Queries to the Author

When you submit your corrections, please either annotate the IEEE Proof PDF or send a list of corrections. Do not send new source files as we do not reconvert them at this production stage.

Q1. Please check and confirm whether the name of the authors Chang-Ai Sun and Tsong Yueh Chen in the byline is correct as set.
Q2. There was a discrepancy in the PDF and the source file. We have followed the source file.
Q3. Please provide complete bibliography details for Ref. [16].
Q4. Please provide page range for Ref. [41].

Dynamic Random Testing of Web Services: A Methodology and Evaluation

Chang-~~Ai~~Sun, *Senior Member, IEEE*, Hepeng Dai, Guan Wang, Dave Towey, *Member, IEEE*, Tsong Yueh Chen, *Member, IEEE*, and Kai-Yuan Cai

Abstract—In recent years, service oriented architecture (SOA) has been increasingly adopted to develop distributed applications in the context of the Internet. To develop reliable SOA-based applications, an important issue is how to ensure the quality of web services. In this article, we propose a dynamic random testing (DRT) technique for web services, which is an improvement over the widely-practiced random testing (RT) and partition testing (PT) approaches. We examine key issues when adapting DRT to the context of SOA, including a framework, guidelines for parameter settings, and a prototype for such an adaptation. Empirical studies are reported where DRT is used to test three real-life web services, and mutation analysis is employed to measure the effectiveness. Our experimental results show that, compared with the three baseline techniques, RT, Adaptive Testing (AT) and Random Partition Testing (RPT), DRT demonstrates higher fault-detection effectiveness with a lower test case selection overhead. Furthermore, the theoretical guidelines of parameter setting for DRT are confirmed to be effective. The proposed DRT and the prototype provide an effective and efficient approach for testing web services.

Index Terms—Software testing, random testing, dynamic random testing, web service, service oriented architecture

1 Introduction

Service oriented architecture (SOA) [1] defines a loosely coupled, standards-based, service-oriented application development paradigm in the context of the Internet. Within SOA, three key roles are defined: service providers (who develop and own services); service requestors (who consume or invoke services); and a service registry (that registers services from providers and returns services to requestors). Applications are built upon services that present functionalities through publishing their interfaces in appropriate repositories, abstracting away from the underlying implementation. Published interfaces may be searched by other services or users, and then invoked. Web services are the realization of SOA based on open standards and infrastructures [2]. Ensuring the reliability of SOA-based applications can become critical when such applications implement important business processes.

Software testing is a practical method for ensuring the quality and reliability of software. However, some SOA features can pose challenges for the testing of web services [3], [4]. For instance, service requestors often do not have access to the source code of web services which are published and owned by another organization, and, consequently, it is not possible to use white-box testing techniques. Testers may, therefore, naturally turn to black-box testing techniques.

Random Testing (RT) [5] is one of the most widely-practiced black-box testing techniques. Because test cases in RT are randomly selected from the input domain (which refers to the set of all possible inputs of the software under test), it can be easy to implement. Nevertheless, because RT does not make use of any information about the software under test (SUT), or the test history, it may be inefficient in some situations. In recent years, many efforts have been made to improve RT in different ways [6], [7], [8]. Adaptive random testing (ART) [7], [9], for example, has been proposed to improve RT by attempting to have a more diverse distribution of test cases in the input domain.

In contrast to RT, partition testing (PT) attempts to generate test cases in a more "systematic" way, aiming, to use fewer test cases to reveal more faults. When conducting PT, the input domain of the SUT is divided into disjoint partitions, with test cases then selected from each and every one. Each partition is expected to have a certain degree of homogeneity—test cases in the same partition should have similar software execution behavior. Ideally, a partition should also be homogeneous in fault detection: If one input can reveal a fault, then all other inputs in the same partition should also be able to reveal a fault.

RT and PT are based on different intuitions, and each have their own advantages and disadvantages. Because it is likely that they can be complementary to each other, detecting different faults, it is intuitively appealing to investigate their integration in random partition testing (RPT).

- C.-A. Sun, H. Dai, and G. Wang are with the School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing 100083, China. E-mail: casun@ustb.edu.cn, daihepeng@sina.cn, jayjaywg@qq.com.
- D. Towey is with the School of Computer Science, University of Nottingham Ningbo China, Ningbo 315100, China. E-mail: dave.towey@nottingham.edu.cn.
- T.Y. Chen is with the Department of Computer Science and Software Engineering, Swinburne University of Technology, Hawthorn, VIC 3122, Australia. E-mail: 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.

Manuscript received 22 Jan. 2019; revised 26 Oct. 2019; accepted 10 Dec. 2019. Date of publication 0 . 0000; date of current version 0 . 0000.
(Corresponding author: Chang-~~Ai~~Sun.)
Digital Object Identifier no. 10.1109/TSC.2019.2960496

In traditional RPT [6], the partitions and corresponding test profiles remain constant throughout testing, which may not be the best strategy. Independent researchers have observed that fault-revealing inputs tend to cluster into "continuous regions" [10], [11]—there is similarity in the execution behavior of neighboring software inputs. Based on software cybernetics, Cai *et al.* proposed adaptive testing (AT) to control the testing process [12], however, AT's decision-making incurs an extra computational overhead. To alleviate this, dynamic random testing (DRT) [6] was proposed by Cai *et al.*, aiming to improve on both RT and RPT.

In practice, web services have usually been tested by the service providers, and simple or easy-to-test faults have been removed, meaning that the remaining faults are normally hard to detect. For ensuring a higher reliability of the web services, a simple RT strategy may not be appropriate [13], especially when the scale is large, or there are some stubborn faults.

In this paper, we present a DRT approach for web services, as an enhanced version of RT adapted to the context of SOA. We examine key issues of such an adaptation, and, accordingly, propose a framework for testing web services that combines the principles of DRT [6] and the features of web services. To validate the fault detection effectiveness and efficiency of the proposed DRT method in the context of SOA, we conduct a comprehensive empirical study. We also explore the impact factors of the proposed DRT, and provide guidelines for setting DRT parameters based on a theoretical analysis. Finally, we compare the performance of the proposed DRT with other baseline techniques.

This paper extends our previous work [14] in the following aspects. First, this paper extensively examines the challenges and practical situations related to testing web services (Section 2.2). It also extensively discusses the limitations of RT, PT, RPT, and AT, when they are used for testing web services (Section 1). Second, although previous work [14] provided a coarse-grained framework for DRT of web services, PT was not studied. In contrast, this paper provides a comprehensive solution based on partitioning (Section 4.4.1). Third, based on a theoretical analysis (Section 3.2), this paper provides guidelines for setting DRT parameters. Such guidelines are crucial to enhance the practical application of DRT, which was not covered in previous work [14]. Fourth, previous work [14] only evaluated the fault detection effectiveness and efficiency of the proposed approach (DRT) in terms of the F-measure and T-measure, and only two small web services (ATM Service and Warehouse Service) were used in the evaluation of its performance. This paper, in contrast, provides a comprehensive evaluation that not only evaluates the fault detection effectiveness of the proposed approach in terms of the F-measure, F2-measure, and T-measure (Section 5.1), but also evaluates its efficiency in terms of F-time, F2-time, and T-time (Section 5.3). Furthermore, we also examine three real-life web services, comparing the fault-detection effectiveness and efficiency of the proposed approach with those of RT, RPT, and AT. Statistical analysis is used to validate the significance of the empirical evaluations and comparisons (Sections 5.1 and 5.3), which was not covered in previous work [14]. Extending again the previous work [14], we also examine the relationship between the number of partitions and the optimal control parameter settings for DRT, evaluating the usefulness of guidelines provided by the theoretical analysis (Section 5.2). The contributions of this work, combined with previous work [14], include:

- We develop an effective and efficient testing method for web services. This includes a DRT framework that addresses key issues for testing web services, and a prototype that partly automates the framework.
- We evaluate the performance of DRT through a series of empirical studies on three real web services. These studies show that DRT has significantly higher fault-detection efficiency than RT and RPT. That is, to detect a given number of faults, DRT uses less time and fewer test cases than RT and RPT.
- We provide guidelines for the DRT parameter settings, supported by theoretical analysis, and validated by the empirical studies.

The rest of this paper is organized as follows. Section 2 introduces the underlying concepts for DRT, and web services. Section 3 presents the DRT framework for web services, guidelines for its parameter settings, and a prototype that partially automates DRT. Section 4 describes an empirical study where the proposed DRT is used to test three real-life web services, the results of which are summarized in Section 5. Section 6 discusses related work and Section 7 concludes the paper.

2 Background

In this section, we present some of the underlying concepts for DRT, and web services.

2.1 Dynamic Random Testing (DRT)

DRT combines RT and PT, with the goal of benefitting from the advantages of both. Given a test suite TS classified into m partitions (denoted $s_1, s_2, \ldots, s_m$), suppose that a test case from s_i ($i = 1, 2, \ldots, m$) is selected and executed. If this test case reveals a fault, $\forall j = 1, 2, \ldots, m$ and $j \neq i$, we then set

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

where ε is a probability adjusting factor, and then

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

Alternatively, if the test case does not reveal a fault, we set

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

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

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

Algorithm 1 describes DRT. In DRT, the first test case is taken from a partition that has been randomly selected according to the initial probability profile $\{p_1, p_2, \ldots, p_m\}$

(Lines 2 and 3 in Algorithm 1). After each test case execution, the test profile $\{\langle s_1, p_1\rangle, \langle s_2, p_2\rangle, \ldots, \langle s_m, p_m\rangle\}$ is updated by changing the values of p_i: If a fault is revealed, Formulas (1) and (2) are used; otherwise, Formulas (3) and (4) are used. The updated test profile is then used to guide the random selection of the next test case (Line 8). This process is repeated until a termination condition is satisfied (Line 1). Examples of possible termination conditions include: "testing resources have been exhausted"; "a certain number of test cases have been executed"; and "a certain number of faults have been detected".

Algorithm 1. DRT

Input: $\varepsilon, p_1, p_2, \ldots, p_m$
1: **while** termination condition is not satisfied
2: Select a partition s_i according to the testing profile $\{\langle s_1, p_1\rangle, \langle s_2, p_2\rangle, \ldots, \langle s_m, p_m\rangle\}$.
3: Select a test case t from s_i.
4: Test the software using t.
5: **if** a fault is revealed by t **then**
6: Update p_j ($j = 1, 2, \ldots, m$ and $j \neq i$) and p_i according to Formulas 1 and 2.
7: **else**
8: Update p_j ($j = 1, 2, \ldots, m$ and $j \neq i$) and p_i according to Formulas 3 and 4.
9: **end if**
10: **end while**

As can be seen from Formulas (1) to (4), updating the test profile involves m simple calculations, thus requiring a constant time. Furthermore, the selection of partition s_i, and subsequent selection and execution of the test case, all involve a constant time. The execution time for one iteration of DRT is thus a constant, and therefore the overall time complexity for DRT to select n test cases is $O(m \cdot n)$.

