Postprint of article in *IEEE Transactions on Services Computing* (2014), doi: 10.1109/TSC.2014.2322621

Preemptive Regression Testing of Workflow-Based Web Services

Lijun Mei, W.K. Chan, *Member*, *IEEE*, T.H. Tse, *Senior Member*, *IEEE*,
Bo Jiang, *Member*, *IEEE*, and Ke Zhai

Abstract—An external web service may evolve without prior notification. In the course of the regression testing of a workflow-based web service, existing test case prioritization techniques may only verify the latest service composition using the not-yet-executed test cases, overlooking high-priority test cases that have already been applied to the service composition before the evolution. In this paper, we propose *Preemptive Regression Testing* (*PRT*), an adaptive testing approach to addressing this challenge. Whenever a change in the coverage of any service artifact is detected, PRT recursively preempts the current session of regression test and creates a sub-session of the current test session to assure such lately identified changes in coverage by adjusting the execution priority of the test cases in the test suite. Then, the sub-session will resume the execution from the suspended position. PRT terminates only when each test case in the test suite has been executed at least once without any preemption activated in between any test case executions. The experimental result confirms that testing workflow-based web service in the face of such changes is very challenging; and one of the PRT-enriched techniques shows its potential to overcome the challenge.

Index Terms—Evolving service composition, adaptive regression testing

1 Introduction

A workflow-based service [24] usually communicates with other web services [46] (referred to as *external services* [27], [43]) to implement all the required functionality. The service together with the external services constitutes a *service-based application*. Any change in the workflow-based service should be fully tested before its deployment, but testers are seldom able to enforce that every external service of a workflow-based service remains unchanged during a test session on the latter service. Hence, if the external services have evolved, the efforts spent on the workflow-based service working under the pre-evolved versions of the external services will not be realized as an assurance of the current service-based application. Testing should be re-conducted.

Regression testing [40] serves two purposes. First, it guards against regression faults [28]. Second, it verifies whether a web service working with external services behaves as expected even though it has not been modified since the "last" test session. To the best of our knowledge, the majority of existing regression testing research for web services only considers the scenarios for the first purpose. The study for the second purpose is still inadequately explored.

Fig. 1 shows an execution trace of a web service P that contains a service port $p1$, which invokes an external service S twice. In the figure, Scenarios 1 and 2 are the classic situations where the environmental context of P is *static*. A vast majority of existing regression testing research (such as [11], [16], [26], [29], [31], [35], [40]) focuses on these scenarios. Scenario 3, identified by the preliminary version [27] of this paper, is a *volatile* situation that has not been explored by existing test case prioritization techniques.

In Scenario 3, service S is bound to more than one version along the execution trace. Existing techniques would consider the regression test session to be completed without executing the whole test suite against the updated version $s2$ of S. Thus, although a test session targets to execute the entire test suite against the final service-based application (including external services), in reality, only some but not all test cases are applied.

In this paper, we propose a novel approach known as *Preemptive Regression Testing* (*PRT*) for the regression testing of workflow-based services. We refer to a change detectable in a regression test session for a service under test as a *late change*. If a late change occurs, PRT *preempts* the current test session and *creates* a new test sub-session to assure immediately the workflow-based service with

- L. Mei is with the Services Quality and Engineering Excellence, IBM Research – China, Tower A, Building 19, Zhongguancun Software Park, 8 Dongbeiwang West Road, Haidian District, Beijing 100193, P.R. China. E-mail: meilijun@cn.ibm.com.
- W.K. Chan is with the Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Hong Kong. E-mail: wkchan@cityu.edu.hk.
- T.H. Tse is with the Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong. E-mail: thtse@cs.hku.hk.
- B. Jiang is the corresponding author. He is with the School of Computer Science and Engineering, Beihang University, 37 Xuanyuan Road, Haidian District, Beijing, P.R. China. E-mail: jiangbo@buaa.edu.cn.
- K. Zhai is with The University of Hong Kong, Pokfulam, Hong Kong. E-mail: kzhai@cs.hku.hk.

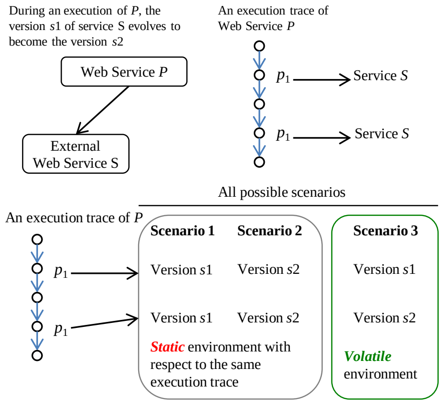
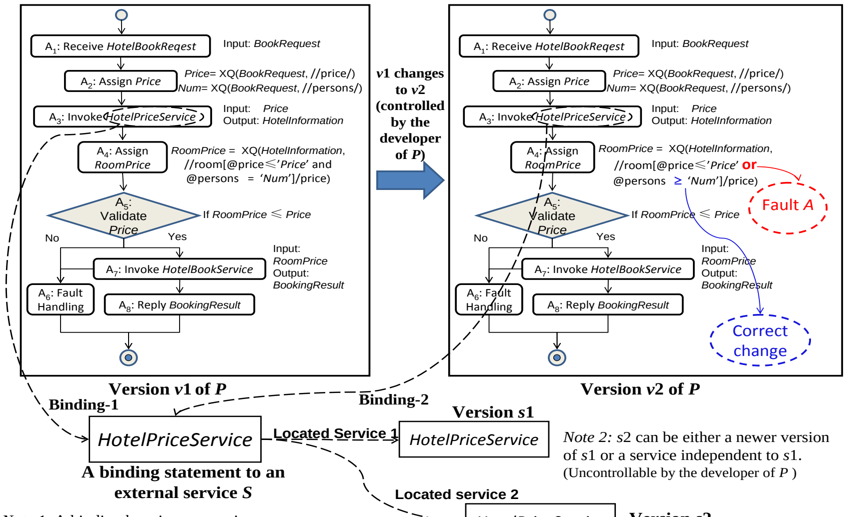


Fig. 1. Static and volatile testing environmental contexts. Traditional techniques are not aware of the changes in context as depicted in Scenario 3.



Note 2: s2 can be either a newer version of s1 or a service independent to s1. (Uncontrollable by the developer of P)

Note 1: A binding kept in a regression test case may be invalid when the test case is re-run.

Fig. 2. Example to illustrate a maintenance scenario for a workflow-based web service.

respect to the change. After completing the assurance of changes, the sub-session will *resume* execution from the suspended point in the updated sequence of test cases. Finally, PRT terminates if the entire test suite has been executed without any sub-session occurring between any two executions of test cases in the test suite.

We have conducted an experiment using all the benchmarks in the experiments reported in [24], [28], and have included a comparison with peer techniques [28]. The result reveals that formulating effective test case prioritization in Scenario 3 can be challenging: Existing techniques may completely miss to reveal regression faults. One series of our PRT-enriched techniques has the potential to overcome the challenge, and one of them outperforms all the others studied in the experiment.

A preliminary version of this paper was presented at the 36th Annual International Computer Software and Applications Conference (COMPSAC '12) [27]. It outlined the PRT strategies and systematically formulated a family of workflow-based PRT test case prioritization techniques. In the present paper, we detail the strategies and also systematically formulate a sister family of PRT test case prioritizations with additional experiments.

The main contribution of the present paper, together with its preliminary version [27], is threefold: (i) To the best of our knowledge, this is the first work that identifies the problems of service regression testing in Scenario 3. (ii) We propose the first work on preemptive regression testing to test service-based applications in the presence of evolving external services. (iii) We present the first empirical study on the efficiency and effectiveness of techniques for preemptive regression testing of services.

The remainder of this paper is organized as follows: Section 2 gives a motivating example. Section 3 presents our adaptive strategies and PRT-enriched techniques. Section 4 reports an empirical study, followed by discussions of the practicality issues in Section 5. Section 6 reviews related work. Finally, Section 7 concludes the paper.

2 Motivating Example

When an external service of a service-based application is deemed unsuitable, the developers or an automated agent may modify the binding address linked to this external service to a replacement external service. Such a revision may occur during a regression test session, and repairing a system configuration is a popular approach to addressing issues by testers. Testers may stop the current test session after the repair and restart a new test session. In other times, they may continue the current test session followed by deciding whether or not to start a new test session. As we will illustrate in Section 2.3, PRT is a refined strategy for the latter case.

2.1 Evolution Example

We motivate our work via an example taken from the *TripHandling* project [1]. We denote the trip handling service by *P*, and refer to the external hotel price enquiry service by *S*. We follow [24] to use an activity diagram to show a scenario in which the developer modifies version *v*1 of *P* to version *v*2. Version *v*1 originally binds to version *s*1 of *S*. The binding of version *s*1 of *S* to version *v*2 of *P* is updated during the test session. Our target for regression testing is to assure the correctness of *v*2.

In each activity diagram, a node and an edge represent a workflow process and a transition between two activities, respectively. We annotate the nodes with extracted program information, such as the input-output parameters of the activities and XPath [38]. We number the nodes as A_i (i = 1, 2, ..., 8).

