j + 1/d_i$.
    d) Set $q = q - 1$.

In this algorithm, n_i is the number of test cases allocated in subdomain D_i, and d_i is the number of all the possible elements in subdomain D_i. In this paper, we set the value of q to be sufficiently large such that the value of q should always be positive before all the defects are detected.

4) ARPT-1: The *Adaptive and Random Partition Testing* #1 strategy uses AT and RPT alternately. The testing lengths for AT and RPT are calculated using the heuristic algorithm described in Section III. In this paper, the initial parameters $k(0)$ and $l(0)$ are both set to 1.

5) ARPT-2: The *Adaptive and Random Partition* testing #2 strategy uses AT to choose the first m test cases and subsequently switches to RPT for the remainder of the testing process. For ARPT-2, the parameter m is set based on our previous knowledge of each subject program. Table III tabulates these parameters. For example, in column two of the third row, ARPT-2-1100 denotes that parameter m in ARPT-2 is set to 1100 for *Space*-36.

6) AT-GA-200: The AT strategy uses a genetic algorithm for parameter estimation by adopting the latest 200 test cases (at most) as the testing history.

Furthermore, AT-GA-200 is used in the ARPT strategies as the AT strategy. However, when the testing strategy switches from AT to RPT in ARPT, the testing history for parameter estimation in AT will be erased, i.e., the AT in every new testing "state" begins with a completely empty testing history.

The parameters of ARPT-1 and ARPT-2 affect the overall effectiveness and efficiency of the ARPT strategies, and thus, they should be carefully chosen by the test engineers based on their knowledge of the SUT and budget constraints (e.g., time limits). However, if such knowledge is not available, the parameters should be arbitrarily assigned at first, and a trial-and-refine procedure should be adopted to obtain suitable parameters during the subsequent testing process. Therefore, a heuristic algorithm is proposed for parameter setting in the ARPT-1 strategy, and the parameters are expected to converge to suitable values as the testing proceeds. In contrast, the ARPT-2 strategy is more similar to the "open-loop" control technique in control engineering in which parameters are preassigned and remain unchanged during the process. In this paper, we empirically choose the value of m to be 50%–80% of the estimated average number of test cases required by AT-GA-200 to detect all the defects in each subject program.

The effectiveness of the testing strategies is measured in terms of the number of test cases required to detect all the defects. In this paper, there are 48 scenarios in total (eight subject programs multiplied by six testing strategies). Because all these strategies are randomized algorithms, the number of trials for each scenario is empirically set to 100 for statistical analysis, and one trial corresponds to the removal of all defects in one subject program.

Moreover, the time efficiency, i.e., the computational overhead incurred by one testing strategy, is measured by recording and comparing the average time consumption per trial $\overline{\Omega}$ (in microseconds) for test case selection. The computational overheads of RT, RPT, and PSS are neglected because they are negligible compared with those of the AT and ARPT strategies. This computational overhead does not include the test case execution time.

C. Experiment Results

Due to the large number of trials, generating boxplots for the subject programs is a suitable method for presenting our results. Fig. 4 shows the boxplots of the experimental results. In each plot, the x-axis corresponds to the various testing strategies, and the y-axis corresponds to the number of test cases executed to detect and remove all defects.

First, for most subject programs, the numbers of test cases required by ARPT-1, ARPT-2, and AT-GA-200 are smaller than those required by RT, RPT, and PSS. The only exception is the *sed* program, in which ARPT-1, ARPT-2, and AT-GA-200 only outperform RPT. Moreover, AT-GA-200 outperforms ARPT-1 and ARPT-2, and ARPT-1 performs slightly better than ARPT-2.

Secondly, the percentage of outliers in these plots can reach 4%, and the data distribution varies. The outliers in the boxplot are typically wrongly recorded data. To avoid such mistakes, we have carefully checked the outlier data in the testing results. After analyzing the records of the testing processes that produce these data, we are confident that these data are not wrongly recorded, and seek the interpretations. It is recognized that the debugging process is far from perfect, and thus not all the invoked defects can be removed because it can be difficult to locate all of them and it is possible to introduce a new defect at the same time. In this paper, when a failure is observed, only one defect is randomly selected from all the invoked defects and is removed to imitate imperfect debugging. Defects typically have different numbers of revealing test cases within the test suite. Thus, when a defect with fewer revealing test cases is the last remaining defect in the program, the testing process will take longer than usual. This situation contributes to the variation of data distribution and the presence of outliers. Because the probability of leaving the difficult-to-reveal defects as the remaining defects varies for different programs, the number of outliers and the data distribution also vary. Indeed, this imitation setting can threaten the experimental validation, which will be further discussed in Section VI.

According to the Kolmogorov–Smirnov test results of the data in Fig. 4, the normal distribution hypothesis is rejected by 28 out of 48 groups of experimental results. Thus, in this paper, we use SPSS to perform the Mann–Whitney U test. The Mann–Whitney U test is a nonparametric test that does not make any assumptions on the data distribution. In Table III, the p-values are tabulated for the Mann–Whitney U-test. The confidence level of this Mann–Whitney U test is 95% ($\alpha = 0.05$), and some p-values are shown as 0.000 because only three significant figures are retained.

In addition, the effect size values are listed in Table IV. In this paper, the effect size in [28] is adopted as follows:

$$\widehat{\mathrm{A}}_{12} = (R_1/m - (m+1)/2)/n \tag{8}$$

where R_1 is the rank sum value of the first sample in the Mann–Whitney U test, and the sizes of the first and second samples are denoted by m and n, respectively. $\widehat{\mathrm{A}}_{12}$ is an unbiased estimate of the stochastic superiority of two populations and is expressed as

$$\mathrm{A}_{12} = P(X > Y) + 0.5P(X = Y). \tag{9}$$

In this paper, A_{12} denotes the superiority of population Y over X because the smaller the number of test cases is, the better the strategy is. Thus, a larger A_{12} indicates a superior population Y. In Table IV, each strategy in the first row is considered to be the sample drawn from population X, and each strategy in column 2 is considered to be the sample drawn from population Y. Based on the value of A_{12}, we obtain the following equation:

$$\mathrm{P}(X > Y) - \mathrm{P}(\mathrm{X} < Y) = 2 * \mathrm{A}_{12} - 1 \tag{10}$$

which indicates the extent to which $X > Y$ occurs more frequently than $X < Y$.