2.2 Web Services

A web service is a platform-independent, loosely coupled, self-contained, programmable, web-enabled application that can be described, published, discovered, coordinated and configured using XML artifacts for the purpose of developing distributed interoperable applications [1]. A web service consists of a description (usually specified in WSDL) and implementation (written in any programming language). Web services present their functionalities through published interfaces, and are usually deployed in a service container. Invocation of a web service requires analysis of the input message in its WSDL, test data generation based on its input parameters, and wrapping of test data in a SOAP message.

A web service is a basic component of SOA software, and, accordingly, the reliability of such SOA software depends heavily on the quality of the component web services. While testing is an obvious potential activity to help assuring the quality of web services, due to the unique features of SOA, web service testing can be more challenging than traditional software testing [4]. Some of these features include:

- *Lack of access to service implementation:* Normally, web service owners will not make the source code of their web services accessible. Typically, service users only have access to the service interface defined in a WSDL file, which means that white-box testing approaches are not possible.
- *Incomplete documentation or specification:* A service provider may only offer an incomplete or inaccurate description of a service's functional and non-functional behavior. This makes it difficult to decide whether or not a test passes, especially when details about behavior or restrictions on implementations are missing [15].
- *Lack of control:* Unlike traditional software testing where testers can control the execution of the software under test, there is usually no opportunity to intervene in the execution of the web service under test, which is often deployed in a remote service container.
- *Side effects caused by testing:* A large number of tests may introduce an additional communication load, and hence impact on the performance of the web service under test. This suggests that the number of tests should be kept as low as possible [16].

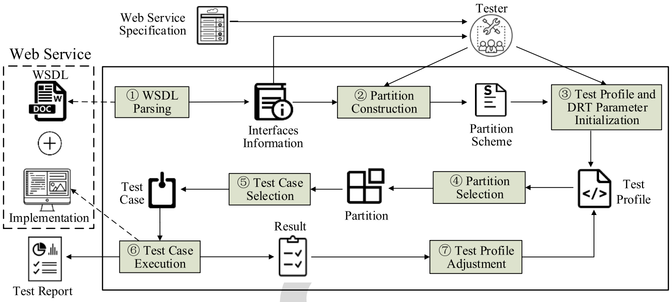


Fig. 1. DRT for web services framework.

3 DRT for Web Services

In this section, we describe a framework for applying DRT to web services, discuss guidelines for setting DRT's parameters, and present a prototype that partially automates DRT for web services.

3.1 Framework

Considering the principles of DRT and the features of web services, we propose a DRT for web services framework, as illustrated in Fig. 1. In the figure, the DRT components are inside the box, and the web services under test and testers are located outside. Interactions between DRT components, the web services, and testers are depicted in the framework. We next discuss the individual framework components.

1) *WSDL Parsing*. Web services are composed of services and the relevant WSDL documents. By parsing the WSDL document, we can get the input information for each operation in the services. This includes the number of parameters, their names and types, and any additional requirements that they may have.
2) *Partition Construction*. Partition testing (PT) refers to a class of testing techniques that classify the input domain into a number of partitions [17]. Because DRT is a black-box testing technique, combining RT and PT, the PT approaches used are at the specification level. Various approaches and principles for achieving convenient and effective partitions have been discussed in

the literature [17], [18]. The input domain of the web service under test (WSUT) can be partitioned based on the WSUT specifications and the parsed parameters. Once partitioned, testers can assign probability distributions to the partitions as an initial testing profile. This initial testing profile can be assigned in different ways, including using a uniform probability distribution, or one that sets probabilities according to the importance of the partition: For example, a partition within which faults were previously detected should be given higher priority.

3) *Test Profile and DRT Parameter Initialization*. Testers need to initialize the test profile, a simple way of doing which would be the use of a uniform probability distribution ($p_1 = p_2 = \ldots = p_k = 1/k$, where k denotes the number of partitions, and p_i $(i = 1, 2, \ldots, k)$ denotes the probability of selecting the ith partition). They also need to set the DRT parameters (guidelines for which are introduced in Section 3.2).
4) *Partition Selection*. DRT randomly selects a partition s_i according to the test profile.
5) *Test Case Selection*. DRT selects a test case from the selected partition s_i according to a uniform distribution.
6) *Test Case Execution*. The relevant DRT component receives the generated test case, converts it into an input message, invokes the web service(s) through the SOAP protocol, and intercepts the test results (from the output message).
7) *Test Profile Adjustment*. Upon completion of each test, its pass or fail status is determined by comparing the actual and expected results (a pass status if both are the same). The pass or fail status is then used to adjust the (partition) probability distribution accordingly. Situations where determination of the test outcome status is not possible (i.e., in the presence of the oracle problem [19], [20]) may potentially be addressed using metamorphic testing [21].

Generally speaking, DRT test case generation is influenced by both the probability distribution (for selection of the relevant partition), and the principles of RT—combining the effectiveness of PT with the ease of RT. Because our technique is based on PT, it is necessary that the partition details be provided (by the tester), which can easily be done through analysis of the input parameters and their constraints, as described in the specification of the web service under test. Once the partition details are available, then it is not difficult to set an initial test profile. The tester can, for example, simply use a uniform probability distribution ($p_1 = p_2 = \ldots = p_m = 1/m$, where m denotes the number of partitions, and p_i $(i = 1, 2, \ldots, m)$ denotes the probability of selecting the ith partition). In Section 3.2, we provide some guidelines for how to set the DRT parameters. Furthermore, many of the components in the DRT for web services framework can be automated. To make it easier for potential adopters of DRT for web services, we have also developed a prototype application (described in Section 3.3).

3.2 Guidelines for Parameter Setting

Our previous work [14] found that the DRT performance depends on the number of partitions and the parameter ε. We next explore these impacts through a theoretical analysis, which, to be mathematically tractable, has the following assumptions:

1) The failure rate θ_i of each partition s_i $(i = 1, 2, \ldots, m,$ and $m > 1)$ is unknown, but can be estimated.
2) Each failure rate θ_i $(i = 1, 2, \ldots, m,$ and $m > 1)$ remains unchanged throughout the testing process (faults are not removed after their detection).
3) Test cases are selected with replacement, which means that some test cases may be selected more than once.

A principle of the DRT strategy is to increase the selection probabilities (by amount ε) of partitions with larger failure rates. In addition to the impact of the parameter ε, the number of partitions also influences the speed of updating the test profile (Formulas (1) to (4)). Therefore, for a given number of partitions, we are interested in investigating what values of ε yield the best DRT performance.

Letting θ_M denote the maximum failure rate, and s_M denote partitions with that failure rate, then p_i^n denotes the probability of executing the nth test case from partition s_i. As testing proceeds, the probability p_M of partition s_M being selected is expected to increase:

$$p_M^{n+1} > p_M^n. \tag{5}$$

In order to achieve the best performance, the probability of selecting the partition s_M (which has the maximum failure rate) is expected to increase. To achieve this, the increase in probability of s_M being selected for the next round should be larger than that for other partitions. We further analyze sufficient conditions for this goal, and can accordingly derive an interval for ε.

Initially, the test profile is $\{\langle s_1, p_1^0\rangle, \langle s_2, p_2^0\rangle, \ldots, \langle s_m, p_m^0\rangle\}$, which, after n test cases have been executed, is then updated to $\{\langle s_1, p_1^n\rangle, \langle s_2, p_2^n\rangle, \ldots, \langle s_m, p_m^n\rangle\}$. During the testing process, p_i^n is increased or decreased by the value ε, which is relatively small (set to 0.05 in previous studies [22], [23]). Because the initial p_i^0 is larger than ε, and the adjustment of p_i is relatively small (Formulas (1) to (4)), the following two situations are rare, and thus not considered here: $p_i < \varepsilon/(m-1)$ or $p_i < \varepsilon$ $(i = 1, 2, \ldots, m)$.

To explore the relationship between p_i^{n+1} and p_i^n, we calculate the conditional probability, $p(i|\delta)$, of the following four situations (denoted as $\delta_1, \delta_2, \delta_3$, and δ_4):

~~Situation 1 (δ_1):~~

1) *If $t_n \notin s_i$ and a fault is detected by t_n*, then $p(i|\delta_1)$ is calculated according to Formula (1):

$$p(i|\delta_1) = \sum_{i \neq j} \theta_j \left(p_i^n - \frac{\varepsilon}{m-1}\right).$$

2) *If $t_n \in s_i$ and a fault is detected by t_n*, then $p(i|\delta_2)$ is calculated according to Formula (2):

$$p(i|\delta_2) = \theta_i(p_i^n + \varepsilon).$$

3) *If $t_n \in s_i$ and no fault is detected by t_n*, then $p(i|\delta_3)$ is calculated according to Formula (3):

$$p(i|\delta_3) = (1 - \theta_i)(p_i^n - \varepsilon).$$

4) *If $t_n \notin s_i$ and no fault is detected by t_n*, then $p(i|\delta_4)$ is calculated according to Formula (4):

$$p(i|\delta_4) = \sum_{i \neq j} (1 - \theta_j) \left(p_i^n + \frac{\varepsilon}{m-1} \right).$$

Therefore, p_i^{n+1} for all cases together is:

$$p_i^{n+1} = \quad p_i^n + Y_i^n, \tag{6}$$

where

$$\begin{aligned} Y_i^n = & \frac{\varepsilon}{m-1} (2p_i^n \theta_i m - p_i^n m - 2p_i^n \theta_i + 1) \\ & - \frac{2\varepsilon}{m-1} \sum_{j \neq i} p_j^n \theta_j. \end{aligned} \tag{7}$$

From Formula (7), we have:

$$Y_M^n - Y_i^n = \quad \frac{2\varepsilon}{m-1} \left(m(p_M^n \theta_M - p_i^n \theta_i) - \frac{m(p_M^n - p_i^n)}{2} \right). \tag{8}$$

Before presenting the final guidelines, we need the following lemma.

Lemma 1. *If $p_i^n - p_M^n > 2(p_i^n \theta_i - p_M^n \theta_M)$, then $p_M^{n+1} > p_M^n$.*

Proof. See Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/TSC.2019.2960496. □

Accordingly, we can now present the following theorem that states a sufficient condition for achieving $p_M^{n+1} > p_M^n$.