(a) A_1 receives a hotel booking request from a user and stores it in the variable *BookRequest*.

(b) A_2 extracts the input room price and the number of persons via two XPaths //price/ and //persons/ from *BookRequest*, and stores these values in the variables *Price* and *Num*, respectively.

(c) A_3 invokes the service *HotelPriceService* to select available hotel rooms with prices not exceeding the input *Price* (that is, within budget), and keeps the reply in *HotelInformation*.

(d) A_4 assigns a price via the XPath //room [@price≤'Price' and @persons='Num']/price/ to *RoomPrice*.

(e) A_5 verifies whether the price in *HotelInformation* does not exceed the inputted *Price*.

(f) If the verification at A_5 passes, A_7 executes *HotelBookService* to book a room followed by A_8 returning the result to the customer.

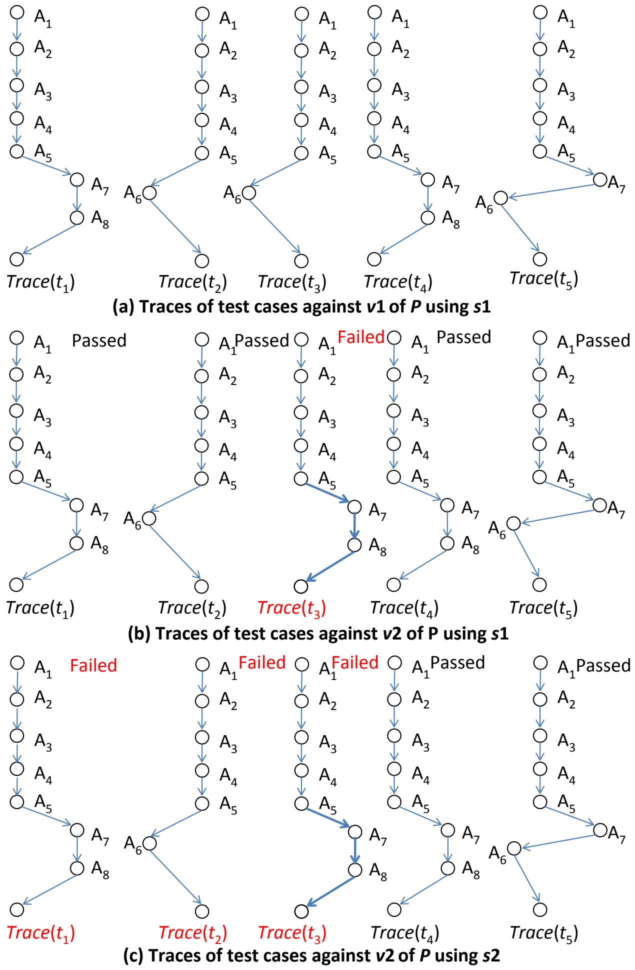
(g) If *RoomPrice* is erroneous or *HotelBookService* in node A_7 produces a failure, A_6 will invoke a fault handler.

Test cases t_1 to t_5 below each contains the price (*Price*) and the number of guests (*Num*) as parametric inputs:

	$\langle Price, Num\rangle$		$\langle Price, Num\rangle$
Test case t_1:	$\langle 200, 1\rangle$	Test case t_4:	$\langle 110, 1\rangle$
Test case t_2:	$\langle 100, 5\rangle$	Test case t_5:	$\langle -1, 1\rangle$
Test case t_3:	$\langle 125, 3\rangle$		

Suppose that only two types of rooms are available, namely, single rooms at a price of \$105 and family rooms (for three persons) at a price of \$150.

Suppose a software engineer Jim decides to make the following changes to the precondition in node A_4 of version $v1$ of P in Fig. 2: He attempts to allow customers to select any room that can accommodate the requested number of persons. However, he wrongly changes the precondition in the XPath by changing "and" to "or". Although he intends to provide customers with more choices, the change does not support his intention (because the process is designed to immediately proceed to book rooms, rather than allowing customers to specify their preferences). This results in a fault (which we will call Fault A), as shown in version $v2$ of P in Fig. 2.



(a) Traces of test cases against *v1* of *P* using *s1*

(b) Traces of test cases against *v2* of P using *s1*

(c) Traces of test cases against *v2* of *P* using *s2*

Fig. 3. Traces of the test cases against different service versions.

Suppose further that $s1$ of S is an exact search, returning all hotel rooms whose prices are smaller than *Price* (in ascending order of the room price). On the other hand, suppose $s2$ of S is a fuzzy search, which returns only one hotel room whose price is closest to *Price*, hoping that the customer will consider it as long as it is affordable. For example, when *Price* is 200, $s2$ will only return a family room. When *Price* is 100, $s2$ will return a single room, rather than returning no room. Replacing $s1$ by $s2$ in Fig. 2 will result in another fault (called Fault B).

Fig. 3a shows the execution traces of the five test cases against version $v1$ of P that uses version $s1$ of S as the hotel price enquiry service. Both test cases t_1 and t_4 result in the successful booking of a single room. Test cases t_2 and t_5 result in unsuccessful bookings. The price validation process rejects t_2 and t_3. The price "−1" of t_5 will trigger a fault in node A_7. Fig. 3b shows the traces of version $v2$ of P using version $s1$ of S. Similarly, Fig. 3c shows the traces of $v2$ of P using $s2$ of S. In particular, only the execution traces of t_3 are different among Fig. 3a, Fig. 3b, and Fig. 3c. The test case t_3 aims to book a family room; however, owing to the modification, a single room is booked. This test case can detect a regression fault. Both t_1 and t_2 are failed test cases for $s2$ because of the fuzzy search, whereas they are both passed test cases in Fig. 3b. Test case t_1 should book a single room, but it results in booking a triple room against $v2$ of P using $s2$. The execution of t_2 should report no available room. However, executing t_2 against $v2$ of P using $s2$ will report a validation failure of room price.

2.2 Inadequacies of Existing Techniques

This section analyzes the inadequacies of existing test case prioritization techniques. For brevity, let us concentrate our discussions on *addtl-workflow-branch*, which is a traditional strategy adopted from the *addtl-statement* test case prioritization technique [28]. Based on the coverage shown in Fig. 3a, we present two test case permutations $T_1 = \langle t_1, t_3, t_5, t_4, t_2\rangle$ and $T_2 = \langle t_1, t_2, t_5, t_3, t_4\rangle$ of *addtl-workflow-branch*. Let us consider three evolution scenarios.

Scenario 1: $v1$ evolves to $v2$ before executing T_1 against $s1$ of S. The application of T_1 to assure the correctness of $v2$ is shown in Fig. 4a. The second test case (t_3) of T_1 detects a failure, thus revealing Fault A.

Scenario 2: $v1$ evolves to $v2$ before executing T_1, and $s1$ evolves to $s2$ after executing t_1. The second test case (t_3) in T_1 reveals Fault A. In theory, the first (t_1) and fifth (t_2) test cases can both reveal Fault B, but only t_2 is executed (as the last test case).

Scenario 3: $v1$ evolves to $v2$ before executing T_2, and $s1$ evolves to $s2$ after executing t_5, as shown in Fig. 4b. The fourth test case (t_3) in T_2 reveals Fault A. In theory, the first (t_1) and second (t_2) test cases in T_2 can also reveal Fault B, but neither of them is executed.

Scenario 1 is the same as the classical setting for regression testing, and thus existing techniques work well. For Scenario 2, although the branch coverage missed by t_3 (namely, $A5 \rightarrow A6$ and $A6 \rightarrow$End) can be achieved by t_2, existing prioritization techniques does not advance its execution. Better test case prioritization techniques should be used to fill the gap.

Scenario 3 illustrates that existing techniques may fail to detect some faults (Fault B in this case). Intuitively, t_1, t_2, and t_5 can be considered as being discarded when executing T_2 against $v2$ and $s2$. This defies the objective of test case prioritization, in which test cases are to be reordered but not discarded. To fix this problem, more test cases need to be scheduled after the current test suite has been applied.

2.3 Illustration of a PRT-Enriched Technique

We now illustrate one technique that uses Strategy 1 (see Section 3.3 for details) to address the above challenge.

We observe from Fig. 4b that, although t_3 is targeted for covering A_6, it actually covers A_7 and A_8. The PRT Strategy 1 then selects test cases from the prioritized test suite to assure the correctness of A_6 immediately. Fig. 4b illustrates that Strategy 1 selects t_2 as a replacement according to the given priority shown by the test suite T, and happens to reveal a failure. Then, it continues with the execution of every remaining prioritized test case after t_2. After executing the remaining test cases in the prioritized test suite, the technique finds that t_1 has been executed before the latest invocation of Strategy 1. Hence, the technique reruns t_1 and reveals another failure. There is no need to suspend test case execution throughout the realization of Strategy 1.

3 Preemptive Regression Testing Strategies

We present the ***Preemptive Regression Testing*** approach and formulate three corresponding strategies.