According to Tables III and IV, we conduct the following analysis on the performance of the ARPT strategies.

First, AT-GA-200 and ARPT-1 outperform RT, RPT and PSS in most cases with $p < 0.05$, except for *sed*. For ARPT-2, the results are more complicated. In most cases, ARPT-2 cannot significantly outperform RT, RPT, and PSS simultaneously,

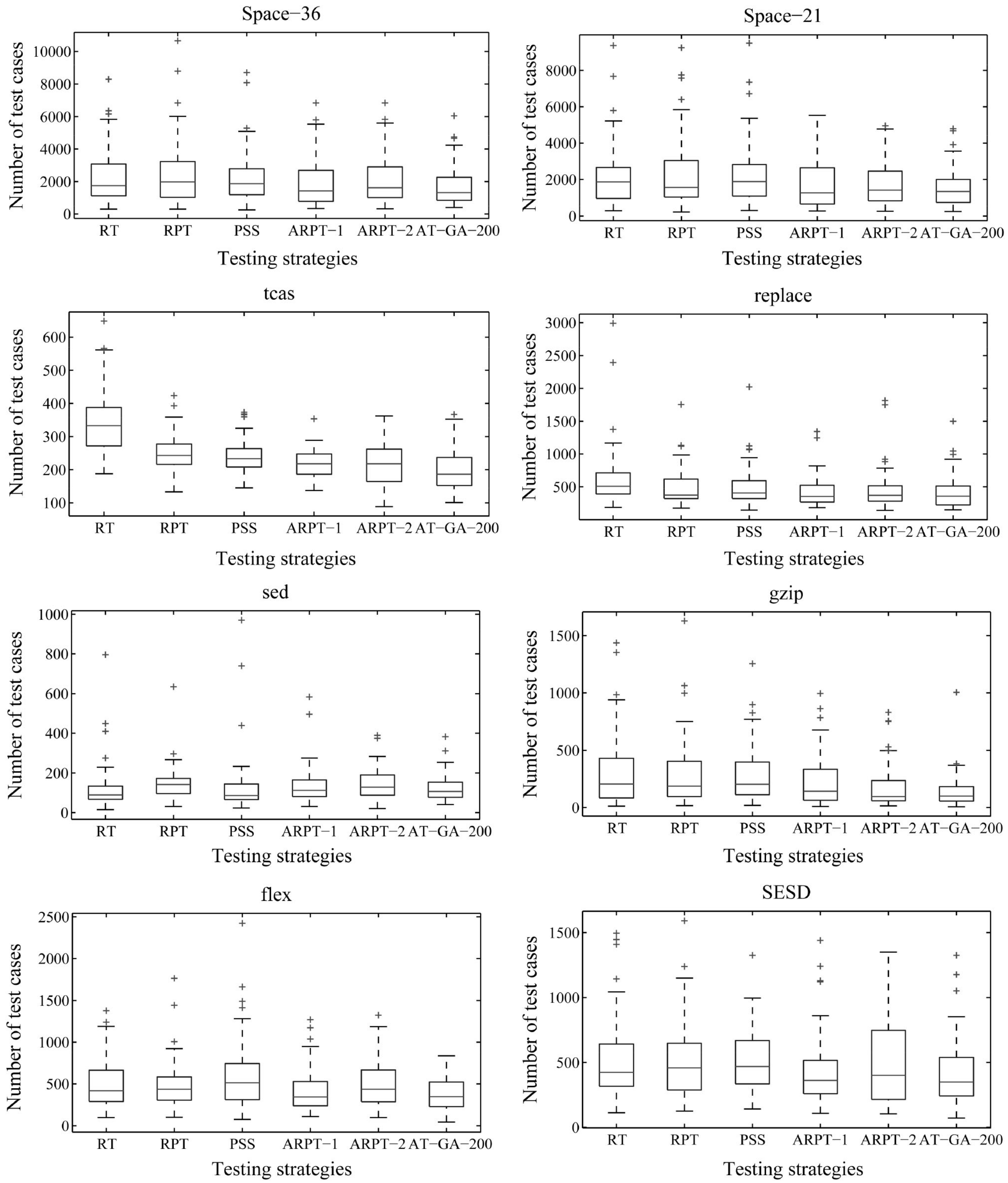


Fig. 4. Boxplots of the experimental results for all subject programs.

except for *tcas* and *gzip*. Although, we choose the *m* value based on our knowledge of the performance of AT-GA-200 for each program, the effectiveness of ARPT-2 can vary considerably. For cases in which AT-GA-200 performs well, ARPT-2 can also perform well, i.e., it provides a lower *p*-value and a greater effect size. Thus, the choice of parameters has a greater impact on the defect detection effectiveness of ARPT-2 than that of ARPT-1, which also indicates that ARPT-1 is more robust than ARPT-2. Although the advantages of AT-GA-200 over RT, RPT, and PSS vary, the performance of ARPT-1 is more stable than that of ARPT-2, which can only achieve acceptable performance when AT-GA-200 has a larger advantage. According to the *p*-values of the Mann–Whitney U test between ARPT-1 and AT-GA-200 and those between ARPT-2