Theorem 1. *For failure rate $\theta_{min} = min\{\theta_1, \ldots, \theta_m\}$, $\theta_M > \theta_{min}$, if $0 < \theta_{min} < \frac{1}{2}$, the following condition is sufficient to guarantee that $p_M^{n+1} > p_M^n$:*

$$\frac{2m\theta_{min}^2}{1 - 2\theta_{min}} < \varepsilon < \frac{(m-1)m\theta_{min}}{2(m+1)}. \tag{9}$$

Proof. See Appendix A, available in the online supplemental material. □

In summary, when $\frac{1}{2} < \theta_M < 1$, there is always an interval $\mathcal{I}$:

$$\varepsilon \in \left(\frac{2m\theta_{min}^2}{1 - 2\theta_{min}}, \frac{(m-1)m\theta_{min}}{2(m+1)}\right) \tag{10}$$

where $\theta_{min} \leq \theta_i, i \in \{1, 2, \ldots, m\}$, and $\theta_i \neq 0$, which can guarantee $p_M^{n+1} > p_M^n$.

From the proof in Appendix A, available in the online supplemental material, it is clear that the value of θ_M affects the upper bound ($\mathcal{I}_{upper}$) of $\mathcal{I}$. When $\theta_{min} < \theta_M < \frac{1}{2}$, the value of $\mathcal{I}_{upper}$ should be close to the lower bound of $\mathcal{I}$. In practice, we should set

$$\varepsilon \approx \frac{2m\theta_{min}^2}{1 - 2\theta_{min}}. \tag{11}$$

For a given partition scheme, with a total of m partitions, identification of the partition with the minimum failure rate (θ_{min}) first requires calculation of the failure rates of each partition, then identification of the minimum. Each partition's failure rate can be obtained in two ways:

1) It can be calculated directly as F/E (F is the number of failures and E is the number of executed tests), if the test history of the web service under test is available.
2) It can be approximated by $1/k_i$, where k_i is the total number of test cases executed before revealing a fault.

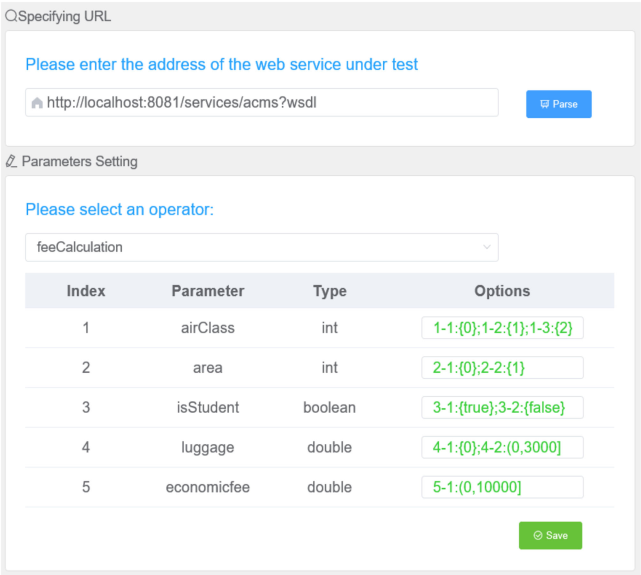
3.3 Prototype

This section describes a tool that partially automates DRT for web services, called DRTester[1]. DRTester supports the following tasks in testing web services: a) WSDL parsing; b) partition construction; c) setting DRT parameters (probability adjusting factor ε and test profiles); and d) test case preparation and execution. The details of DRTester are as follows:

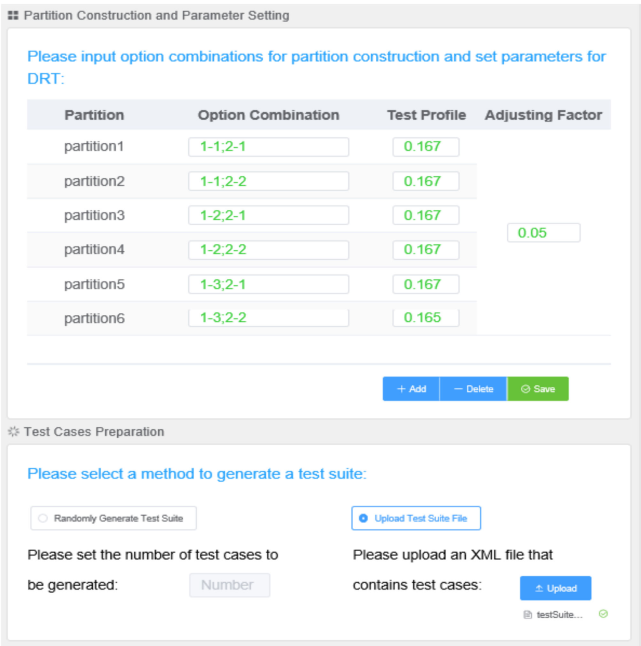
1) *Guidance*. This feature describes the steps the tester should follow when testing a web service.
2) *Configuration*. This feature, as shown in Fig. 2, interacts with the testers to obtain and set the information related to testing the web service, including: the address of the web service under test; the DRT parameters and partition details; and the test case preparation. The detailed steps are as follows:
   - *Inputting and parsing the URL (Fig. 2a)*: We integrate the WSDL parsing functionality provided by MT4WS [24]. This enables all the (WSDL) parameters and their types to be automatically obtained.
   - *Parameter setting (Fig. 2a)*: The tester is responsible for selecting which operations of the current web service under test are to be tested, and for partitioning each parameter into disjoint choices.
   - *Partition setting (Fig. 2b)*: The tester is responsible for specifying the partitions by combining the choices associated with each parameter.
   - *Test case generation (Fig. 2b)*: The tester is responsible for specifying the mode of test case generation (either randomly generating test cases based on the parameters, or uploading test cases generated using other techniques).
3) *Execution*. This feature presents a summary of the testing results, including details of the test case execution (input, expected output, partition, and result (pass or fail)). For randomly generated tests, the tester has to check each individual result. Otherwise, when all tests have been completed, a report is generated in a downloadable file.

The back-end logic is composed of several Restful APIs and Java classes: The APIs are responsible for communicating HTTP messages to and from the front-end interface. The controller class is responsible for updating the test profile according to the test results, and for selecting test cases from the partitions. The selected test cases are wrapped in SOAP messages and sent to the web service under test through the proxy class, which also intercepts the test results.

1. The prototype tool, together with a number of accompanying resources, has been made available at: https://github.com/phantomDai/DRTester.git



(a) WSDL Parsing and Parameters Setting



(b) Partition and DRT Parameter Setting

Fig. 2. DRTester configuration snapshots.

4 Empirical Study

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

4.1 Research Questions

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

RQ1 How effective is DRT at detecting web service faults?
Fault-detection effectiveness is a key criterion for evaluating the performance of a testing technique. This study used three popular real-life web services as subject programs, and applied mutation analysis to evaluate the effectiveness.

RQ2 How do the number of partitions and the DRT parameter ε impact on the failure detection effectiveness and efficiency of DRT?
In our earlier work [14], we found that the DRT parameter ε had a significant effect on DRT efficiency, and that the optimal value of the parameter could be related to the number of partitions. The relationship between ε and the number of partitions is examined through theoretical analysis, and verified through the empirical studies.

RQ3 Compared with the baseline techniques, how efficient is DRT at detecting web service faults in terms of time?
Compared with RT and PT, DRT incorporates the selection of partitions and test cases within a partition. Compared with AT, which also introduces feedback and adaptive control principles to software testing, DRT has a simple but efficient control strategy. Thus, we are interested in comparing the fault detection efficiency of DRT, RT, PT, and AT in terms of their time costs.

TABLE 1
Subject Web Services

Web service	LOC	Number of mutants
ACMS	116	3
CUBS	131	11
PBS	129	4

4.2 Subject Web Services

Although a number of web services are publicly available, for various reasons, their implementations are not. This renders them unsuitable for our experiments, which involve the creation of faulty mutants (requiring access to the implementations). We therefore selected three web services as the subject programs for our study, and implemented them ourselves, based on real-life specification:[2] `Aviation Consignment Management Service (ACMS)`; `China Unicom billing service (CUBS)`; and `Parking billing service (PBS)`. We used the tool MuJava [25] to conduct mutation analysis [26], [27], generating a total of 1563 mutants. Each mutant was created by applying a syntactic change (using one of all applicable mutation operators provided by MuJava) to the original program. Equivalent mutants, and those that were too easily detected (requiring less than 20 randomly generated test cases), were removed. To ensure the statistical reliability, we obtained 50 different test suites using different random seeds, then tested all mutants with all test suites, calculating the average number of test cases needed to kill (detect) a mutant. Table 1 summarizes the basic information of the used web services and their mutants. A detailed description of each web service is given in the following.

4.2.1 Aviation Consignment Management Service (ACMS)

`ACMS` helps airline companies check the allowance (weight) of free baggage, and the cost of additional baggage. Based

2. The implementations have been made available at: https://github.com/phantomDai/subjects4tsc.git

TABLE 2
ACMS Baggage Allowance and Pricing Rules

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

on the destination, flights are categorised as either domestic or international. For international flights, the baggage allowance is greater if the passenger is a student (30 kg), otherwise it is 20 kg. Each aircraft offers three cabin classes from which to choose (economy, business, and first), with passengers in different classes having different allowances. The detailed price rules are summarized in Table 2, where $price$ means economy class fare and $weight$ is the weight that exceeds the weight of the free carry.

4.2.2 China Unicom Billing Service (CUBS)

CUBS provides an interface through which customers can know how much they need to pay according to cell-phone plans, calls, and data usage. The details of several cell-phone plans are summarized in Tables 3, 4, and 5.

4.2.3 Parking Billing Service (PBS)

Consider a parking billing service that accepts the parking details for a vehicle, including the vehicle type, day of the week, discount coupon, and hours of parking. This service rounds up the parking duration to the next full hour, and then calculates the parking fee according to the hourly rates in Table 6. If a discount voucher is presented, a 50 percent discount off the parking fee is applied.

To facilitate better parking management, at the time of parking, customers may provide an estimation of parking duration, in terms of three different time ranges ((0.0,2.0], (2.0,4.0], and (4.0,24.0]). If the estimation and actual parked hours fall into the same time range, then the customer will receive a 40 percent discount; but if they are different ranges, then a 20 percent markup is applied. A customer may choose to either use a discount coupon, or provide an estimation of parking duration, but may not do both. No vehicles are allowed to remain parked for two consecutive days on a continuous basis.

4.3 Variables

4.3.1 Independent Variables

The independent variable is the testing technique. RT, RPT, DRT, and AT [12] were used for comparison.