3.1 Test Case Prioritization Revisited

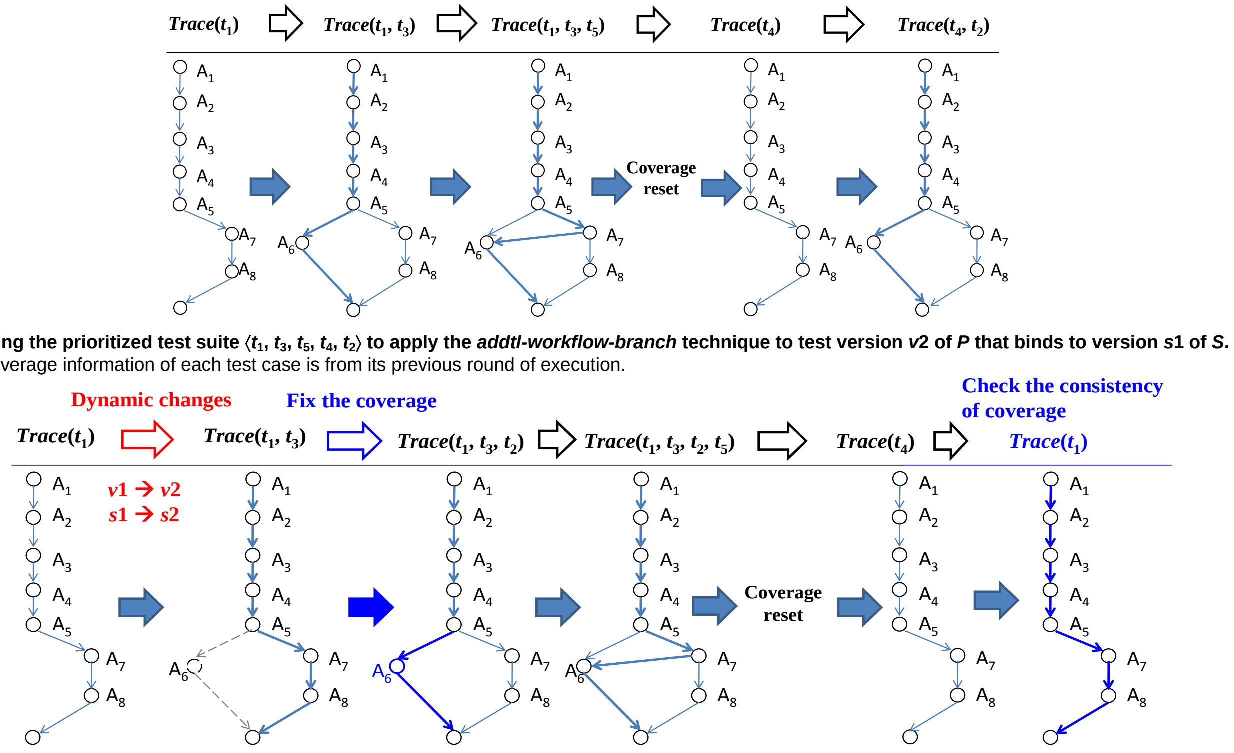
We would like to design techniques to make use of data obtained in previous software executions and to run test cases to achieve target goals in the regression testing of the next modified versions. Test case prioritization [27], [40] is an important aspect of regression testing. A well-designed test case prioritization technique may increase the fault detection rate of a test suite.

The test case prioritization problem [27] is: *Given*: T, a test suite; PT, the set of permutations of T; and f, a function from PT to real numbers. *Objective*: To find $T' \in PT$ such that $\forall T'' \in PT, f(T') \geq f(T'')$.

3.2 Regression Testing Model

Consider a service-based application divided into two parts. The first part is the workflow-based service under test, denoted by P. Following existing research [16], [28], our primary objective is to safeguard P from faulty modifications of its implementations. Typically, the testers of P use a regression test suite T to test a given version v of P. They may conduct testing in laboratory so that they can collect the coverage information on v.

The second part is a set of services outside the service P. P needs to communicate with them in order to compute its functions properly. We call them external services of P. In other words, in our setting, executing a test case against a given version v of P may involve the invocation of external services and obtaining their results. It would be too restrictive to assume that these external



(a) Using the prioritized test suite ⟨t_1, t_3, t_5, t_4, t_2⟩ to apply the *addtl-workflow-branch* technique to test version *v2* of *P* that binds to version *s1* of *S*. The coverage information of each test case is from its previous round of execution.

(b) Applying *Strategy* 1 when *v1* → *v2* and *s1* → *s2* during a regression test session using the same prioritized test suite ⟨t_1, t_3, t_5, t_4, t_2⟩. The coverage information of each test case is from its latest execution.

Fig. 4. Example to illustrate regression testing of workflow-based service with external service evolution.

services remain unchanged during any regression test session. We do not assume that the testers of *P* can always control the evolution of all external services either.

To facilitate discussions, we first present a generalized regression testing model.

Definition 1 (Generalized regression testing model). *A generalized RT model for a service P under test is a five-tuple* $\langle V, T, \Delta, \varphi, \theta \rangle$ *such that:*

- $V = (v_1, v_2, \ldots, v_n)$ *is a series of sequentially modified versions of P.*
- *T is a regression test suite of P.*
- $\Delta(v_i, v_{i+1})$ *is the time period between two consecutive versions* $v_i, v_{i+1} \in V$.
- $\varphi(t, v)$ *is the time taken to execute* $t \in T$ *against* $v \in V$.
- $\theta(t, v)$ *is the set of coverage after executing* $t \in T$ *against* $v \in V$.

When we consider only two consecutively modified versions v_i and v_{i+1} of *P,* we may or may not be able to completely execute all applicable test tests against v_i, which results in either of the following inequalities that constrains the number of test cases applied to v_i:

$$\Delta(v_i, v_{i+1}) \geq \sum_{k=0}^{|T|} \varphi(t_k, v_i) \tag{1}$$

$$\Delta(v_i, v_{i+1}) < \sum_{k=0}^{|T|} \varphi(t_k, v_i) \tag{2}$$

Many existing test case prioritization techniques [40] implicitly assume that constraint (1) is satisfied. They cannot accurately model the values of Δ and φ. However, a service composition may change rapidly, turning the regression testing scenario to fall within constraint (2).

Continuous delivery (also known as *DevOps*) is a real-life example satisfying constraint (2) and is very useful for the system integration phase. *DevOps* always requires the shipping of trunk code (branching code only for release). Code changes are minor and directly submitted to the trunk, which needs to be ready for incremental deployment that happens very frequently (in terms of hours rather than months). To achieve the goal of always shipping high-quality trunk code, continuous integration testing is thus necessary. During such testing, frequent minor changes may occur because different developers may continuously contribute to the debugging process, or because of small feature updates. Moreover, *DevOps* uses the real runtime environment or something highly similar (including application servers, system configurations, related data sources, external web services), which is also changing, and the changes may not always be controllable. Considering the continuous evolution in the code and the runtime environment, a regression test suite may not necessarily be fully executed against a stable version.

To transfer test data among test sessions, we formalize in (3)–(6) below the relations between the existing coverage (based on the regression test session against version v_j) and the new coverage (for the current test session against version v_{j+1}). $\emptyset$ denotes the empty set. The notation $\overline{\theta(t, v)}$ stands for the complementary set of $\theta(t, v)$. Thus, $\overline{\theta(t, v)}$ comprises all the elements in the total coverage set outside of $\theta(t, v)$.

$$\theta(t_i, v_j) \cap \overline{\theta(t_i, v_{j+1})} = \emptyset \textbf{ and } \overline{\theta(t_i, v_j)} \cap \theta(t_i, v_{j+1}) = \emptyset \tag{3}$$

$$\theta(t_i, v_j) \cap \overline{\theta(t_i, v_{j+1})} = \emptyset \textbf{ and } \overline{\theta(t_i, v_j)} \cap \theta(t_i, v_{j+1}) \neq \emptyset \tag{4}$$

$$\theta(t_i, v_j) \cap \overline{\theta(t_i, v_{j+1})} \neq \emptyset \textbf{ and } \overline{\theta(t_i, v_j)} \cap \theta(t_i, v_{j+1}) = \emptyset \tag{5}$$

$$\theta(t_i, v_j) \cap \overline{\theta(t_i, v_{j+1})} \neq \emptyset \textbf{ and } \overline{\theta(t_i, v_j)} \cap \theta(t_i, v_{j+1}) \neq \emptyset \tag{6}$$

In short, there are four types of relations for a test case: Relations (3), (4), and (5) mean, respectively, that the coverage achieved by the new execution of the test case t_i are the same as, more than, and less than that achieved in the previous session. Relation (6) means that the existing coverage and the new coverage of the test case t_i satisfy none of the above (such as when no existing coverage of the test case t_i is available). We note that (3) indicates there is no coverage change. The remaining three equations indicate that some change has occurred, and hence regression testing should be conducted, which lead to our PRT strategies to be presented in the next section.

3.3 PRT Strategies

This section presents three adaptive strategies, namely, *fix*, *reschedule*, and a hybrid approach *fix-and-reschedule*.