TABLE IV
EFFECT SIZE VALUES

Subject Programs	Sample X / Sample Y	RT	RPT	PSS	AT-GA-200
Space-36	ARPT-1	0.581	0.598	0.611	0.500
	ARPT-2-1100	0.524	0.547	0.553	0.442
	AT-GA-200	0.585	0.606	0.621	-
Space-21	ARPT-1	0.581	0.591	0.599	0.491
	ARPT-2-1000	0.549	0.549	0.566	0.446
	AT-GA-200	0.610	0.606	0.634	-
tcas	ARPT-1	0.910	0.679	0.594	0.358
	ARPT-2-150	0.874	0.653	0.586	0.412
	AT-GA-200	0.919	0.754	0.694	-
replace	ARPT-1	0.702	0.590	0.609	0.484
	ARPT-2-300	0.683	0.553	0.579	0.464
	AT-GA-200	0.696	0.582	0.602	-
sed	ARPT-1	0.404	0.581	0.409	0.484
	ARPT-2-100	0.372	0.527	0.370	0.430
	AT-GA-200	0.415	0.607	0.419	-
gzip	ARPT-1	0.586	0.582	0.604	0.422
	ARPT-2-100	0.648	0.655	0.674	0.494
	AT-GA-200	0.664	0.668	0.694	-
flex	ARPT-1	0.598	0.605	0.641	0.492
	ARPT-2-300	0.499	0.494	0.545	0.393
	AT-GA-200	0.608	0.613	0.655	-
SESD	ARPT-1	0.606	0.581	0.635	0.447
	ARPT-2-300	0.554	0.534	0.566	0.485
	AT-GA-200	0.619	0.594	0.648	-

and AT-GA-200, ARPT-1 is more similar to AT-GA-200 than ARPT-2. Furthermore, the effect of different parameter choices on the performance perturbation of the ARPT strategies is investigated in Section V.

Secondly, the effect size values of ARPT-1 can be comparable with those of AT-GA-200. The effect size values of ARPT-1 are no smaller than 0.581 when compared with RT, RPT, and PSS, except for *sed*. In this case, we have $\mathrm{P}(\mathrm{X} > \mathrm{Y}) - \mathrm{P}(\mathrm{X} < \mathrm{Y}) = 2 \times 0.581 - 1 = 0.162$. This result means that there is at least a 16.2% greater chance that ARPT-1 will outperform RT, RPT, and PSS. With such effect size values, the performance of ARPT-1 is more attractive.

According to the boxplots, the reason why AT statistically outperforms the RT series, i.e., RT, RPT, and PSS, can be addressed. As the boxplots indicate, AT-GA-200 exhibits more compact distributions than the RT series, corresponding to more stable performance for defect detection. The advantage of AT-GA-200 is that it estimates the failure rates according to the testing history and makes an appropriate decision based on the theory of CMC. Because the optimization goal of AT is to minimize the total cost of testing, i.e., the number of required test cases in this paper, the optimal action determined at every step should be based on the parameter estimation. Since the optimal goal should be deterministic if the true values of the parameters are known, AT-GA-200 attempts to converge to this optimization goal in any decision-making process involving the estimated parameters. Thus, AT-GA-200 may have a compact data distribution. Consequently, the ARPT strategies can also exhibit relatively compact distributions.

Additionally, the average computational overhead per trial is also calculated. The detailed data are tabulated in Table V.

Table V shows that the computational overheads of ARPT-1 and ARPT-2 are lower than that of AT-GA-200. The overhead for the test case selection incurred by ARPT-1 is less than 46% of that incurred by AT-GA-200 and is less than 35% in most cases. For ARPT-2, the overhead is less than 67.1% of that incurred by AT-GA-200 and is less than 52% in most cases.

TABLE V
COMPUTATIONAL OVERHEAD INCURRED BY TEST CASE SELECTION

	Computational overhead per trial ($\overline{\Omega}$) (μs)		
	ARPT-1	ARPT-2	AT-GA-200
Space-36	1.32E+07	1.62E+07	3.93E+07
Space-21	1.04E+07	1.42E+07	3.39E+07
tcas	6.51E+05	1.49E+06	2.58E+06
replace	1.31E+06	3.32E+06	6.39E+06
sed	6.33E+05	1.21E+06	2.64E+06
gzip	2.12E+05	3.10E+05	4.62E+05
flex	1.19E+06	3.28E+06	6.35E+06
SESD	1.31E+06	3.22E+06	6.28E+06

It is reasonable to observe such a reduction in computational overhead. For AT-GA-200, the cost of test case selection increases dramatically before 200 testing results are included in the testing history, i.e., the cost incurred by the selection of the next test case increases as additional tests are executed, because a larger testing history is involved in the decision-making process. When the capacity of testing history reaches 200, the cost of the decision-making process for the test case selection stabilizes at a high level.

In applying ARPT-1, the decision-making cost is reduced by two factors. First, the number of test cases employed by RPT is comparable with the number of test cases employed by AT-GA-200 in ARPT-1, whereas the cost of RPT can be negligible compared with that of AT-GA-200. Secondly, in ARPT-1, AT-GA-200 begins with a completely empty testing history in every new testing "state," which reduces the decision-making cost as well. With these two factors, even though additional test cases are required, the average cost incurred by ARPT-1 for each trial is considerably lower than that incurred by AT-GA-200. In applying ARPT-2, the length of AT-GA-200 is fixed, which costs the same as the procedure for a pure AT-GA-200, whereas the remainder of ARPT-2 is RPT, which costs considerably less than the corresponding phase in AT-GA-200.

V. SENSITIVITY ANALYSIS

The sensitivity analysis in this section aims to investigate the performance robustness of the proposed ARPT strategies. Although the experimental results in Tables III–V indicate that the proposed ARPT strategies meet our expectations, the initial parameters are set arbitrarily. Whether ARPT can maintain its robust performance for different initial parameters will be studied experimentally.