4.3.2 Dependent Variables

The dependent variable for RQ1 is the metric for evaluating the fault-detection effectiveness. Several effectiveness metrics exist, including: the P-measure [28] (the probability of at least one fault being detected by a test suite); the E-measure [29] (the expected number of faults detected by a test suite); the F-measure [30] (the expected number of test case executions required to detect the first fault); and the T-measure [31] (the expected number of test cases required to detect all faults). Since the F- and T-measures have been widely used for evaluating the fault-detection efficiency and effectiveness of DRT-related testing techniques [6], [8], [22], [23], [31], [32], they are also adopted in this study. We use F and T to represent the F-measure and the T-measure of a testing method. As shown in Algorithm 1, the testing process may not terminate after the detection of the first fault. Furthermore, because the fault detection information can lead to different probability profile adjustment mechanisms, it is also important to see what would happen after revealing the first fault. Therefore, we introduce the F2-measure [30] as the number of additional test cases required to reveal the second fault after detection of the first fault. We use $F2$ to represent the F2-measure of a testing method, and $SD_{measure}$ to represent the standard deviation of metrics (where $measure$ can be F, $F2$, or T).

An obvious metric for RQ3 is the time required to detect faults. Corresponding to the T-measure, in this study we used T-$time$, the time required to detect all faults. F-$time$ and $F2$-$time$ denote the time required to detect the first fault, and the additional time needed to detect the second fault (after detecting the first), respectively. For each of these metrics, smaller values indicate a better performance.

TABLE 3
Plan A

Plan details		Month charge (CNY)		
		op_A^1	op_A^2	op_A^3
Basic	Free calls (min)	260	380	550
	Free data (MB)	40	60	80
	Free incoming calls	Domestic (including video calls)		
Extra	Incoming calls (CNY/min)	0.25	0.20	0.15
	Data (CNY/KB)	3E-4	3E-4	3E-4
	Video calls (CNY/min)	0.60	0.60	0.60

TABLE 4
Plan B

Plan details		Month charge (CNY)					
		op_B^1	op_B^2	op_B^3	op_B^4	op_B^5	op_B^6
Basic	Free calls (min)	120	200	450	680	920	1180
	Free data (MB)	40	60	80	100	120	150
	Free incoming calls	Domestic (including video calls)					
Extra	Incoming calls (CNY/min)	0.25	0.20	0.15	0.15	0.15	0.15
	Data (CNY/KB)	3E-4	3E-4	3E-4	3E-4	3E-4	3E-4
	Video calls (CNY/min)	0.60	0.60	0.60	0.60	0.60	0.60

TABLE 5
Plan C

Plan details		Monthly charge (CNY)										
		op_C^1	op_C^2	op_C^3	op_C^4	op_C^5	op_C^6	op_C^7	op_C^8	op_C^9	op_C^{10}	op_C^{11}
Basic	Free calls (min)	50	50	240	320	420	510	700	900	1250	1950	3000
	Free data (MB)	150	300	300	400	500	650	750	950	1300	2000	3000
	Free incoming calls	Domestic (including video calls)										
Extra	Incoming calls (CNY/min)	0.25	0.20	0.15	0.15	0.15	0.15	0.15	0.15	0.15	0.15	0.15
	Data (CNY/KB)	3E-4	3E-4	3E-4	3E-4	3E-4	3E-4	3E-4	3E-4	3E-4	3E-4	3E-4
	Video calls (CNY/min)	0.60	0.60	0.60	0.60	0.60	0.60	0.60	0.60	0.60	0.60	0.60

TABLE 6
Hourly Parking Rates

Actual parking hours	Weekday			Saturday and sunday		
	Motorcycle	Car: 2-door coupe	Car: others	Motorcycle	Car: 2-door coupe	Car: others
(0.0,2.0]	\$4.00	\$4.50	\$5.00	\$5.00	\$6.00	\$7.00
(2.0,4.0]	\$5.00	\$5.50	\$6.00	\$6.50	\$7.50	\$8.50
(4.0,24.0]	\$6.00	\$6.50	\$7.00	\$8.00	\$9.00	\$10.00

TABLE 7
Decision Table for ACMS

	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}	R_{12}	R_{13}	R_{14}	R_{15}	R_{16}	R_{17}	R_{18}	R_{19}	R_{20}	R_{21}	R_{22}	R_{23}	R_{24}
class	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2
destination	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1
isStudent	N	N	N	N	N	N	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	Y	Y	Y	Y	Y	Y
isOverload	N	N	N	N	N	N	N	N	N	N	N	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
$f_{1,1}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓												
$f_{1,2}$													✓						✓					
$f_{1,3}$														✓						✓				
$f_{1,4}$															✓						✓			
$f_{1,5}$																✓						✓		
$f_{1,6}$																	✓						✓	✓
$f_{1,7}$																		✓						

4.4 Experimental Settings

4.4.1 Partitioning

In our study, we set the partitions by making use of a decision table (DT) [33]. A DT presents a large amount of complex decisions in a simple, straightforward manner, representing a set of decision rules under all exclusive conditional scenarios in a pre-defined problem. Typically, a DT consists of four parts:

1) The upper-left part lists the conditions denoted C_i ($i = 1, \ldots, n$, where n is the number of conditions in the pre-defined problem, and $n \geq 1$). Each condition C_i contains a set of possible options $O_{i,q} \in CO_i = \{O_{i,1}, \ldots, O_{i,t_i}\}$, where t_i is the number of possible options for C_i, and $q = \{1, \ldots, t_i\}$.
2) The upper-right part shows the condition space, which is a Cartesian product of all the CO_i ($SP(C) = CO_1 \times CO_2 \times \ldots \times CO_n$). Each element in the $SP(C)$ is a condition entry (CE) with the ordered n-tuple.
3) The lower-left part shows all possible actions, represented A_j ($j = 1, \ldots, m$, where m is the number of possible actions and $m \geq 1$). Similar to CO_i, an action A_j contains a set of possible options $O'_{j,p} \in AO_j = \{O'_{j,1}, \ldots, O'_{j,k_j}\}$, where k_j is the number of alternatives for A_j, and $p = \{1, \ldots, k_j\}$.
4) The lower-right part shows the action space $SP(A)$, which is also a Cartesian product of all the AO_j ($SP(A) = AO_1 \times AO_2 \times \ldots \times AO_m$). Similar to CE, each element in the $SP(A)$ is an action entry (AE) with the ordered m-tuple.

A DT *rule* is composed of a *CE* and its corresponding *AE*. With DT, it is possible to obtain partition schemes with different granularities. For fine-grain partition schemes, each *CE* of a DT *rule* corresponds to a partition; while for coarse-grained schemes, a partition corresponds to the union of a group of partitions for which all *CE* of DT *rules* have the same *AE*. The decision tables for ACMS, CUBS, and PBS are shown in Tables 7, 8 and 9, respectively. In the tables, R_i ($i = 1, 2, \ldots, n$) denotes the identified ith *rule*; n is the total number of *rules*; and the checkmark (✓) under each *rule* indicates that the corresponding action should be taken. The details of actions are provided in Table 10, where w is the weight of baggage; *price* means economy class fare; *op* means the monthly charge; *call* and *data* mean the call duration and data usage, respectively; $freeCall$ and $freeData$

TABLE 8
Decision Table for CUBS

	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}	R_{12}	R_{13}	R_{14}	R_{15}	R_{16}	R_{17}	R_{18}	R_{19}	R_{20}
plan	A	A	A	B	B	B	B	B	B	C	C	C	C	C	C	C	C	C	C	C
option	op_A^1	op_A^2	op_A^3	op_B^1	op_B^2	op_B^3	op_B^4	op_B^5	op_B^6	op_C^1	op_C^2	op_C^3	op_C^4	op_C^5	op_C^6	op_C^7	op_C^8	op_C^9	op_C^{10}	op_C^{11}
$f_{2,1}$	✓	✓	✓																	
$f_{2,2}$				✓	✓	✓	✓	✓	✓											
$f_{2,3}$										✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

TABLE 9
Decision Table for PBS

	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	R_{11}	R_{12}	R_{13}	R_{14}	R_{15}	R_{16}	R_{17}	R_{18}
vehicle	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2
time	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1
discount	0	0	0	0	0	0	1	1	1	1	1	1	2	2	2	2	2	2
$f_{3,1}$	✓	✓	✓	✓	✓	✓												
$f_{3,2}$							✓	✓	✓	✓	✓	✓						
$f_{3,3}$													✓	✓	✓	✓	✓	✓

mean the free calls and free data, respectively; and *baseFee* means the cost before the discount. In Table 7, the condition for calculating the cost of the baggage includes *class* (0: first class; 1: business class; and 2: economy class), *isStudent* (Y: the passenger is a student; and N: the passenger is not a student), *isOverload* (Y: the baggage exceeds the free carry-on weight limit; and N: the baggage does not exceed the free carry-on weight limit.), and *Destination* (0: domestic flight; and 1: international flight). In Table 8, conditions that influence cell-phone bills include *plan* (A: plan A; B: plan B; and C: plan C) and *option* y under *plan* x, represented as op_x^y, where $x \in \{A, B, C\}$, and $y \in \{w | 1 \leq w \leq 11 \wedge w \in \mathcal{Z}\}$. In Table 9, conditions that affect the parking fee include the type of *vehicle* (0: motorcycle; 1: 2-door coupe; and 2: others), *day of week* (0: weekday; and 1: saturday or sunday), and *discount information* (0: customers provide a discount coupon; 1: the estimated hours of parking and the actual hours of parking fall into the same time range; and 2: estimated hours and the actual hours are in different time ranges).

TABLE 10
Formulas of the Actions in Table 7 ∼ 9

Web Service	Formulas
ACMS	$f_{1,1} = 0$ $f_{1,2} = (w - 25) \times price \times 1.5\%$ $f_{1,3} = (w - 35) \times price \times 1.5\%$ $f_{1,4} = (w - 25) \times price \times 1.5\%$ $f_{1,5} = (w - 47) \times price \times 1.5\%$ $f_{1,6} = (w - 37) \times price \times 1.5\%$ $f_{1,7} = (w - 27) \times price \times 1.5\%$
CUBS	$f_{2,1} = op + (call - freeCall) \times 0.25$ $+ (data - freeData) \times 0.0003$ $f_{2,2} = op + (call - freeCall) \times 0.20$ $+ (data - freeData) \times 0.0003$ $f_{2,3} = op + (call - freeCall) \times 0.15$
CUBS	$+ (data - freeData) \times 0.0003$ $f_{3,1} = baseFee \times 50\%$ $f_{3,2} = baseFee \times (1 - 40\%)$ $f_{3,3} = baseFee \times (1 + 20\%)$

As can be seen from the description above, because the DT considers all parameters, and identifies their invalid combinations, it can provide a systematic and efficient way to partition an input domain into disjoint subdomains, and then generate test cases. In practice, each DT *rule* condition entry corresponds to a partition in which test cases cover some paths—thus, the faults in those paths have a chance of being detected.