Strategy 1 (*Fix*). Suppose a test case *t* misses at least one coverage item that it has covered in its last execution. Let *F* be the set of missed coverage items of *t*. This strategy selects a sequence *U* of test cases in *T* such that the last execution of all the test cases in *U* can minimally cover all the missed coverage items in *F*. Moreover, this strategy records the coverage items achieved by each newly selected test case in *U* with respect to *F*.

Because the coverage achieved by many not-yet-executed test cases in *T* in their corresponding last executions may cover some item in *F*, Strategy 1 adopts the following criterion to construct *U* and run these test cases. For every missed coverage item in *F*, Strategy 1 chooses and executes one test case among the not-yet-executed test cases in *T* in a round-robin fashion (starting from the position of *t* in *T*) in descending order of the number of items covered by each test case.

Executing such a replacement test case may discover additional coverage items that have been missed as well. In this case, Strategy 1 will preempt its current session, and invoke a new session. The new session will adjust the prioritized test cases, resume execution from the preemption point, and then remove from *F* of the current session those coverage items already covered by the recursively invoked sessions of Strategy 1.

Strategy 2 (*Reschedule*). If a test case covers new item(s) that have not been covered in its last execution, the strategy records the additional coverage items achieved by the test case, and reprioritizes the not-yet-executed test cases according to the additional item coverage technique.

Strategy 3 (*Fix-and-Reschedule*). This strategy is a hybrid of Strategies 1 and 2. If a test case does not cover some item(s) it has covered in its last execution, Strategy 3 first invokes Strategy 1. After the completion of Strategy 1, if there are any additional coverage items that have not been covered in the last execution of the test cases by Strategy 1, it will invoke Strategy 2.

Algorithm 1. RegressionRun (T, ρ, θ)

```
1.   $T_1 \leftarrow \rho(T)$;                   // $T_1$ is the prioritized test suite
2.   $i \leftarrow 0$;                        // $i$ is the index of the test case
3.   while ($i$ < sizeof($T_1$)) {
4.       $t \leftarrow T_1(i)$;
5.       $\theta(t)$ = execute($t$);  // update the execution trace of $t$
6.       $i \leftarrow i + 1$;
7.   }
```

Algorithm 2. PRTRegressionRun (T, ρ, θ, S)

```
1.   $T_1 \leftarrow \rho(T)$;                   // $T_1$ is the prioritized test suite
2.   $i \leftarrow 0$;                        // $i$ is the index of the test case
3.   while ($i$ < sizeof($T_1$)) {
4.       $t \leftarrow T_1(i)$;
5.       $\theta'(t)$ = execute($t$);  // update the execution trace of $t$
6.       if($\theta(t)$ != $\theta'(t)$){     // start a test sub-session
7.           PRTSubRegressionRun($T_1, \rho, \theta, \theta', S, i$, 1);
8.           break;
9.       }
10.      $i \leftarrow i + 1$;
11.  }
PRTSubRegressionRun ($T, \rho, \theta, \theta', S, start, count$)
12.  $i \leftarrow$ ($start$ + $count$) % sizeOf($T$);  // round-robin search for $i$
     // apply PRT strategy $S$ to adjust the prioritized suite
13.  $T_2 \leftarrow S(T, \rho, \theta, \theta', i)$;  // adjust the prioritized suite
14.  while ($count$ < sizeOf($T_2$))  {
15.      $t \leftarrow T_2(i)$;
16.      $\theta'(t)$ = execute($t$);  // obtain the execution trace of $t$
17.      if($\theta(t)$ != $\theta'(t)$){     // start a test sub-session
18.          PRTSubRegressionRun($T_2, \rho, \theta, \theta', S, i$, 1);
19.          break;
20.      }
21.      $i \leftarrow (i + 1)$ % sizeOf($T_2$);  // round-robin search for $i$
22.      $count \leftarrow count + 1$;
23.  }
```

TABLE 1
COMPARISONS BETWEEN EXISTING AND PRT TECHNIQUES

Question	Existing Techniques	PRT Techniques
Is evolution during regression testing considered?	No	Yes
What is the number of test cases executed in one execution of a technique?	Equal to |T| (test suite size)	Maybe larger than |T|
Is a test case executed more than once during one session?	No	Yes
Is the latest coverage data used during one session?	No	Yes
No. of test sessions per execution of a technique	Single test session	Hierarchical test session

The PRT strategies require additional storage so that they can mark the end of each session of iteration. In our implementation, we use an integer array (of the same length as the size of the test suite), which is sufficient to support the marking, and hence the incurred extra space requirement is light.

Algorithm 1 (RegressionRun) and Algorithm 2 (PRTRegressionRun) show the existing strategy and the PRT strategy, respectively.

The algorithm RegressionRun accepts three input parameters: A test suite T, a baseline prioritization technique ρ, and the set of execution traces θ obtained from a given regression test session. We denote the ith test case in T by $T(i)$, and the execution trace of a test case t by $\theta(t)$. The function sizeof(T) returns the number of test cases in T. The function execute(t) executes the test case t and returns the execution trace of t against the service under test. Algorithm 1 first prioritizes the given test suite T using ρ, and then iterates on the prioritized test suite T_1 to execute each test case t using the function execute(t).

The algorithm PRTRegressionRun is significantly different from the algorithm RegressionRun. After the execution of a test case t at line 5, Algorithm 2 compares the trace $\theta(t)$ and the trace $\theta'(t)$. If there is any difference between the two, it starts a new test sub-session at line 7 using PRTSubRegressionRun. In PRTSubRegressionRun, the algorithm first identifies the start position at line 12 and then applies PRT strategy S (see Strategies 1–3 above) to construct a test suite. It applies all the test cases in this latter test suite one by one in the current sub-session. If there is any difference in the traces observed (line 17), it recursively starts a new test sub-session.

Let us use Fig. 4b as an example to illustrate Algorithm 2. We use *addtl-workflow-branch* as ρ, Strategy 1 as S, and T_1 is $\langle t_1, t_3, t_5, t_4, t_2\rangle$. After executing t_1 and t_3, the execution trace of t_3 changes, items that fail to be covered (namely, $A_5 \rightarrow A_6$ and $A_6 \rightarrow end$) are identified, and PRTSubRegressionRun is invoked. By using Strategy 1, the algorithm finds that t_2 can fix the coverage, and thus adjusts T_1 to T_2 $\langle t_1, t_3, t_2, t_5, t_4\rangle$. Then, PRTSubRegressionRun continues to execute t_2, t_5, t_4, *and* t_1 until *count* reaches 5.

In summary, a PRT-enriched technique may generate one or more regression test sub-sessions (lines 7 and 18). It considers the latest coverage data, and conducts fast adjustment in line 13 using a PRT strategy. Moreover, compared with existing techniques, PRT-enriched techniques may schedule additional test cases after the last test case in a given prioritized test suite has been executed, thus increasing the probability of fault detection. Table 1 summarizes the effects of the two types of strategies.

3.4 PRT-Enriched Test Case Prioritization Techniques

This section presents a family of test case prioritization techniques, as summarized in Table 2. M1 and M2 are existing techniques on workflow, and M6 and M7 are existing techniques on workflow and XRG. They are adapted from existing work (such as [16], [26], [28], [40]).

Addtl-branch technique is the most effective in terms of APFD in the literature. Therefore, we realize our three strategies on top of the *addtl-workflow-branch* technique and the *addtl-workflow-XRG-branch* technique to build six evolution-aware techniques. M3–M5 and M8–M10, listed in italics in Table 2, are the new techniques based on the application of our adaptive strategies.

These techniques have a common stopping criterion: Given a service P and a regression test suite T for P, the technique stops if the entire test suite T has been executed and no test case results in further changes in the coverage of P (in terms of workflow for M1–M5 and both workflow and XRG for M6–M10).

M1 (Total-Workflow-Branch) [28], [40]. This technique sorts the test cases in T in descending order of the total number of workflow branches executed by each test case. If multiple test cases cover the same number of workflow branches, M1 orders them randomly.

TABLE 2
CATEGORIES OF TEST CASE PRIORITIZATION TECHNIQUES

Category	Name	Index
Workflow-based	Total-Workflow-Branch [28], [40]	M1
	Addtl-Workflow-Branch [28], [40]	M2
	Addtl-Workflow-Branch-Fix	M3
	Addtl-Workflow-Branch-Reschedule	M4
	Addtl-Workflow-Branch-FixReschedule	M5
XRG-based	Total-Workflow-XRG-Branch [28], [40]	M6
	Addtl-Workflow-XRG-Branch [28], [40]	M7
	Addtl-Workflow-XRG-Branch-Fix	M8
	Addtl-Workflow-XRG-Branch-Reschedule	M9
	Addtl-Workflow-XRG-Branch-FixReschedule	M10

M2 (Addtl-Workflow-Branch) [28], [40]**.** This technique iteratively selects a test case t that yields the greatest cumulative workflow branch coverage, and then removes the covered workflow branches from all remaining test cases to indicate that these branches have been covered by the selected test cases. Additional iterations will be conducted until all workflow branches have been covered by at least one selected test case. If multiple test cases cover the same number of workflow branches in the current session of selection, M2 selects one of them randomly. If no remaining test cases can further improve the cumulative workflow branch coverage, M2 resets the workflow branch covered by each remaining test case to its original value. It applies the above procedure until all the test cases in T have been selected.