A. Parameter Setting

To investigate the robustness of ARPT-1, the initial parameters, i.e., $k(0)$ and $l(0)$, are set from 1 to 10 independently. In this case, there are 100 pairs of $k(0)$ and $l(0)$. For ARPT-2, the initial parameter, i.e., m, is set based on the values given in Section IV. For each subject program, the values of m vary from 50% to 150% of the corresponding value used in Section IV for a step size of 5%. Thus, there are 20 different *m* values for each subject program. To avoid any statistical bias, the number of trails is empirically set to 100 for each setting in this section.

B. Subject Programs and Performance Measurements

Due to time and space limitations, not all the subject programs in Section IV are studied: only four subject programs are considered in this sensitivity analysis, i.e., *Space*-36, *Space*-21, *tcas*, and *replace*. The test suite and partitioning remain the same as in Section IV. In addition, the effect size and computational overhead are used to measure the performance. Since there is no significant difference for most settings at a confidence level of 95% ($\alpha = 0.05$) according to the experimental results in this section, not all the detailed *p*-values are included and only typical results are analyzed instead.

C. Experiment Results

Figs. 5 and 6 show the effect size values and computational overhead values of the sensitivity analysis. The computational overhead values are standardized, which means that the values are divided by the corresponding value of AT-GA-200 in Table V for each subject program, because these values should not be larger than the corresponding value of AT-GA-200. In addition, the effect size values are also calculated against the experimental results in Section IV. That is, the results of ARPT-1 with $k(0) = 1$ and $l(0) = 1$ as well as the results of ARPT-2 with the corresponding setting for each subject program (e.g., ARPT-2-300 for *replace*) are adopted as sample X to calculate the effect size values. For ARPT-2, the values of m are chosen based on the values adopted in Section IV, and thus, the ticks on the x-axis for ARPT-2 correspond to the relative values of m that range from 50% to 150%.

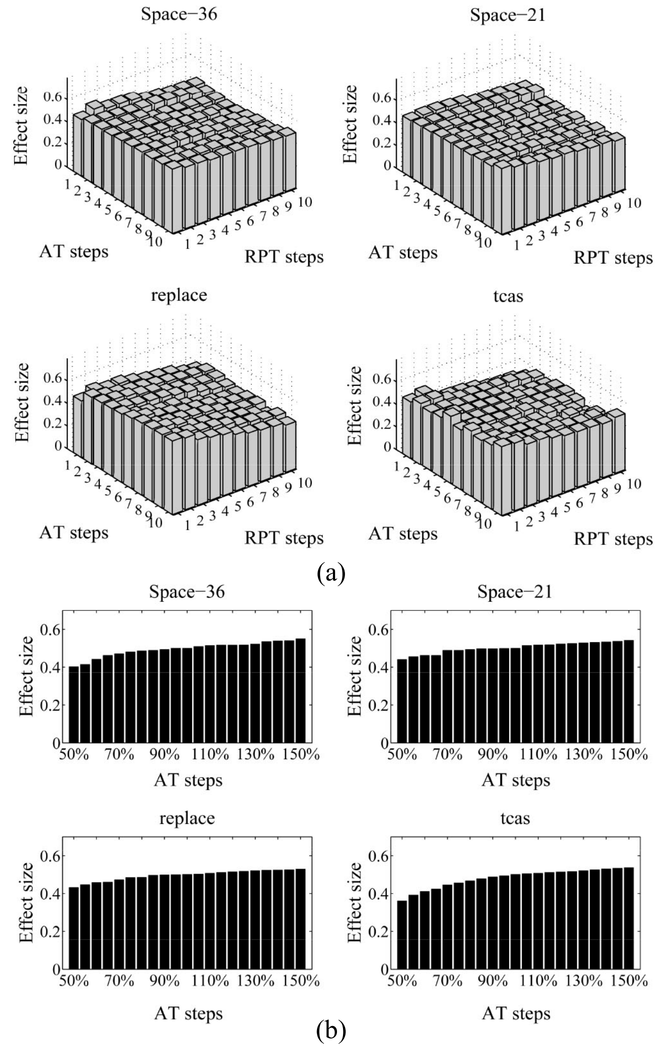


Fig. 5. Effect size values for the sensitivity analysis on (a) ARPT-1 and (b) ARPT-2.

The following observations can be made from Figs. 5 and 6. First, the effect size and computational overhead for ARPT-1 exhibit similar trends. If two initial settings have the same length for RPT, then the setting with more AT steps will have a larger effect size and a higher computational overhead. In addition, if two initial settings have the same length for AT, then the setting with more RPT steps will have a smaller effect size and a lower computational overhead. However, the trends are slightly different for different programs. For example, the skewness of the effect size of ARPT-1 is larger in *replace* than in the other three programs, which implies that the parameter setting for *replace* has a greater impact on the effectiveness of ARPT-1 than those for the other three subject programs.

The sensitivity analysis indicates that the performance of ARPT is robust. For ARPT-1, the effect size values are near 0.5 for most cases, except for the "corner" settings, e.g., $k(0) = 1$, $l(0) = 10$ or $k(0) = 10$, $l(0) = 1$. Thus, the effectiveness of ARPT-1 fluctuates slightly for most parameter settings, and the computational overhead is considerably lower than that of AT-GA-200. These two facts guarantee the stable performance of ARPT-1.

Although the effect size is calculated, there is no significant difference for most cases, in which the effect size is near 0.5. Due to space limitation, the *p*-values for the Mann–Whitney U test are not presented individually. Indeed, there are certain "corner" cases in which the differences are significant. This result indicates that the effectiveness of ARPT-1 can still be affected when the proportion of the AT segment suffers a sufficiently large perturbation. According to the analysis, choosing parameters for ARPT-1 with "mild" proportions for AT ensures both effectiveness and efficiency.