4.4.2 Initial Test Profile

Because test cases may be generated randomly during the test process, a feasible method is to use a uniform probability distribution as the initial testing profile. On the other hand, testers may also use past experience to help guide selection of a different probability distribution as the initial profile. In our experiment, we used a uniform probability distribution for the initial test profile. The initial test profiles of each web service are summarized in Table 11, where $< s_i, p_i >$ means that the probability of selecting partition s_i is p_i.

4.4.3 Constants

In the experiments, we were interested in exploring the relationship between the number of partitions and the DRT

TABLE 11
Initial Test Profile for Subject Web Services

Actual parking hours	Hourly parking rates	Initial test profile
ACMS	24 7	$\{\langle s_1, \frac{1}{24}\rangle, \langle s_2, \frac{1}{24}\rangle, \ldots, \langle s_{24}, \frac{1}{24}\rangle\}$ $\{\langle s_1, \frac{1}{7}\rangle, \langle s_2, \frac{1}{7}\rangle, \ldots, \langle s_7, \frac{1}{7}\rangle\}$
CUBS	20 3	$\{\langle s_1, \frac{1}{20}\rangle, \langle s_2, \frac{1}{20}\rangle, \ldots, \langle s_{20}, \frac{1}{20}\rangle\}$ $\{\langle s_1, \frac{1}{3}\rangle, \langle s_2, \frac{1}{3}\rangle, \ldots, \langle s_3, \frac{1}{3}\rangle\}$
PBS	18 3	$\{\langle s_1, \frac{1}{18}\rangle, \langle s_2, \frac{1}{18}\rangle, \ldots, \langle s_{18}, \frac{1}{18}\rangle\}$ $\{\langle s_1, \frac{1}{3}\rangle, \langle s_2, \frac{1}{3}\rangle, \ldots, \langle s_3, \frac{1}{3}\rangle\}$

strategy parameter ε, and therefore selected a set of parameter values: $\varepsilon \in \{1.0E\text{-}05, 5.0E\text{-}05, 1.0E\text{-}04, 5.0E\text{-}04, 1.0E\text{-}03, 5.0E\text{-}03, 1.0E\text{-}02, 5.0E\text{-}02, 1.0E\text{-}01, 2E\text{-}01, 3E\text{-}01, 4E\text{-}01, 5E\text{-}01\}$. It should be noted that $\varepsilon = 5E$-01 is already a large value. Consider the following scenario. For PBS, when the test is carried out under partition scheme 2, if $\varepsilon = 7.5E$-01 and a uniform probability distribution is used as the testing profile (that is, $p_i = 1/3$) , then suppose that the first test case belonging to c_1 is executed and does not reveal any faults, then, according to Formula (3), the value of p_1 would become 0. It is important, therefore, that the initial value of ε should not be set too large.

4.5 Experimental Environment

Our experiments were conducted on a virtual machine running the Ubuntu 11.06 64-bit operating system, with two CPUs, and a memory of 2GB. The test scripts were written in Java. To ensure statistically reliable values [34] of the metrics (F-measure, F2-measure, T-measure, F-time, F2-time, and T-time), each testing session was repeated 30 times with 30 different seeds, and the average value calculated.

4.6 Threats To Validity

4.6.1 *Internal Validity*

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

4.6.2 *External Validity*

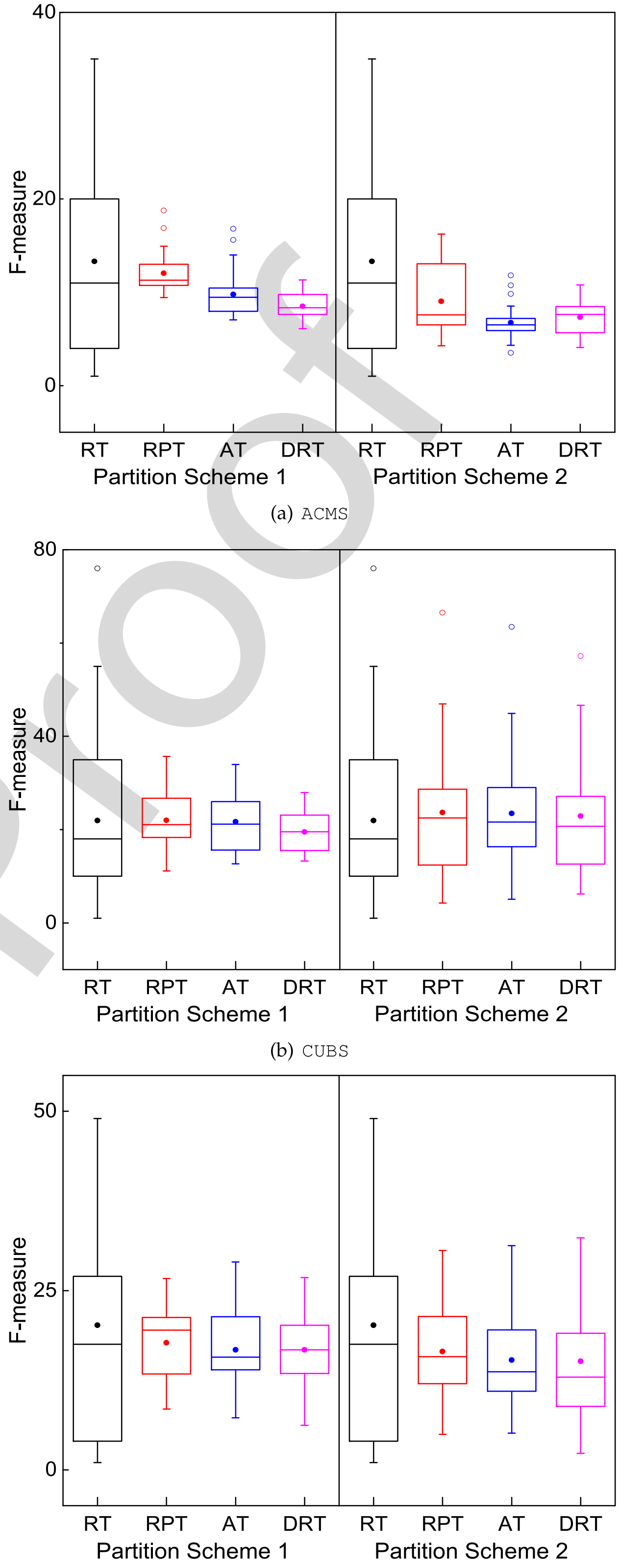
The possible threat to external validity is related to the subject programs and seeded faults under evaluation. Although the three subject web services are not very complex, they do implement real-life business scenarios of diverse application domains. Furthermore, 18 distinct faults were used to evaluate the performance. These faults cover different types of mutation operators and require an average of more than 20 randomly generated test cases to be detected. Although we have tried to improve the generalisability of the findings by applying different partitioning granularities, and 13 kinds of parameters, we anticipate that the evaluation results may vary slightly with different subject web services.

4.6.3 *Construct Validity*

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

4.6.4 *Conclusion Validity*

As reported for empirical studies in the field of software engineering [34], at least 30 observations are necessary to ensure the statistical significance of results. Accordingly, we have run a sufficient number of trials to ensure the reliability of our experimental results. Furthermore, as will be discussed in Section 5, we also conducted statistical tests to confirm the significance of the results.



(a) ACMS

(b) CUBS

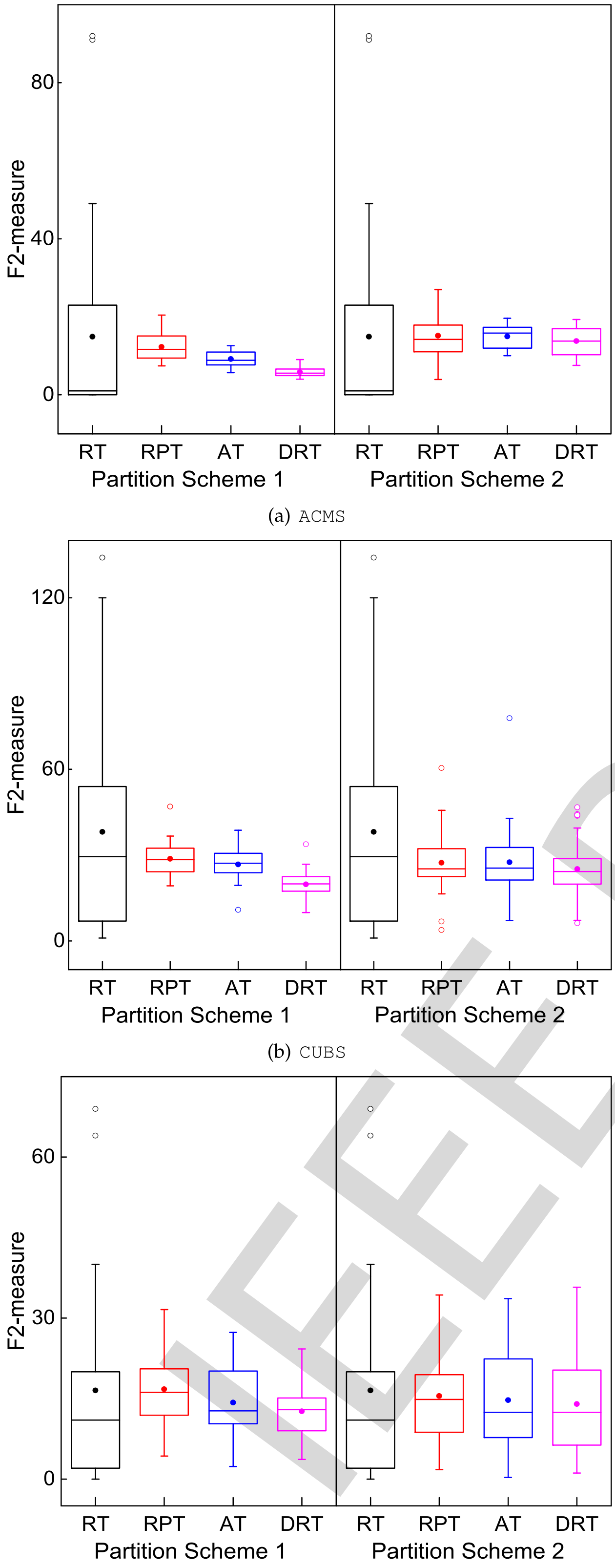
(c) PBS

Fig. 3. F-measure boxplots for each web service.