M3 (Addtl-Workflow-Branch-Fix). This technique consists of two phases. *Phase 1: preparation*. It first updates the workflow branches covered by individual test cases to be the same as M2 to generate a sequence of test cases. *Phase 2: runtime adjustment*. Right after the execution of a test case, it runs Strategy 1 to adjust the sequence of prioritized test cases, and then continues to apply the adjusted sequence of prioritized test cases in a round-robin fashion until the entire test suite has been executed and no test case changes its achieved coverage between the current execution and the last execution.

M4 (Addtl-Workflow-Branch-Reschedule). This technique consists of two phases: *Phase 1: preparation*. This phase is the same as Phase 1 of M3. *Phase 2: runtime adjustment*. It is the same as Phase 2 of M3, except that it runs Strategy 2 rather than Strategy 1.

M5 (Addtl-Workflow-Branch-FixReschedule). This technique strikes a balance between M3 and M4 by using Strategy 3. It also consists of two phases. *Phase 1: preparation*. This phase is the same as Phase 1 of M3. *Phase 2: runtime adjustment*. It is the same as Phase 2 of M3, except that it runs Strategy 3 instead of Strategy 1.

M6 (Total-Workflow-XRG-Branch) [28], [40]. This technique is the same as M1, except that it uses the total number of workflow branches and XRG branches, instead of the total number of workflow branches covered by each test case.

M7 (Addtl-Workflow-XRG-Branch) [28], [40]. This technique is the same as M2, except that it uses the workflow branches and XRG branches, rather than the workflow branches covered by each test case.

M8 (Addtl-Workflow-XRG-Branch-Fix). This technique is the same as M3, except that it uses the workflow branches and XRG branches, instead of the workflow branches covered by each test case.

M9 (Addtl-Workflow-XRG-Branch-Reschedule). This technique is the same as M4, except that it uses the workflow branches and XRG branches, instead of the workflow branches covered by each test case.

M10 (Addtl-Workflow-XRG-Branch-FixReschedule). This technique is the same as M5, except that it uses the workflow branches and XRG branches, instead of the workflow branches covered by each test case.

A PRT-enriched technique has two types of costs, namely, preparation cost and adjustment cost. Take M3 as an example. Its preparation cost is the same as M2 (an existing technique), but its adjustment cost depends on the number of coverage changes of test cases. The time complexity of the adjustment cost for one change is $O(n)$, where n is the test suite size, while existing techniques require rescheduling the whole test suite after detecting changes in coverage, so that the time complexity is $O(n^2)$.

With reference to existing regression testing studies [11], [16], [22], [23], [24], [28], [29], [31], [32], [34], we also include ***random ordering*** (referred to as *Random* in this paper, and as *Rand* in Figs. 6, 7, and 8) for comparison in the experiment to be presented in the next section.

4 EVALUATION

4.1 Experimental Setup

We chose a set of eight subject service-based applications to evaluate our adaptive strategies, as listed in Table 3. Six benchmarks (*atm, gymlocker, loanapproval, marketplace, purchase,* and *triphandling*) were downloaded from the IBM BPEL repository [1] and the BPWS4J repository [8].

TABLE 3
BENCHMARKS AND THEIR DESCRIPTIVE STATISTICS

Benchmark Ref.	Benchmark Description	Modified Versions	Elements	LOC	XPaths	XRG Branches	WSDL Elements	Used Versions
A	*atm*	8	94	180	3	12	12	5
B	*buybook*	7	153	532	3	16	14	5
C	*dslservice*	8	50	123	3	16	20	5
D	*gymlocker*	7	23	52	2	8	8	5
E	*loanapproval*	8	41	102	2	8	12	7
F	*marketplace*	6	31	68	2	10	10	4
G	*purchase*	7	41	125	2	8	10	4
H	*triphandling*	9	94	170	6	36	20	8
Total		60	527	1352	23	114	106	43

TABLE 4
STATISTICS OF TEST SUITE SIZES

Benchmark Size	A	B	C	D	E	F	G	H	Mean
Maximum	146	93	128	151	197	189	113	108	140.6
Average	95	43	56	80	155	103	82	80	86.8
Minimum	29	12	16	19	50	30	19	27	25.3

The benchmark *buybook* [18] was downloaded from Oracle Technology Network for Oracle BPEL Process Manager. The benchmark *dslservice* was downloaded from the Web Services Innovation Framework [24].

They were representative service-based applications developed in WS-BPEL [36]. This set of benchmarks was also used in previous empirical studies reported in [24], [28]. To the best of our knowledge, this set of benchmarks is larger than the set of benchmarks used by Ni et al. in their experiment reported in [30] in terms of number of benchmarks, variety of benchmarks, number of versions, and sizes of individual benchmarks.

We used the set of faults and the associated test suites in the benchmark packages to measure the effectiveness of different prioritization techniques. We follow the spirit of mutation testing [2] to seed faults in the major artifacts (BPEL, XPath, and WSDL [37]) of the benchmark applications. Andrews et al. [2] suggested that mutation faults can be representative of real faults. Many researchers thus used mutation testing for empirical evaluation of test case prioritization techniques [14]. We used three typical types of mutations in fault seeding: value mutations, decision mutations, and statement mutations. Since BPEL can be treated as Control Flow Graphs (CFG), the above mutations can be seeded in the way as seeding faults in CFG. An XPath fault is the wrong usage of XPath expressions, such as extracting the wrong content, or failing to extract any content. Fig. 2 gives an example of an XPath fault. A WSDL fault is the wrong usage of WSDL specifications, such as binding to a wrong WSDL specification, or inconsistent message definitions. The faults in the modified versions have been reported by [24]. The statistics of the selected modified versions are shown in the rightmost column of Table 3.

Strictly following the methodology in [24], [28], the fault in any modified version could be detected by some test case in every test suite, and we discarded any modified version if more than 20 percent of the test cases could detect the failures in that version. All the 43 remaining versions were used in the empirical study.

We used the implementation tool of Mei et al. [28] for test case generation, test suite construction, and fault seeding in our empirical study. We revisit the procedure here: First, it randomly generated test cases based on the WSDL specifications, XPath queries, and workflow logics of the original version of each benchmark (rather than the modified versions). For each benchmark, 1,000 test cases were generated to form a test pool. The tool then added a test case to a constructing test suite (initially empty) only if the test case can increase the coverage achieved by executing the test suite against the workflow branches, XRG branches, or WSDL elements. We successfully obtained 100 test suites for each benchmark. The descriptive statistics of the test suites are shown in Table 4. It presents the maximum, average, and minimum numbers of test suites for each benchmark.

To simulate different evolution scenarios, we set the evolution points (that is, the times when the evolutions happen) to be the instances that 40, 60, and 80 percent of a test suite has been executed, respectively. To simulate scenarios in a dynamic service environment, we randomly selected them from the original version and the modified versions. Since all the test case execution results of the applications can be determined, we can figure out whether a fault has been revealed by a test case through comparing the test result of the modified version with that of the original program. Our tool automatically performed the comparisons.

4.2 Measurement Metric

We measured the effectiveness using the *Average Percentage of Faults Detected* (*APFD*) [40]. It is a widely adopted metric to evaluate test case prioritization techniques. A higher *APFD* value indicates faster fault detection. Let *T* be a test suite of *n* test cases, *F* be a set of *m* faults revealed by *T*, and TF_i be the index of the first test case in a reordering *T'* of *T* that reveals fault *i*. The *APFD* value for the reordering *T'* is computed by

$$APFD = 1 - \frac{TF_1 + TF_2 + \ldots + TF_m}{n\,m} + \frac{1}{2\,n}$$

4.3 Data Analysis

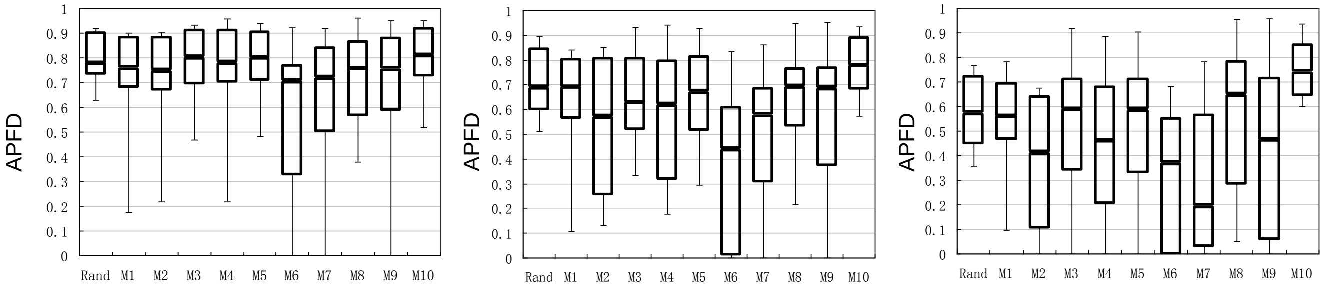
4.3.1 Effectiveness

We first study the overall effectiveness of each technique in terms of APFD. The corresponding results for the evolution points 40, 60, and 80 percent of the test suites are represented using boxplots in Fig. 5a, Fig. 5b, and Fig. 5c, respectively. Each boxplot graphically shows the 25th percentile, median, and 75th percentile of the APFD result achieved by a technique.