A similar conclusion can be obtained from the analysis results for ARPT-2. In ARPT-2, a larger effect size is obtained if more AT steps are involved. This result means that if a larger proportion of ARPT-2 is an AT strategy, the strategy will be more likely to require fewer test cases to detect all the defects. Thus, the problem of choosing an appropriate m may be alleviated by using the experience of the test engineers. Choosing a relatively large value for m can improve the effectiveness of ARPT-2; however, an inappropriately large value will lead to an unbearable computational overhead (e.g., the results for the 150% settings), and will therefore reduce the efficiency. When the initial value grows sufficiently large, the AT segment can detect all defects such that no RPT segment exists, which leads to a pure AT-GA-200. Determining how to balance the

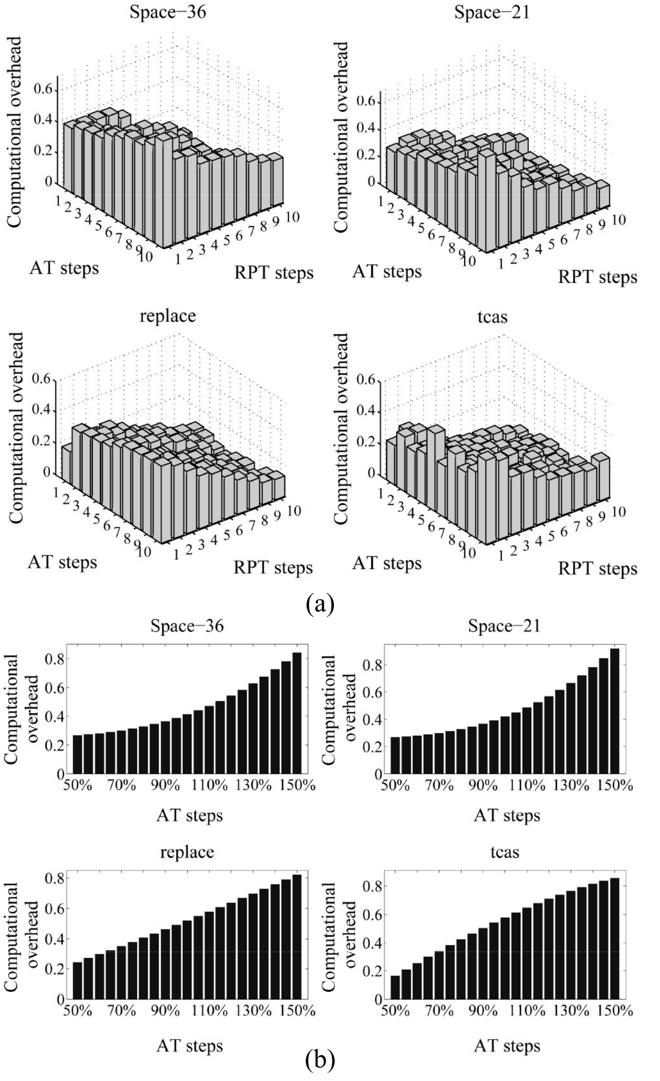


Fig. 6. Computational overhead results for the sensitivity analysis on (a) ARPT-1 and (b) ARPT-2.

effectiveness and efficiency requires additional knowledge and experience about the effectiveness of AT. If there are only a few AT steps, the effectiveness will be affected. For example, the 50% setting for ARPT-2 in *tcas* has a p-value of 0.348 with an effect size of 0.534 when performing the Mann–Whitney U test against the results of PSS, which means that the actual advantage of ARPT-2 in this case is smaller than 0.534 because no significant difference is found. This result is weaker than the effectiveness of ARPT-2 for *tcas* in Tables III and IV. Meanwhile, for *Space*-21, ARPT-2 with a 130% setting significantly outperforms RT (p = 0.005) and incurs an acceptable computational overhead. Using this setting yields better results than using ARPT-2 for *Space*-21 in Table III. Thus, an appropriate choice will result in ARPT-2 taking the advantages of both AT and RPT. We suggest that the effectiveness of AT for the corresponding program should be estimated before using ARPT-2. If the effectiveness of AT is expected to be sufficiently high, then the value of m can be relatively small to avoid wasting computational resources on decision-making; otherwise, the value of m should be sufficiently large (but not overly large) to ensure the effectiveness of ARPT-2 without incurring an unbearable computational overhead.

Overall, the performance of ARPT-1 is robust, and this strategy is recommended if there is insufficient information on the performance of AT to choose an appropriate m for ARPT-2.

VI. General Discussion

A. ARPT Performance

The effectiveness of ARPT is validated by the experiments shown in Tables III and IV. The experimental results indicate that ARPT-1 outperforms RT, RPT, and PSS in seven out of eight subject programs with statistically significant differences at a confidence level of 95%. However, ARPT-2 obtains a "weaker" score because some differences are not highly significant. The ARPT-1 strategy is implemented with additional flexibility because the testing process is adjusted according to the testing results. For ARPT-2, the effectiveness is strongly affected by the empirically chosen parameter m. The performance varies for different subject programs. For example, the settings are more appropriate for *tcas* and *gzip* than for the other subject programs. Although the effect size of ARPT-2 appears to be acceptable for most subject programs, the current advantage does not hold because there is no statistically significant difference. Moreover, even if the differences were statistically significant, the advantage of ARPT-2 would not be comparable with that of ARPT-1.

For the overhead reduction via ARPT, RPT contributes greatly with a nearly zero cost for test case selection. However, the computational overhead incurred by AT increases dramatically with the increasing testing history used for parameter estimation, i.e., adopting fewer testing results in testing history should reduce the computational overhead. The detailed study on the computational overhead incurred by AT can be found in [10].