5 Experimental Results

5.1 RQ1: Fault Detection Effectiveness

F-, F2-, and T-measure results for ACMS, CUBS, and PBS are shown using boxplots in Figs. 3, 4, and 5, where the DRT

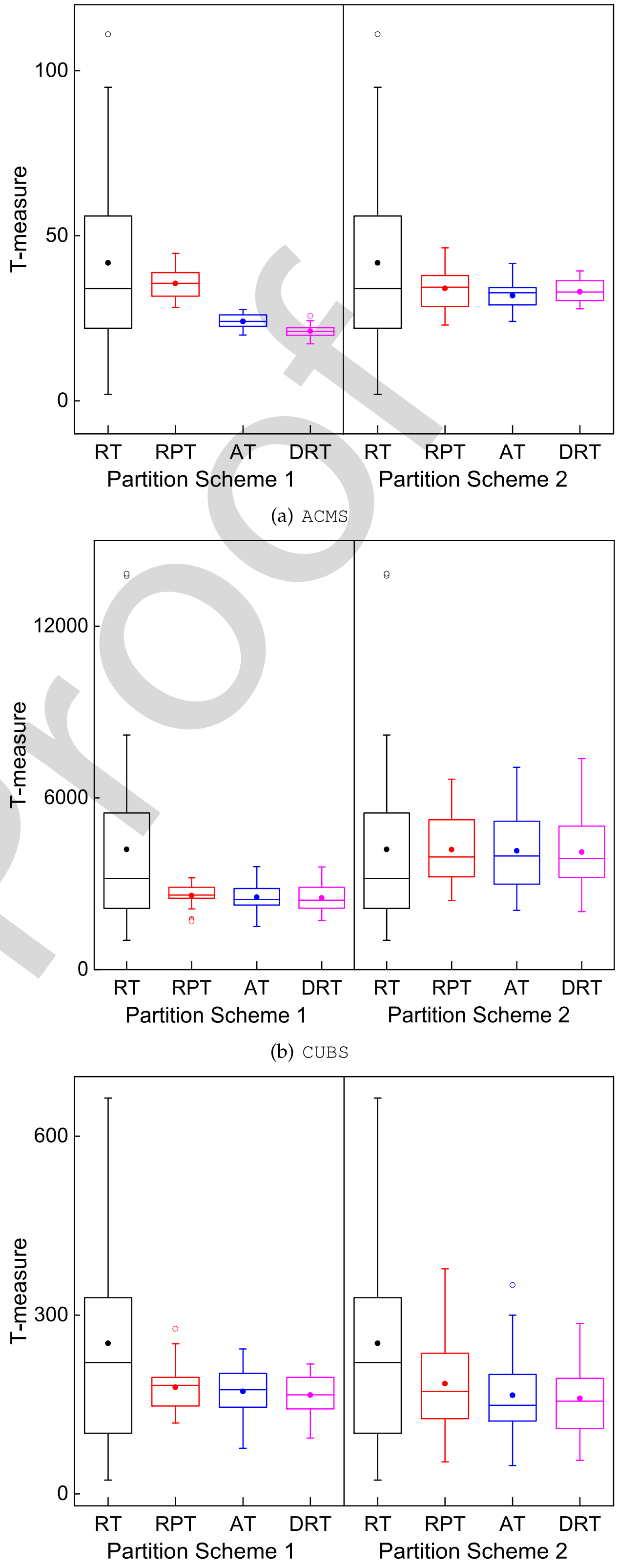


(a) ACMS

(b) CUBS

(c) PBS

Fig. 4. F2-measure boxplots for each web service.



(a) ACMS

(b) CUBS

(c) PBS

Fig. 5. T-measure boxplots for each web service.

parameter ε was set to the optimal values, as described in Section 5.2. The experimental results of DRT with other values of ε are shown in Appendix B, available in the online supplemental material. In each boxplot, the upper and lower bounds of the box represent the third and first quartiles of the metric, respectively; the middle line represents the median value; the upper and lower whiskers mark, respectively, the largest and smallest data within the range of $\pm 1.5 \times IQR$ (where IQR is the interquartile range); outliers beyond the IQR are denoted with hollow circles;

TABLE 12
Number of Scenarios Where the Technique on the Top Row has a Lower Metric (F-/F2-/T-Measure) Score Than the Technique on the Left Column

	F-measure				F2-measure				T-measure			
	RT	RPT	AT	DRT	RT	RPT	AT	DRT	RT	RPT	AT	DRT
RT	—	4	5	5	—	4	5	6	—	6	6	6
RPT	2	—	6	6	2	—	5	6	0	—	6	6
AT	1	0	—	4	1	1	—	6	0	0	—	5
DRT	1	0	2	—	0	0	0	—	0	0	1	—

TABLE 13
Theoretical Optimal Values of DRT Parameter

Web service	Partition scheme	θ_{min}	ε^*
`ACMS`	1	5.452E-2	1.601E-1
	2	2.797E-3	1.102E-4
`CUBS`	1	1.193E-3	5.702E-5
	2	1.397E-3	1.734E-5
`PBS`	1	1.760E-3	1.118E-4
	2	1.492E-3	1.340E-5

and each solid circle represents the mean value of the metric.

It can be observed from the figures that, in an overwhelming majority of cases, DRT was the best performer in terms of F-, F2-, and T-measure, followed by AT, RPT, and RT. On the other hand, RT may be the best performer occasionally or the worst performer in terms of F-, F2-, and T-measure, which means that the fault detection effectiveness of RT is not stable. In contrast, DRT and AT show a relatively stable fault detection effectiveness. We also conducted statistical testing to verify the significance of this observation, using the Holm-Bonferroni method [30] (with p-value equal to 0.05) to determine which pairs of testing techniques had significant differences. The statistical data are shown in Table 12, where each cell gives the number of scenarios where the technique above (in the table) performed better than one to the left. For example, the "6" in the top right cell of Table 12 indicates that, of 6 scenarios (two partition schemes $\times$ three web services), DRT had lower T-measure scores than RT for 6, with the fault-detection capabilities of these two techniques being significantly different.

Table 12 clearly shows that the differences between pairs of testing techniques are all significant.

5.2 RQ2: Relationship between Partition Number and ε

In Section 3.2, we analyzed the relationship between the number of partitions and the DRT parameter ε. In this section, we show that our theoretical analysis provides useful guidance to testers to set the value of ε.

We used three web services to validate our theoretical analysis. Before starting the test, it is necessary to know the failure rate θ_i of partition s_i. From Tables 2, 3, 4 and 5, it can be observed that the values of some parameters (such as the baggage weight, the call duration, and parking duration) are such that the total number of test case values in a partition could be infinite. For such a situation, we approximate the failure rate θ_i of s_i by $1/k_i$ (where k_i is the total number of test cases executed before revealing a fault). According to Formula (19), the theoretically optimal values of ε in each scenario for each web service are shown in Table 13, where ε^* denotes the theoretical value of ε. We ran a series of experiments with the parameters set according to those in Table 13: The F-, F2-, and T-measure results for each program are shown in Fig. 6, where ε_1^* and ε_2^* denote the theoretical values of parameter ε in the two different partition schemes, respectively. For ease of presentation and understanding, we used $log_{100}(1.0E05 \times \varepsilon)$ for the horizontal axis in Fig. 6. Apart from the DRT strategy parameter ε, all other experimental settings remained the same as in Section 5.1.

From Fig. 6, we have the following observations:

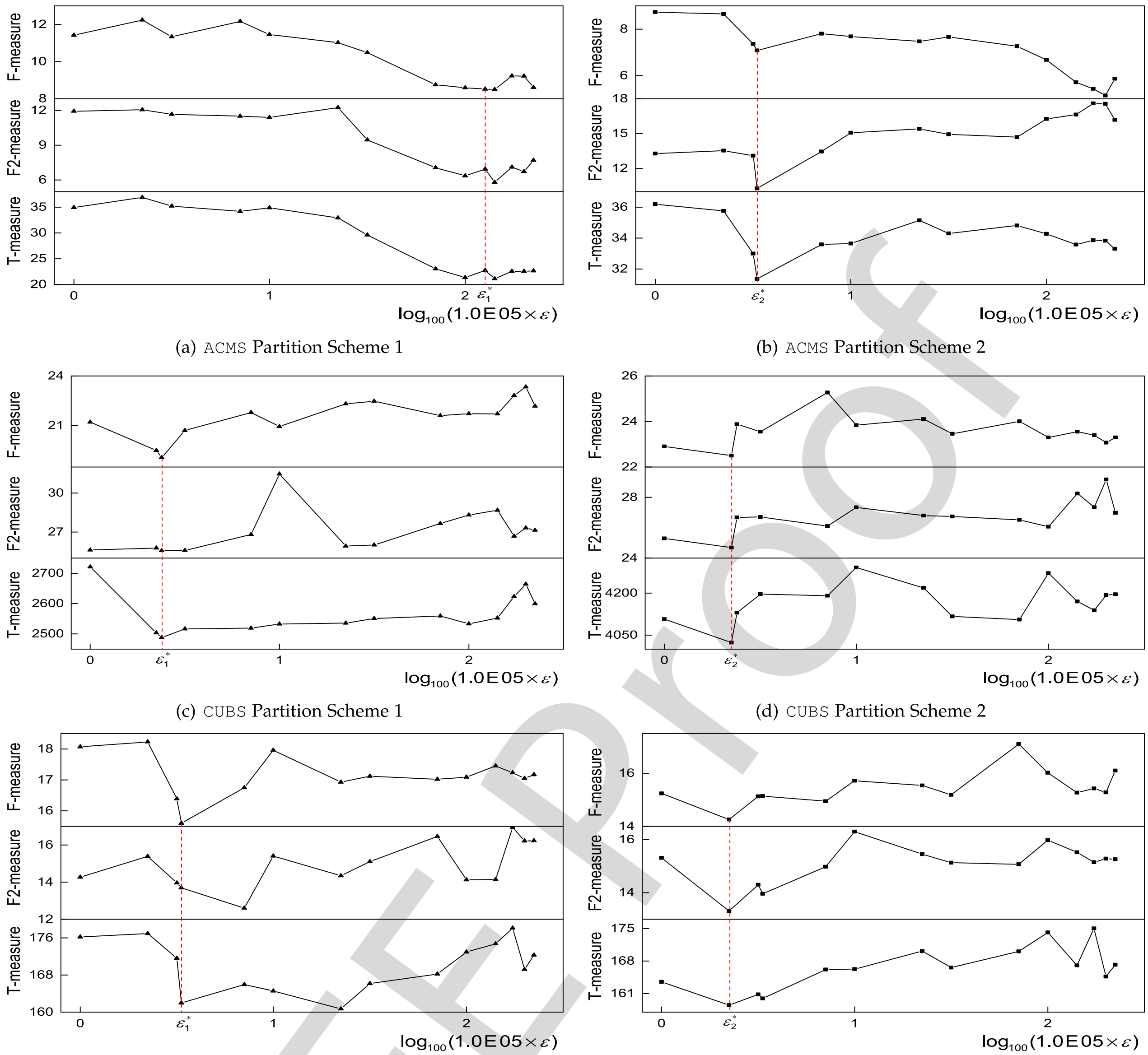
- In most scenarios, the DRT strategy with theoretically optimum parameter value performs best. Furthermore, the DRT strategy performs better when the parameter values are near the theoretically optimum value than when not.
- From Fig. 6a, it can be observed that the DRT strategy with larger parameter values performs better than with the theoretically optimum value, in terms of the F-measure. The main reason for this is that, for this scenario, the maximum failure rate ($\theta_M = 4.781E-3$) is large and the number of partitions is small: When the parameter value is large, the probability of selecting partitions with lower failure rates is quickly reduced, and the probability of selecting partitions with larger failure rates is quickly increased, according to Formulas (3) and (4).