We find across the three subfigures that, for each technique, as the evolution point progresses (namely, from 40 to 60 percent and from 60 to 80 percent), the performance of the technique shows a downward trend. For example, the mean APFD values of *Random* in Fig. 5a, Fig. 5b, and Fig. 5c are 0.79, 0.70, and 0.57, respectively, and the mean APFD value of the adaptive technique M10 only slightly decreases from 0.79 to 0.77, and then to 0.75. The bars for other techniques across the three plots can be interpreted similarly.

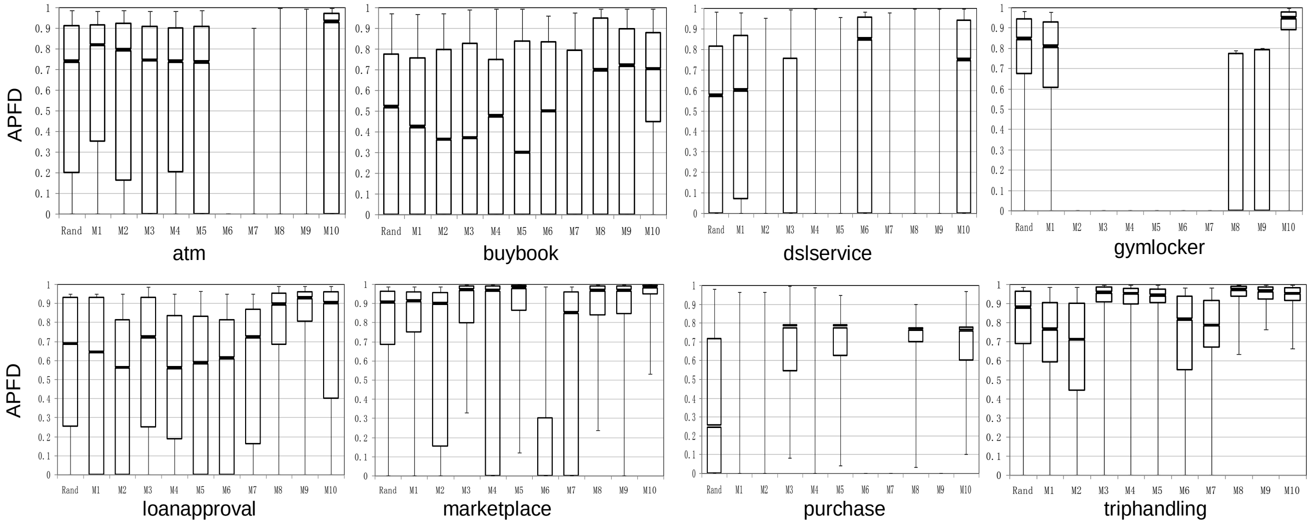
The differences among XRG-based techniques (M6–M10) widen as the evolution point progresses. A similar observation can also be made among the workflow-based techniques (M1–M5). We also observe that M10 is the most stable and the most effective among the ten techniques. The results indicate that integrating the *fix* strategy and the *reschedule* strategy together can be more effective for earlier revelation of failures.

Fig. 6 shows the APFD result of each technique for each benchmark when the evolution point is 80 percent of the test suite. Take the plot for *atm* as an example. We observe that there are missing bars for M6, M7, M8, and M9. Similarly, we can observe missing bars for M2-M7 for *gymlocker*. In our experiment, services may evolve within a regression test session. Hence, a test case that can detect a failure in a particular service composition may not be able to do so in a next service composition. Moreover, all fault-revealing test cases may have been executed before the evolution. Failing to add such test cases again to supplement testing after the evolution will miss to detect the failure caused by the evolution. Hence, if a test case



(a) Evolution point: 40% test suite size (b) Evolution point: 60% test suite size (c) Evolution point: 80% test suite size

Fig. 5. Overall comparisons of mean APFD measures.



atm buybook dslservice gymlocker

loanapproval marketplace purchase triphandling

Fig. 6. APFD comparisons of test case prioritization techniques for each benchmark.

scheduling strategy is inadequate, it may miss to detect the failures.

Random does not report missing bars in our experiment. We have examined the reason and found that fault-revealing test cases of the same fault appeared in indexed positions of the prioritized test cases both before and after evolutions. Thus, some faults can still be revealed after the evolution. Since *Random* includes no guidelines for arranging test cases before or after evolution, we tend to believe that the arrangement of test cases in the experiment by *Random* is coincidental. Moreover, *Random* does not always outperform the PRT-enriched techniques (in terms of APFD), and we will further discuss it below.

Another interesting phenomenon is that the workflow-based series without PRT enrichment perform better than the XRG-based series without PRT enrichment (e.g., M2 outperforms M7 by 10 percent in Fig. 5c), whereas the XRG-based series with PRT enrichment outperform the workflow-based series with PRT enrichment (e.g., M10 outperforms M2 by 40 percent in Fig. 5c). On closer look, we find that in the case of no evolution in the previous experiment [28], the XRG-based series is more effective in revealing faults earlier. However, in case evolution occurs, the number of remaining fault-revealing test cases in the suite prioritized by the XRG-based series is significantly less than that remaining in the suite prioritized by the workflow-based series. Hence, the workflow-based series outperforms the XRG-based series. Our strategy uses a round-robin approach. Thus, fault-revealing test cases that are executed before the evolution can still have the chance to be re-executed. The XRG-based series with PRT enrichment may, therefore, outperform the workflow-based series.

TABLE 5
SUMMARY OF WHETHER A TECHNIQUE HAS AT LEAST 50 PERCENT OF THE COLLECTED APFD VALUES TO BE ZERO

Benchmark	Random			M1			M2			M3			M4			M5			M6			M7			M8			M9			M10		
	0.4	0.6	0.8	0.4	0.6	0.8	0.4	0.6	0.8	0.4	0.6	0.8	0.4	0.6	0.8	0.4	0.6	0.8	0.4	0.6	0.8	0.4	0.6	0.8	0.4	0.6	0.8	0.4	0.6	0.8	0.4	0.6	0.8
atm																				×	×			×			×			×			
buybook																								×									
dslservice								×	×			×		×	×			×					×	×			×			×			
gymlocker								×	×		×	×		×	×		×	×	×	×	×	×	×	×		×	×		×	×			
loanapproval																					×												
marketplace																				×	×												
purchase				×	×	×	×	×	×					×	×				×	×	×	×	×	×				×	×	×			
triphandling																					×			×						×			

TABLE 6
COMPARISON OF MEAN APFD VALUES
BETWEEN *RANDOM* AND M1–M10

Benchmark	*Random*	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10
atm	0.61	0.61	0.65	0.65	0.66	0.63	0.00	0.04	0.05	0.03	0.78
buybook	0.36	0.36	0.36	0.40	0.38	0.38	0.45	0.33	0.62	0.56	0.60
dslservice	0.48	0.51	0.11	0.18	0.24	0.18	0.64	0.06	0.08	0.07	0.62
gymlocker	0.71	0.70	0.00	0.00	0.00	0.00	0.00	0.00	0.36	0.37	0.83
loanapproval	0.54	0.51	0.47	0.53	0.54	0.54	0.52	0.59	0.68	0.69	0.71
marketplace	0.75	0.78	0.67	0.92	0.72	0.90	0.29	0.56	0.92	0.79	0.93
purchase	0.36	0.10	0.09	0.66	0.10	0.66	0.00	0.00	0.74	0.00	0.66
triphandling	0.77	0.69	0.64	0.88	0.89	0.86	0.68	0.78	0.95	0.96	0.92

TABLE 7
COMPARISON OF MEAN APFD VALUES
BY COVERAGE STRATEGIES

Strategy / Benchmark	Fix		Reschedule		Fix & Reschedule	
	M3	M8	M4	M9	M5	M10
atm	0.65	0.05	0.66	0.03	0.63	0.79
buybook	0.40	0.62	0.38	0.62	0.38	0.60
dslservice	0.18	0.08	0.24	0.08	0.18	0.62
gymlocker	0.00	0.36	0.00	0.36	0.00	0.83
loanapproval	0.53	0.68	0.54	0.68	0.54	0.71
marketplace	0.92	0.92	0.72	0.92	0.90	0.93
purchase	0.66	0.74	0.10	0.74	0.66	0.66
triphandling	0.88	0.95	0.89	0.95	0.86	0.92

TABLE 5 summarizes whether a technique has at least 50 percent of the collected APFD values to be 0 at different evolution points (namely, 40, 60, and 80 percent), which corresponds to a boxplot in which the medium line of a bar is on the x-axis. If this is the case, we put a cross ("✖") in the corresponding cell. For the marked cells, if the upper compartment of a boxplot is clearly visible, as for the case of M6 in the plot for *marketplace* in TABLE 5, we darken the background of the cell.We find that, apart from *Random*, M10 is the only technique that does not have any cell marked with a cross in TABLE 5. All the other techniques have 3 to 12 crosses, which indicate that they can be less desirable than M10. The result seems to indicate that both the traditional techniques (M1, M2, M5, and M6) as well as some PRT-enriched techniques (M3, M4, M5, M8, and M9) may miss to expose failures that should be feasible to be exposed.