To investigate the effect of the initial parameter setting on the performance of the ARPT strategies, a sensitivity analysis is conducted in Section V. The experimental results validate the robustness of the ARPT strategies. In ARPT, the performance does not fluctuate greatly when the initial parameter setting varies slightly. However, the extent of the fluctuation varies for different subject programs. In general, as the AT segment takes additional test cases, the ARPT strategies become less efficient but more effective. However, particular attention should be paid to the parameter setting, especially for ARPT-2, because the effectiveness and efficiency should be considered simultaneously, and the performance of ARPT-2 can be significantly affected by the choice of m. A value of m that is too small will lead to poor effectiveness, whereas an excessive m value will lead to poor efficiency. Thus, m should be carefully chosen. The choice of m should be based on the knowledge of the ATs performance and the subject program. Choosing a value that is close to but not overly close to the average performance of AT may be a safe approach.

B. Motivation and General Concerns for ARPT

The motivation for proposing ARPT strategies is two-fold. First, we intend to utilize complex testing strategies (e.g., AT) to improve defect detection effectiveness; additionally, we intend to avoid the unbearable computational overhead incurred by frequent execution of such advanced testing strategies by mixing them with less computationally complex strategies (e.g., RT and DRT [7], [8]). Secondly, different testing strategies have different favorable conditions under which their performance can be fully exploited. For example, AT-GA quickly adapts the selection of a test case class after defect removal, whereas DRT adjusts the test profile during the testing process without defect removal. A natural idea is to combine these two strategies: AT-GA is used to adjust the test profile when the defect is removed, and DRT is used for the remaining tests until the next defect removal.

However, there are other performance metrics for testing strategies (e.g., coverage information), and defect detection effectiveness is only one of these metrics. The ARPT strategies are proposed to improve defect detection effectiveness. More importantly, these testing strategies can balance testing effectiveness and efficiency in a sufficiently generalized manner such that this method can be used to solve many other similar problems.

The general concern for the performance of ARPT is the problem of how to set relevant parameters. Indeed, the improvement in the defect detection effectiveness primarily depends on the advantage offered by the AT strategy [3], [9], [20], [29]. However, the reduction of decision-making cost is determined by both RPT and AT. In this paper, a heuristic method is proposed, and an example is implemented for ARPT-1. Although the optimality cannot be proved theoretically, the experimental results indicate that this algorithm can perform well under different scenarios. This method should not be the only method used in practice, and it is unlikely to be the optimal choice. Future research should focus on the use of heuristic methods. For ARPT-2, the problem appears to be more complicated. We set the parameter m according to our experience; however, this approach does not work consistently on the subject programs, and the effectiveness of ARPT-2 fluctuates more strongly than that of ARPT-1. Compared with ARPT-1, implementing ARPT-2 requires additional a priori experience, and its performance strongly depends on such experience.

C. Threats to Validity

The following describes several potential threats to the validity of the experimental study and provides details on how these threats are addressed.

First, threats to construct validity in this paper focus on how the performance of a testing strategy is defined. The primary focus is on defect detection effectiveness and time efficiency, and indeed, there are more metrics in software testing than these two metrics. However, AT improves the testing effectiveness at a high cost, which can make it inapplicable to certain testing scenarios in practice. Thus, in this paper, we focus on effectiveness and efficiency to pursue a balance such that the testing effectiveness can be improved without producing an unbearable overhead. Moreover, the hybrid idea proposed in this paper can also be used to solve other testing problems with different metrics, e.g., code coverage, by combining two or more different strategies with different advantages to achieve a balance. This strategy should be investigated in future research.

Secondly, threats to internal validity may stem from how the experiments are designed without bias. In this paper, the testing process is automatically conducted by an own-developed testing platform with the same subject programs and test suites for different testing strategies. To avoid possible defects in the testing platform itself, we have carefully tested the platform and the implemented testing strategies. Additionally, the test suites are partitioned either randomly or based on functionality. Because the comparison between RT and PT will involve the cost of partitioning the test suite, we focus on the results of PT-based strategies, i.e., RPT, PSS, AT, and ARPT, and RT only provides a baseline for performance comparison. Another threat that may lead to bias is the imperfect debugging. In practice, it is usually impossible to remove all the defects by debugging, which may also introduce new defects. Therefore, we consider the imperfect debugging in this paper. Since the introduction of new defects will lead to an unpredictable bias in the experimental design, we choose to randomly remove one defect out of all the defects invoked by the failure-revealing test case to imitate imperfect debugging. Other choices could be used to determine latent defects, e.g., removing more defects at a time or removing different defects with different probabilities. However, all these choices lead to different levels of bias because the debugging effectiveness can vary considerably in practice. The choice in this paper is relatively fair because the removed defect is randomly determined. The effect of "imperfect" debugging on the performance of the testing strategies will be investigated in future work. Since all the testing strategies are randomized, we adopt the nonparametric Mann–Whitney U test and the effect size to perform a statistical analysis of the testing results based on the fact that the normality hypothesis is rejected by more than half of the experimental results, which means that the use of a common parametric statistical test could be risky. Thus, using a nonparametric test and the corresponding effect size can provide statistical significance and confidence in the results.

Thirdly, the external validity for any empirical study is the generalization of the obtained results. It is impossible to sample a sufficiently large set of subject programs to capture all the types of defect distributions within the programs due to the rich and diverse nature of the programs. However, we use a variety of programs with different characteristics in this paper. These subject programs have different origins with different numbers of LOC, defects, and test suite partitions. These programs represent a large variety of relationships between defect distribution and test suite partition, and the performance of the AT strategy can be affected by such relationships. For subject programs with similar relationships to those used in this paper, we can have more confidence in the generalization, but care should be taken until any knowledge of such relationships is confirmed.