5.3 RQ3: Fault Detection Efficiency

The F-, F2-, and T-time results for `ACMS`, `CUBS`, and `PBS` are summarized in Table 14, where the values of DRT parameters for the subject web services are the same as those in Section 5.1. The F-, F2-, and T-time results of DRT with different parameter values are summarized in Appendix B, available in the online supplemental material. It can be observed from the table that, in general, DRT had the best performance; RPT marginally outperforms RT; and AT had the worst performance.

As was done for the F-, F2-, and T-measure data, we used the Holm-Bonferroni method to check the difference between each pair of testing strategies in terms of F-time, F2-time, and T-time, as shown in Table 15. Table 15 shows that: a) DRT was significantly better than AT in terms of F-/F2-/T-time; b) DRT was significantly better than RT and RPT in terms of F2-/T-time; and c) DRT marginally outperformed RT and RPT in terms of F-time. In other words, the additional computation incurred in DRT by updating the test profile is compensated for in terms of test execution savings.

In summary, the DRT strategy is considered the best testing technique across this three metrics, RPT marginally outperformed RT, and DRT, RPT, and RT significantly outperformed AT.



(a) ACMS Partition Scheme 1

(b) ACMS Partition Scheme 2

(c) CUBS Partition Scheme 1

(d) CUBS Partition Scheme 2

(e) PBS Partition Scheme 1

(f) PBS Partition Scheme 2

Fig. 6. Line charts of F-measure, F2-measure, and T-measure values for each web service (for both the theoretically optimum parameter value, and other values.

5.4 Summary

Based on the evaluation, we have the following observations:

- DRT outperformed RT, RPT, and AT, according to all the applied metrics for all three studied web services. DRT marginally outperformed AT in terms of the F-, F2-, and T-measure, for all the studied web services. Moreover, AT incurs heavier computational overhead, and takes a significantly longer time. For instance, AT required 32429.07ms to select and execute sufficient test cases to detect all faults in CUBS, while DRT only needed 30.21ms (Table 14). This indicates that among RT, RPT, and AT, DRT should be chosen.
- DRT is more effective in terms of the F-, F2-, and T-measure when the parameter settings are optimal (according to the theoretical analysis): In most cases, DRT has the best performance for all three web services, according to these three metrics (F-measure, F2-measure, and T-measure) when following the guidelines for the parameter settings. This highlights the usefulness of the parameter-setting guidelines.

We also note the following limitations:

- While DRT outperformed RT and RPT in terms of fault detection effectiveness and efficiency, this was achieved at the cost of the additional effort required to set the partitions and test profiles.
- Applying DRT involves setting parameters, which may not be trivial. Even when following the theoretical guidelines.

6 Related Work

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

TABLE 14
F-time, F2-time, and T-time in ms for Subject Web Services

Partition Scheme	Metric	ACMS				CUBS				PBS			
		RT	RPT	AT	DRT	RT	RPT	AT	DRT	RT	RPT	AT	DRT
1	F-time	0.43	0.57	0.49	0.23	0.82	0.91	140.13	0.95	0.81	0.85	22.25	0.68
	F2-time	0.29	0.31	2.47	0.12	1.14	0.87	172.27	0.86	0.42	0.52	25.40	0.34
	T-time	0.85	1.08	2.47	0.43	34.69	30.54	32429.07	30.21	4.12	3.83	289.04	3.20
2	F-time	0.43	0.33	15.53	0.24	0.82	0.75	16.82	0.87	0.81	0.66	12.99	0.49
	F2-time	0.29	0.45	363.47	0.28	1.14	0.79	15.76	0.83	0.42	0.35	17.44	0.34
	T-time	0.85	0.78	459.67	0.65	34.69	34.59	2666.17	36.49	4.12	2.98	200.54	2.26

6.1 Testing Techniques for Web Services

In recent years, a lot of effort has been made to test web services [4], [13], [35], [36]. Test case generation or selection is core to testing web services, and model-based [37] and specification-based [38] techniques are two common approaches. Before making services available on the Internet, testers can use model-based techniques to verify whether or not the behavior of the WSUT meets their requirements. In these techniques, test data can be generated from a data model that specifies the inputs to the software—this data model can be built before, or in parallel to, the software development process. Verification methods using technologies such as theorem-proving [39], models [40] and Petri-Nets [41] also exist.

All of the above approaches aim to generate test cases without considering the impact of test case execution order on test efficiency. In contrast, Bertolino *et al.* [42] proposed using the category-partition method [43] with XML schemas to perform XML-based partition testing. Because PT aims to find subsets of all possible test cases to adequately test a system, it can help reduce the required number of test cases. Our approach involves software cybernetics and PT: In DRT, selection of a partition is done according to the testing profile, which is updated throughout the testing process. An advantage of DRT is that partitions with larger failure rates have higher probabilities of selection. Zhu and Zhang [44] proposed a collaborative testing framework, where test tasks are completed using collaborating test services—a test service is a service assigned to perform a specific testing task. Our framework (Section 3.1) aims to find more faults in the WSUT, with the result of the current test case execution providing feedback to the control system so that the next test case selected has a greater chance to reveal faults.

Most web service testing techniques assume that the computed output for any test case is verifiable, which is, however, not always true in practice (a situation known as the oracle problem [19]). Thus, many testing techniques may not be applicable in some cases. To address the oracle problem for testing web services, Sun *et al.* [21] proposed a metamorphic testing [45], [46] approach that not only alleviates the oracle problem, but is also a practical and efficient option for testing web services. They conducted a case study that showed that up to 94.1 percent of seeded faults could be detected without the need for oracles.

TABLE 15
Number of Scenarios Where the Technique on the Top Row has a Lower Metric (F-/F2-/T-Time) Score Than the Technique on the Left Column

	F-time				F2-time				T-time			
	RT	RPT	AT	DRT	RT	RPT	AT	DRT	RT	RPT	AT	DRT
RT	—	3	0	4	—	3	0	6	—	5	0	6
RPT	3	—	1	4	3	—	0	5	1	—	0	6
AT	6	5	—	6	6	6	—	6	6	6	—	6
DRT	2	2	0	—	0	1	0	—	0	0	0	—

6.2 Improving RT and PT

Based on the observation that failure-causing inputs tend to cluster into contiguous regions in the input domain [10], [11], much work has been done to improve RT [6], [7], [9]. Adaptive random testing [7], [9] is a family of techniques based on random testing that aim to improve the failure-detection effectiveness by evenly spreading test cases throughout the input domain. One well-known ART approach, FSCS-ART, selects a next test input from the fixed-size candidate set of tests that is farthest from all previously executed tests [47]. Many other ART algorithms have also been proposed, including RRT [48], DF-FSCS [49], and ARTsum [50], with their effectiveness examined and validated through simulations and experiments.

Adaptive testing (AT) [8], [51], [52] takes advantage of feedback information to control the execution process, and has been shown to outperform RT and RPT in terms of the T-measure and the number of detected faults, which means that AT has higher efficiency and effectiveness than RT and RPT. However, AT may require a rather long execution time in practice. To alleviate this, Cai *et al.* [6] proposed DRT, which uses testing information to dynamically adjust the testing profile. There are several things that can impact on DRT's test efficiency. Yang *et al.* [32] proposed A-DRT, which adjusts parameters during the testing process.

7 Conclusion

In this paper, to address the challenges of testing SOA-based applications, we have presented a dynamic random testing (DRT) method for web services. Our method uses random testing to generate test cases, and selects test cases from different partitions in accordance with a testing profile that is dynamically updated in response to the test data collected. In this way, the proposed method enjoys benefits from both random testing and partition testing.

We proposed a framework that examines key issues when applying DRT to test web services, and developed a prototype to make the method practical and effective. To guide testers to correctly set the DRT parameters, we used a theoretical analysis to study the relationships between the number of partitions (m) and the probability adjusting factor (ε). Three real web services were used as experimental subjects to validate the feasibility and effectiveness of our approach. Our experimental results show that, in general, DRT has better performance than both RT and RPT, in terms of the F-, F2-, and T-measures, and always outperforms when the ε settings follow our guidelines. In other words, our theoretical analysis can provide genuinely useful guidance to use DRT.

In our future work, we plan to conduct experiments on more web services to further validate the effectiveness, and identify the limitations of our method.

ACKNOWLEDGMENTS

This research was supported by the National Natural Science Foundation of China (Grant Nos. 61872039 and 61872167), the Beijing Natural Science Foundation (Grant No. 4162040), the Aeronautical Science Foundation of China (Grant No. 2016ZD74004), and the Fundamental Research Funds for the Central Universities (Grant No. ~~FRF-GF-17-B29~~). A preliminary version of this paper was presented at the 36th Annual IEEE Computer Software and Applications Conference (COMPSAC 2012) ~~[1]~~.