We also compare *Random* with M1–M10 in terms of their mean APFD values. Table 6 summarizes the results. If *Random* outperforms M*i* (*i* = 1, 2, ..., 10) by at least 10 percent, we mark the background of the cell in light gray. If M*i* outperforms *Random* by at least 10 percent, we mark the background in blue (or dark gray if viewed in black and white).

We observe from Table 6 that *Random* has achieved better results than most conventional techniques (M1, M2, M6, and M7). *Random* seems to perform comparably with all the three techniques (M3, M4, and M5) in the workflow-based series. M10 always outperforms *Random* while M8 and M9 are close to *Random* in mean effectiveness. As a whole, the XRG-based series is either better than or the same as *Random* in terms of their mean effectiveness. The result also shows that regression testing in an environment that an external service may evolve during a test session can be challenging.

A comparison of the mean APFD values using different (workflow-based and XRG-based) strategies is shown in Table 7. For two techniques using the same strategy, if the workflow-based technique outperforms the XRG-based technique by at least 10 percent, we mark the background of the cell in light gray. If the XRG-based technique outperforms the workflow-based technique by at least 10 percent, we mark the background of the cell in blue (or dark gray if viewed in black and white).

The number of blue cells is much more than the number of cells in light gray. This observation indicates that it can be more effective to involve more types of coverage.

The result of *gymlocker* in Fig. 6 indicates that some of the proposed techniques (such as M3–M7) do not work well, but M10 shows good outcomes. We have further investigated this situation, and found the following: If all the fault-revealing test cases have been executed before the evolution, some PRT-enriched techniques failed to re-add such test cases to supplement testing after evolution (e.g., M3–M7 against *gymlocker*), which misses to detect the failure caused by the evolution. The result of *gymlocker* shows that, after the evolution, M10 can still detect minor changes in coverage, and requires sufficient number of additional test cases from the executed test cases. Therefore, M10 still performs well when some other PRT-enriched techniques do not work.

4.3.2 HYPOTHESIS TESTING

We further apply hypothesis testing to confirm our observations made in the last section.

The t-test assesses whether the means of two groups are statistically different from each other. If the significance value is less than 0.05, the difference among the metric is statistically significant. We summarize in Table 8 the comparison results of three strategies, M3–M5 in one group and M8–M10 in another group. We also highlight the significant entries by setting the background of these cells in gray. We observe that M10 is significantly better than M8 and M9. This observation indicates that a combination of fix and reschedule strategies is better than either the fix or reschedule strategy.

TABLE 8
COMPARISON OF t-TEST RESULTS

Benchmark	Workflow-based			XRG-based		
	M3, M4	M3, M5	M4, M5	M8, M9	M8, M10	M9, M10
atm	0.69	0.80	0.45	0.98	0.00	0.00
buybook	0.68	0.53	0.84	0.42	0.22	0.03
dslservice	0.41	0.16	0.58	0.36	0.00	0.00
gymlocker	0.00	0.00	0.00	0.03	0.00	0.00
loanapproval	0.02	0.12	0.48	0.15	0.08	0.00
marketplace	0.00	0.31	0.00	0.00	0.00	0.00
purchase	0.00	0.48	0.00	0.00	0.02	0.00
triphandling	0.42	0.93	0.39	0.49	0.01	0.06

Since rejecting the hypothesis only indicates that the means of the two groups are statistically different from each other, we further examine Fig. 5c to determine which technique is better. We follow [17] to conduct the *multiple comparison procedure* to study whether the means of the test case prioritization techniques differ from each other at a significance level of 5 percent. We present the

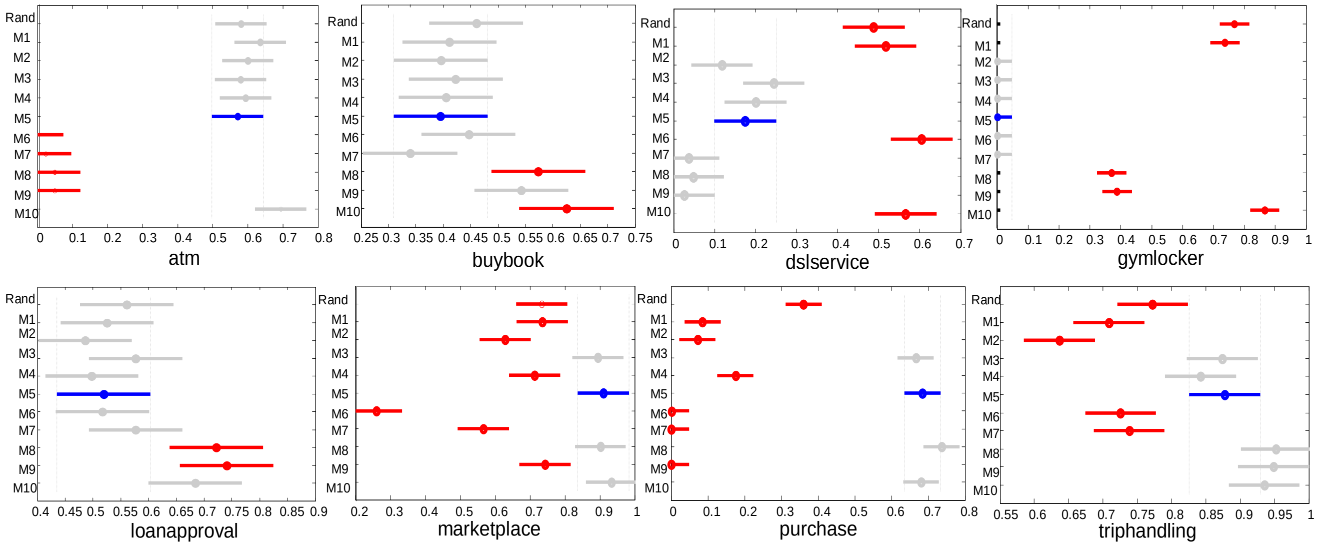


Fig. 7. Multiple comparisons with *Random* and M1–M10 for each subject service.
The *x*-axis represents the APFD values, the *y*-axis represents the test pair selection strategies, and M5 is the target technique.

multiple comparison results for each benchmark in Fig. 7.

In these figures, the blue (darkest) line represents the target technique that we want to compare with others. The gray (lightest) lines represent techniques comparable to the target technique. The remaining (red) lines represent the techniques whose means differ significantly from the target technique.

In Fig. 7, M1 to M10 perform comparably with *Random* in 7, 5, 4, 5, 4, 4, 2, 1, 2, and 3 benchmarks, respectively. Only some techniques can outperform *Random*: 2 for M3, 2 for M5, 4 for M8, 3 for M9, and 5 for M10. Moreover, only M10 never performs worse than *Random*. For other techniques, M1 to M9 perform worse than *Random* by 1, 3, 3, 3, 2, 5, 5, 3, and 3 benchmarks, respectively.

Across the board, at a significance level of 5 percent, we find that M10 is superior to *Random*; M1, M3, M5, M8 and M9 perform comparably with *Random*; whereas M2, M4, M6, and M7 perform worse than *Random*. If we need to find the next candidate, then *Random*, M8, and M9 can be considered. The result also confirms statistically that the XRG-based series can be better than the workflow-based series in terms of the mean APFD.

With respect to the workflow-based series, we find from the figures that M3 and M5 can be superior to M2 (if not comparable in individual benchmarks). We cannot decide whether they are comparable to, better than, or worse than M1.

Comparing between the workflow-based series and the XRG-based series, we find that M3 to M5 are more effective than M8 to M10 for 2, 4 and 2 benchmarks, the same in effectiveness for 3, 2, and 4 benchmarks, and less effective for 0, 3, and 5 benchmarks.

4.3.3 Threats to Validity

Construct validity relates to the metrics used to evaluate the effectiveness of test case prioritization techniques. We adopt the APFD measure to evaluate the techniques. Using other metrics may give different results.

Threats to internal validity are the influences that can affect the dependency of the experimental variables involved. During the execution of a test case, the contexts (such as database status) of individual services involved in a service composition may affect the outcome and give nondeterministic results. In the experiment, we used our tool to reset the contexts to the same values every time before rerunning any test case. This approach was also advocated by agile software development.

External validity refers to whether the experiment can be generalized. The faults in various versions of external services are simulated problematic requests and responses from the environment of the web service under test. The simulated evolution scenarios do not represent all real-life evolution scenarios. The interpolation and extrapolation of the data points to other change windows should be interpreted with care. Our subject service-based applications were based on WS-BPEL. It is not clear how the results of other types of artifacts may look like. In general, having more subjects can strengthen the generalization of the work.

5 Practicality Issues

This section discusses the practicality issues of our work.