As with all experimental studies, further experiments on more subject programs are required. However, the results

TABLE VI
HIERARCHY OF TESTING TECHNIQUES

Levels	Objective	Current Testing Techniques
Organization Level	Choose the initial test profile	Adaptive Testing Partition Testing Proportional Sampling
Coordination Level	Dynamic adjustment of the test profile	Dynamic Random Testing Dynamic Partitioning
Execution Level	Choose test cases according to the test profile	Random Testing Adaptive Random Testing

indicate that ARPT strategies, especially ARPT-1, can indeed improve the testing effectiveness without incurring an unreasonable computational overhead, which increases the feasibility of using these strategies in testing scenarios with computational constraints.

D. Future Research Topics

The general approach of the ARPT strategies proposed in this paper is similar to that of hierarchical control techniques widely adopted in smart machine control, which is designed to organize, coordinate, and execute anthropomorphic tasks by a machine for minimum interaction with a human operator. For software testing, a similar classification can be instituted to divide the testing process into three levels, i.e., test organization, test coordination, and test execution. A specific testing strategy should be adopted at each level to improve the defect detection effectiveness.

More specifically, the organization level involves SUT version progression due to defect removal in which a testing strategy is adopted to determine the initial allocation of testing resources (test profile) to different functional modules to incorporate the changes introduced in the debugging process. The coordination level involves dynamic adjustment of the test profile when increasing numbers of testing results are obtained during the testing process, e.g., failures and successes. The corresponding testing strategy is required to quickly respond to the feedback from the test execution and incur as little computational overhead as possible to conduct a reallocation of testing resources. At the execution level, test cases are chosen based on the test profile. Following the principle of partitioning, the chosen test cases from each class should be representative of that class. Therefore, the testing strategy should select a test case according to the test profile given by the upper levels and attempt to achieve a homogeneous distribution of test cases within each test case class. Table VI provides the classification of several current testing techniques according to the above hierarchy.

VII. Conclusion and Future Work

RT and RPT are efficient testing strategies with respect to the computational overhead incurred by test case selection. However, their disadvantages lie in the defect detection effectiveness. AT has been shown to be more effective than RT and RPT. However, a major concern is that the application of AT can be infeasible in certain cases because its complicated test case selection scheme results in low time efficiency.

To achieve a balance between testing effectiveness and efficiency, in this paper, we proposed a hybrid approach using AT and RPT, which is known as the ARPT strategy. The motivation for this approach is to combine both strategies such that the underlying computational complexity of AT is reduced by introducing RPT into the testing process without affecting the defect detection effectiveness. Two variants for ARPT are proposed, i.e., ARPT-1 and ARPT-2, and a heuristic method is implemented to solve the parameter-setting problem involved in ARPT-1. To validate the new ARPT strategies, experiments are conducted on seven real-life programs. The experimental results indicate that compared with the original AT strategy, the defect detection effectiveness is maintained by the ARPT strategies; at the same time, the ARPT strategies considerably reduce the computational overhead for test case selection. In addition, the sensitivity analysis on ARPT shows that both the ARPT strategies are robust to a certain extent. ARPT-1 exhibits better performance for different subject programs, whereas ARPT-2 requires considerable knowledge of ATs performance to achieve an acceptable overall performance. Therefore, ARPT-1 is recommended over ARPT-2 due to the higher and more robust performance of ARPT-1.

The general methodology of the ARPT strategies proposed in this paper also represents a novel paradigm. This paradigm can be adopted not only for testing effectiveness and time efficiency but also for many other tradeoff problems in testing. For example, the testing process should occasionally be stopped for product delivery. Thus, the decision on when to stop testing is typically determined as a tradeoff among the budget, available time, and product quality. Techniques that provide high quality data, e.g., more accurate software reliability estimates, may require additional budgets and more time. Thus, different testing strategies with various costs and guarantees of the software quality can be combined to provide a desired outcome. A hybrid method that combines two strategies with different advantages may effectively cancel the drawbacks of these two strategies without affecting their advantages.

Future work on ARPT should include different heuristic methods and examples of these methods, a theoretical analysis on balancing defect detection effectiveness and testing efficiency, and optimal parameter settings for different subject programs.

Acknowledgment

A brief abstract describing a portion of this paper was presented at the 22nd Annual International Symposium on Software Reliability Engineering (ISSRE 2011) [11].

References

[1] J. W. Duran and S. C. Ntafos, “An evaluation of random testing,” *IEEE Trans. Softw. Eng.*, vol. 10, no. 4, pp. 438–444, Jul. 1984.

[2] W. J. Gutjahr, “Partition testing vs. random testing: The influence of uncertainty,” *IEEE Trans. Softw. Eng.*, vol. 25, no. 5, pp. 661–674, Sep./Oct. 1999.

[3] K. Y. Cai, B. Gu, H. Hu, and Y. C. Li, “Adaptive software testing with fixed-memory feedback,” *J. Syst. Softw.*, vol. 80, no. 8, pp. 1328–1348, Aug. 2007.

[4] K. Y. Cai, T. Jing, and C. G. Bai, "Partition testing with dynamic partitioning," in *Proc. 29th Annu. Int. COMPSAC*, Edinburgh, U.K., 2005, pp. 113–116.

[5] T. Y. Chen and R. G. Merkel, "An upper bound on software testing effectiveness," *ACM Trans. Softw. Eng. Meth.*, vol. 17, no. 3, Article 16, Jun. 2008.

[6] K. P. Chan, T. Y. Chen, and D. Towey, "Restricted random testing," in *Proc. 7th ECSQ*, Berlin, Germany, 2002, pp. 321–330.

[7] K. Y. Cai, H. Hu, C. H. Jiang, and F. Ye, "Random testing with dynamically updated test profile," in *Proc. 20th ISSRE*, Mysore, Karnataka, India, Nov. 2009.