REFERENCES

[1] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-oriented computing: A research roadmap," *Int. J. Cooperative Inf. Syst.*, vol. 17, no. 2, pp. 223–255, 2008.
[2] C.-A. Sun, E. el Khoury, and M. Aiello, "Transaction management in service-oriented systems: Requirements and a proposal," *IEEE Trans. Services Comput.*, vol. 4, no. 2, pp. 167–180, Apr.-Jun. 2011.
[3] C. Bartolini, A. Bertolino, S. Elbaum, and E. Marchetti, "Whitening SOA testing," in *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Int. Symp. Foundations Softw. Eng.*, 2009, pp. 161–170.
[4] G. Canfora and M. D. Penta, "Service-oriented architectures testing: A survey," *Lecture Notes Comput. Sci.*, vol. 5413, no. 2, pp. 78–105, 2009.
[5] R. Hamlet, "Random testing," in *Encyclopedia of Software Engineering*. Hoboken, NJ, USA: Wiley, 2002.
[6] K.-Y. Cai, H. Hu, C. Jiang, and F. Ye, "Random testing with dynamically updated test profile," in *Proc. 20th Int. Symp. Softw. Rel. Eng.*, 2009, pp. 1–2.
[7] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. H. Tse, "Adaptive random testing: The art of test case diversity," *J. Syst. Softw.*, vol. 83, no. 1, pp. 60–66, 2010.
[8] 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, 2007.
[9] R. Huang, W. Sun, Y. Xu, H. Chen, D. Towey, and X. Xia, "A survey on adaptive random testing," *IEEE Trans. Softw. Eng.*, to be published, doi: 10.1109/TSE.2019.2942921.
[10] P. E. Ammann and J. C. Knight, "Data diversity: An approach to software fault tolerance," *IEEE Trans. Comput.*, vol. 37, no. 4, pp. 418–425, Apr. 1988.
[11] G. B. Finelli, "NASA software failure characterization experiments," *Reliability Eng. Syst. Safety*, vol. 32, no. 1, pp. 155–169, 1991.
[12] K.-Y. Cai, Y.-C. Li, and K. Liu, "Optimal and adaptive testing for software reliability assessment," *Inf. Softw. Technol.*, vol. 46, no. 15, pp. 989–1000, 2004.
[13] Y.-F. Li, P. K. Das, and D. L. Dowe, "Two decades of web application testing — A survey of recent advances," *Inf. Syst.*, vol. 43, no. 1, pp. 20–54, 2014.
[14] C.-A. Sun, G. Wang, K.-Y. Cai, and T. Y. Chen, "Towards dynamic random testing for web services," in *Proc. 36th IEEE Int. Comput. Softw. Appl. Conf.*, 2012, pp. 164–169.
[15] C.-A. Sun, M. Li, J. Jia, and J. Han, "Constraint-based model-driven testing of web services for behavior conformance," in *Proc. 16th Int. Conf. Service Oriented Comput.*, 2018, pp. 543–559.
[16] C.-A. Sun, Z. Wang, Q. Wen, P. Wu, and T. Y. Chen, "An empirical study on iterative metamorphic testing for web services," ~~*submitted for publication*~~.
[17] E. J. Weyuker and B. Jeng, "Analyzing partition testing strategies," *IEEE Trans. Softw. Eng.*, vol. 17, no. 7, pp. 703–711, Jul. 1991.
[18] T. Y. Chen and Y.-T. Yu, "On the relationship between partition and random testing," *IEEE Trans. Softw. Eng.*, vol. 20, no. 12, pp. 977–980, Dec. 1994.
[19] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Trans. Softw. Eng.*, vol. 41, no. 5, pp. 507–525, May 2015.
[20] S. Segura, D. Towey, Z. Q. Zhou, and T. Y. Chen, "Metamorphic testing: Testing the untestable," *IEEE Softw.*, to be published, doi: 10.1109/MS.2018.2875968.
[21] C.-A. Sun, G. Wang, B. Mu, H. Liu, Z. Wang, and T. Y. Chen, "Metamorphic testing for web services: Framework and a case study," in *Proc. 9th IEEE Int. Conf. Web Services*, 2011, pp. 283–290.
[22] J. Lv, H. Hu, and K.-Y. Cai, "A sufficient condition for parameters estimation in dynamic random testing," in *Proc. 35th IEEE Annu. Int. Comput. Softw. Appl. Conf. Workshops*, 2011, pp. 19–24.
[23] Y. Li, B. B. Yin, J. Lv, and K.-Y. Cai, "Approach for test profile optimization in dynamic random testing," in *Proc. 39th IEEE Annu. Int. Comput. Softw. Appl. Conf.*, 2015, pp. 466–471.
[24] C.-A. Sun, G. Wang, Q. Wen, D. Towey, and T. Y. Chen, "MT4WS: An automated metamorphic testing system for web services," *Int. J. High Perform. Comput. Netw.*, vol. 9, no. 1/2, pp. 104–115, 2016.
[25] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "MuJava: An automated class mutation system," *Softw. Testing Verification Rel.*, vol. 15, no. 2, pp. 97–133, 2005.
[26] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Comput.*, vol. 11, no. 4, pp. 34–41, 1978.
[27] C.-A. Sun, F. Xue, H. Liu, and X. Zhang, "A path-aware approach to mutant reduction in mutation testing," *Inf. Softw. Technol.*, vol. 81, pp. 65–81, 2017.
[28] J. W. Duran and S. C. Ntafos, "An evaluation of random testing," *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 4, pp. 438–444, Jul. 1984.
[29] T. Y. Chen and Y.-T. Yu, "Optimal improvement of the lower bound performance of partition testing strategies," *IEE Proc.-Softw. Eng.*, vol. 144, no. 5, pp. 271–278, 1997.
[30] C.-A. Sun, H. Dai, H. Liu, T. Y. Chen, and K.-Y. Cai, "Adaptive partition testing," *IEEE Trans. Comput.*, vol. 68, no. 2, pp. 157–169, Feb. 2019.
[31] L. Zhang, B.-B. Yin, J. Lv, K.-Y. Cai, S. S. Yau, and J. Yu, "A history-based dynamic random software testing," in *Proc. 38th Int. Comput. Softw. Appl. Conf. Workshops*, 2014, pp. 31–36.
[32] Z. Yang, B. Yin, J. Lv, K.-Y. Cai, S. S. Yau, and J. Yu, "Dynamic random testing with parameter adjustment," in *Proc. 38th IEEE Annu. Int. Comput. Softw. Appl. Conf. Workshops*, 2014, pp. 37–42.
[33] D. Gettys, "If you write documentation, then try a decision table," *IEEE Trans. Professional Commun.*, vol. PC-29, no. 4, pp. 61–64, Dec. 1986.
[34] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 1–10.
[35] M. Bozkurt, M. Harman, and Y. Hassoun, "Testing web services: A survey," Dept. Comput. Sci., King's College London, London, U. K., Tech. Rep. TR-10–01, 2010.
[36] D. Qiu, B. Li, S. Ji, and H. Leung, "Regression testing of web service: A systematic mapping study," *ACM Comput. Surveys*, vol. 47, no. 2, pp. 1–21, 2015.
[37] S. R. Dalal *et al.*, "Model-based testing in practice," in *Proc. 21st Int. Conf. Softw. Eng.*, 1999, pp. 285–294.
[38] Z. J. Li, J. Zhu, L.-J. Zhang, and N. M. Mitsumori, "Towards a practical and effective method for web services test case generation," in *Proc. 4th Int. Workshop Automat. Softw. Test*, 2009, pp. 106–114.
[39] A. Sinha and A. Paradkar, "Model-based functional conformance testing of web services operating on persistent data," in *Proc. Workshop Testing Anal. Verification Web Services Appl.*, 2006, pp. 17–22.
[40] A. M. Paradkar *et al.*, "Automated functional conformance test generation for semantic web services," in *Proc. 5th Int. Conf. Web Services*, 2007, pp. 110–117.
[41] D. Xiang, N. Xie, B. Ma, and K. Xu, "The executable invocation policy of web services composition with petri net," *Data Sci. J.*, vol. 14, 2015.
[42] A. Bertolino, J. Gao, E. Marchetti, and A. Polini, "Automatic test data generation for xml schema-based partition testing," in *Proc. 2nd Int. Workshop Automat. Softw. Test*, 2007, pp. 10–16.
[43] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating fuctional tests," *Commun. ACM*, vol. 31, no. 6, pp. 676–686, 1988.

[44] H. Zhu and Y. Zhang, "Collaborative testing of web services," *IEEE Trans. Services Comput.*, vol. 5, no. 1, pp. 116–130, Jan.-Mar. 2012.

[45] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: A new approach for generating next test cases," Dept. Comput. Sci., Hong Kong Univ. Sci. Technol., Hong Kong, Tech. Rep. HKUST-CS98–01, 1998.

[46] T. Y. Chen *et al.*, "Metamorphic testing: A review of challenges and opportunities," *ACM Comput. Surveys*, vol. 51, no. 1, pp. 1–27, 2018.

[47] T. Y. Chen, H. Leung, and I. Mak, "Adaptive random testing," in *Proc. 9th Annu. Asian Comput. Sci. Conf.*, 2004, pp. 320–329.

[48] K. P. Chan, T. Y. Chen, and D. Towey, "Restricted random testing: Adaptive random testing by exclusion," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 16, no. 4, pp. 553–584, 2006.

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

[50] 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 Trans. Comput.*, vol. 65, no. 12, pp. 3509–3523, Dec. 2016.

[51] 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 Int. Conf. Quality Softw.*, 2005, pp. 135–141.

[52] 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, 2009.

Chang-~~Ai~~ Sun received the bachelor degree in computer science from the University of Science and Technology Beijing, China, and the PhD degree in computer science from Beihang University, China. He is currently a professor with 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. His research interests include software testing, program analysis, and service-oriented computing. He is a senior member of the IEEE.

Hepeng Dai received the bachelor'~~s~~ degree in information and computing sciences from the China University of Mining and Technology, Beijing, and the master'~~s~~ degree in software engineering from the University of Science and Technology Beijing, China. He is currently working toward the PhD degree in the School of Computer and Communication Engineering, University of Science and Technology Beijing, China. His current research interests include software testing and debugging.

Guan Wang received the bachelor'~~s~~ degree in computer science from the University of Science and Technology Beijing. He is currently working toward the master'~~s~~ degree at the School of Computer and Communication Engineering, University of Science and Technology Beijing. His current research interests include software testing and service-oriented computing.

Dave Towey received the BA and MA degrees from the University of Dublin, Trinity College, Dublin, Ireland, PgCertTESOL degree from the Open University of Hong Kong, Hong Kong, the MEd degree from the University of Bristol, Bristol, United Kingdom, and the PhD degree from the University of Hong Kong, Hong Kong. He is currently an associate professor with the School of Computer Science, University of Nottingham Ningbo China. His current research interests include technology-enhanced teaching and learning, and software testing, especially metamorphic testing and adaptive random testing. He is a member of both the IEEE and the ACM.

Tsong Yueh Chen received the BSc and MPhil degrees from the University of Hong Kong, Hong Kong, the MSc and DIC degrees from the Imperial College of Science and Technology, London, United Kingdom, and the PhD degree in computer science from the University of Melbourne, ~~Parkville,~~ Australia. He is currently a professor of software engineering at the Department of Computer Science and Software Engineering, Swinburne University of Technology. He is the inventor of metamorphic testing and adaptive random testing. He is a member of the IEEE.

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