[8] J. P. Lv, H. Hu, and K. Y. Cai, "A sufficient condition for parameters estimation in dynamic random testing," in *Proc. IEEE 35th COMPSACW*, Munich, Germany, 2011, pp. 19–24.

[9] K. Y. Cai, "Optimal software testing and adaptive software testing in the context of software cybernetics," *Inf. Softw. Tech.*, vol. 44, no. 14, pp. 841–855, Nov. 2002.

[10] F. Ye, C. Liu, H. Hu, C. H. Jiang, and K. Y. Cai, "On the computational complexity of parameter estimation in adaptive testing strategies," in *Proc. 15th IEEE PRDC*, Shanghai, China, 2009, pp. 87–92.

[11] H. Hu, J. P. Lv, K. Y. Cai, and T. Y. Chen, "A case study of hybrid adaptive testing strategies," in *Proc. IEEE 22nd ISSRE*, Hiroshima, Japan, 2011.

[12] P. G. Frankl, R. G. Hamlet, B. Littlewood, and L. Strigini, "Evaluating testing methods by delivered reliability," *IEEE Trans. Softw. Eng.*, vol. 24, no. 8, pp. 586–601, Aug. 1998.

[13] E. J. Weyuker and B. Jeng, "Analyzing partition testing strategies," *IEEE Trans. Softw. Eng.*, vol. 17, no. 7, pp. 703–711, Jul. 1991.

[14] T. Y. Chen, T. H. Tse, and Y. T. Yu, "Proportional sampling strategy: A compendium and some insights," *J. Syst. Softw*., vol. 58, no. 1, pp. 65–81, Aug. 2001.

[15] S. Morasca and S. Serra-Capizzano, "On the analytical comparison of testing techniques," in *Proc. ACM SIGSOFT ISSTA*, Boston, MA, USA, 2004, pp. 154–164.

[16] T. Y. Chen, H. Leung, and I. K. Mak, "Adaptive random testing," in *Proc. 9th Asian Comput. Sci. Conf.*, Berlin, Germany, 2005, pp. 320–329.

[17] A. Shahbazi, A. F. Tappenden, and J. Miller, "Centroidal Voronoi tessellations—A new approach to random testing," *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, pp. 163–183, Feb. 2013.

[18] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proc. 29th Int. Conf. Softw. Eng.*, Minneapolis, MN, USA, 2007, pp. 75–84.

[19] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Proc. ACM SIGPLAN Conf. PLDI*, Chicago, IL, USA, 2005, pp. 213–223.

[20] H. Hu, W. E. Wong, C. H. Jiang, and K. Y. Cai, "A case study of the recursive least squares estimation approach to adaptive testing for software components," in *Proc. 5th QSIC*, Melbourne, VIC, Australia, 2005, pp. 135–141.

[21] K. Y. Cai, C. H. Jiang, H. Hu, and C. G. Bai, "An experimental study of adaptive testing for software reliability assessment," *J. Syst. Softw.*, vol. 81, no. 8, pp. 1406–1429, Dec. 2007.

[22] H. Hu, C. H. Jiang, and K. Y. Cai, "An improved approach to adaptive testing," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 19, no. 5, pp. 679–705, Aug. 2009.

[23] H. Pham, *System Software Reliability*. London, U.K.: Springer, 2006.

[24] W. J. Van der Linden and C. A. W. Glas, *Computerized Adaptive Testing: Theory and Practice*. Boston, MA, USA: Kluwer, 2000.

[25] H. Wainer *et al.*, *Computerized Adaptive Testing*: *A Primer*. Hillsdale, NJ, USA: Lawrence Erlbaum Associates, 1990.

[26] O. Hernandez-Lerma, *Adaptive Markov Control Processes*. Secaucus, NJ, USA: Springer-Verlag, 1989.

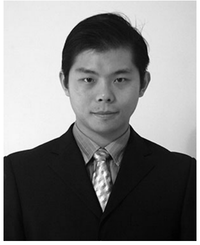
[27] C. Liu, L. Fei, X. Yan, J. W. Han, and S. P. Midkiff, "Statistical debugging: A hypothesis testing-based approach," *IEEE Trans. Softw. Eng.*, vol. 32, no. 10, pp. 831–848, Oct. 2006.

[28] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of McGraw and Wong," *J. Educ. Behav. Stat*., vol. 25, no. 2, pp. 101–132, 2000.

[29] H. Hu, C. H. Jiang, and K. Y. Cai, "Adaptive software testing in the context of an improved controlled Markov chain model," in *Proc. 32nd Annu. IEEE Int. COMPSAC*, Turku, Finland, 2008, pp. 853–858.

Junpeng Lv received the B.S. degree from Beihang University, Beijing, China, in 2009, and is currently pursuing the Ph.D. degree from the same university.

His current research interests include software testing and software reliability assessment.



Hai Hu was born in August 1983. He received the B.S. and Ph.D. degrees from Beihang University, Beijing, China, in 2004 and 2012, respectively.

He was a Visiting Scholar with the University of Texas at Dallas, Dallas, TX, USA, from 2005 to 2006. He is currently an Entrepreneur with enterprise mobilization and cloud computing. His current research interests include software testing and software cybernetics.

Kai-Yuan Cai received the B.S., M.S., and Ph.D. 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 current research interests include software testing, software reliability, reliable flight control, and software cybernetics.

Tsong Yueh Chen (M'03) received the B.Sc. and M.Phil. degrees from the University of Hong Kong, DIC, Hong Kong, and the M.Sc. degree from Imperial College, University of London, London, U.K., and the Ph.D. degree from the University of Melbourne, VIC, Australia.

He is currently a Professor of Software Engineering with Swinburne University of Technology, Hawthorn, VIC, Australia. Prior to joining Swinburne University of Technology, he taught at the University of Hong Kong and the University of Melbourne. His current research interests include software testing and debugging.