In practice, *Random* may not always be a cost-effective technique. We find that although the overall performance of *Random* is good when compared with some PRT-enriched techniques, its performance for individual applications varies. For example, M3, M5, and M8 outperform *Random* by more than 50 percent when applied to *purchase*, and M8 and M9 outperform *Random* by more than 20 percent when applied to *buybook* and *loanapproval*.

If an evolution of service takes place after all the fault-revealing test cases have been executed, a prioritization technique may not reveal faults by executing the remaining test cases. In theory, our PRT-enriched series reruns some of the already-executed test cases and may therefore have the potential to reveal further faults. We believe that a technique with such a potential is much better than another technique without it. In reality, the performance depends on the applications, the faults, and the test suites.

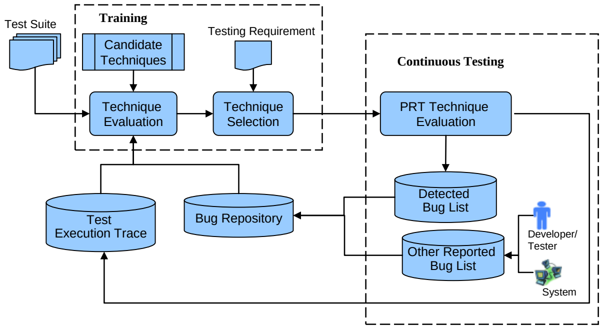


Fig. 8. A potential framework to apply PRT techniques.

In particular, the same technique may report different levels of performance for different applications. For example, M8 and M9 perform well for *loanapproval*, but do not have good results for *gymlocker*. When using our techniques to test a real-life application, it is thus important to find the appropriate technique for the application. For instance, if M8, M9, and M10 have similar performance, using M8 or M9 could be more cost-effective. To find the best technique for a given testing requirement, one possible idea is to use the execution history of the previous test as training data (as depicted in Fig. 8). The selected technique will be used in continuous testing until another candidate technique is found to be more suitable. The training can be triggered either periodically or manually. Forced manual training may be triggered when a developer reports a fault that is unrevealed by the selected technique.

We also note that the PRT-enriched techniques can be very sensitive to minor changes and are dependent on the external services in the field. When a change of coverage is detected, the PRT-enriched techniques will verify the application until the entire test suite has been executed and no test case changes its coverage between the current and the last executions. In practice, developers may set a ceiling for the number of test cases to be verified.

The assumption of our PRT-enriched techniques is stated in constraint (2) in Section 3.2. For the regression testing of services that satisfy constraint (2), our PRT strategies can be applied. When there are uncontrollable changes, PRT strategies can be directly used. When there are controllable or known changes, PRT strategies can be incorporated with existing change-based prioritization strategies.

Our strategies can work together with test case generation and test suite reduction [40]. When a workflow branch is deleted, for instance, the coverage information of test cases used to cover this branch will change. Our strategies can detect such a change and will try to search for other test cases to fix the coverage. Therefore, if the branch is wrongly deleted, the test cases used to cover the deleted branch may help reveal the fault. Otherwise, if the branch is correctly deleted, we can conduct test suite reduction before applying our strategies. Since existing test cases may not be able to cover newly added workflow branches, new test cases need to be generated before applying our strategies.

6 Related Work

Regression testing has been extensively studied [40]. This section reviews the work that is closely related to our study, in the context of stopping criterion, revision identification, test case prioritization, code coverage, external services, execution monitoring, test oracles, cloud-based service testing, and service environment evolution.

A feature of PRT strategies is the use of a stopping criterion. Mei et al. studied both traditional test adequacy criteria [25] and new dataflow-based test adequacy criteria [24] to test WS-BPEL web services. Casado et al. [9] used a classification-tree methodology [12], [15] to determine the test coverage of web services transactions.

PRT strategies do not require knowing whether the service under test undergoes any revision. However, if the details of service revision are available, one may further refine such a criterion by only picking test cases that affect the changed part of the service under test: Ruth and Tu [31], Chen et al. [11], Li et al. [19], Liu et al. [21], and Tarhini et al. [34] conducted such impact analysis on web service implementations to identify revised fragments of code in a service by comparing the flow graph of the new version with that of the previous version. Li et al. [19], [35] explored the use of messages to obtain a comprehensive view of service behavior in selecting paths for regression testing. Liu et al. [21] studied the changes in the concurrency control in BPEL process executions. Tarhini et al. [34] exploited impact analysis from a model-based perspective.

Quite a number of test case prioritization techniques have been proposed. Hou et al. [16] also observed the need to test service-oriented applications with external services. Their techniques consider invocation quotas to constrain the total number of requests for specific web services. Mei et al. considered both black-box coverage [26] and gray-box coverage [22], [23], [28] in test case prioritization. They did not, however, consider the need for dynamic changes in test case ordering based on the feedback collected from the service under regression test. Nguyen et al. [29] integrated test case prioritization with audit testing to control resource consumption. Zhai et al. [41] used the dynamic features of service selection to reduce the service invocation cost, and they further studied [42] different diversity strategies to reorder test cases. Chen et al. [11] prioritized test cases based on the weights thus identified. However, the evolution of external web services was not considered in the above work.

Our work uses dynamic coverage data of the BPEL process achieved by test cases against both the original web service and the evolved web service(s) to determine adequacy. The closest related work in this aspect is Zou et al. [47], who studied an integration of coverage of program statements and HTML elements for testing dynamic web applications. It is akin to our concept of coverage of BPEL process and WSDL documents. On the other hand, their work has not studied whether the test cases applied to the evolved version of the web service produce compatible results as the original service. Becker et al. [6] described a technique to check whether a service description is backward-compatible. Their technique can significantly enhance the practicability of our techniques, while our techniques complement their work.

Sometimes, regression test suites require augmentation (such as fixing some errors and adding new test cases). Bai et al. [3] and Bartolini et al. [5] generated test cases that conformed to WSDL schemas. Belli et al. [7] and Zheng et al. [44] each studied a model-based approach to construct both abstract and concrete test cases. Li et al. [20] studied the generation of control-flow test cases for the unit testing of BPEL programs. de Almeida and Vergilio [13] as well as Xu et al. [39] perturbed inputs to produce test cases for robustness testing.

Our present work monitors the execution traces of services under test. Bartolini et al. [4] extracted state machine information based on messages from opaque web services. Their technique can be integrated with the work presented in this paper to identify changes in external services during regression testing. Ni et al. [30] proposed to model a WS-BPEL web service as a message-sequence graph and attempted to coordinate messages among message sequences to control the message passing mechanism on WS-BPEL execution.

When applying a regression test case to an evolved version of a web service, we have assumed that the execution result can indicate whether a failure has been revealed. Although it is a typical assumption made by many pieces of regression testing research, in practice, the results may require further examination in order to reveal failures. Both Chan et al. [10] and Sun et al. [33] studied the use of metamorphic relations to address this problem.

In recent years, there is an increasing effort to move the testing platform to cloud environments. For instance, Zhu [45] as well as Zhu and Zhang [46] each proposed a framework that integrates different test components wrapped as a web service to realize testing techniques using service-oriented approaches. It is unclear to what extent our work may benefit from the use of such a platform, which warrants further research.

Our present paper is not the only work that addresses the changing nature of the service environment. Zhang [43], for instance, also considered that web services are changing, and proposed to use mobile agent to select reliable web services for testing. However, her work has not considered the regression testing scenarios.

7 Conclusion

This paper has proposed *Preemptive Regression Testing*, a novel strategy that reschedules test cases in a planned regression test session based on the changes of the service under test detected in the course of each actual regression test session. It has proposed three particular PRT strategies, integrated with existing test case prioritization techniques to generate new techniques. Experiments have shown that handling changes in external services in a regression test session by existing techniques and some PRT-enriched techniques can be challenging. We have identified one PRT-enriched technique that has the potential to cope with such changes.

We will generalize PRT further to devise other strategies and integrate PRT with QoS testing.

Acknowledgments

This research was supported in part by the National Natural Science Foundation of China (project no. 61202077), the CCF-Tencent Open Research Fund (project no. CCF-TencentAGR20130111), the National High Technology Research and Development Program of China (863) (project no. 2011AA01A204), the National Science and Technology Infrastructure Program (project no. 2012ZX01039-004), and the Early Career Scheme and the General Research Fund of the Research Grants Council of Hong Kong (project nos. 111410, 123512, 717811, and 716612).

References

[1] *alphaWorks Technology: BPEL Repository*, IBM, 2006, https://www14.software.ibm.com/webapp/iwm/web/preLogin.do?source=AW-0KN.

[2] J.H. Andrews, L.C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" *Proceedings of the 27th International Conference on Software Engineering* (*ICSE '05*), pp. 402–411, 2005.

[3] X. Bai, W. Dong, W.-T. Tsai, and Y. Chen, "WSDL-based automatic test case generation for web services testing," *Proceedings of the IEEE International Symposium on Service-Oriented System Engineering* (*SOSE '05*), pp. 207–212, 2005.

[4] C. Bartolini, A. Bertolino, S.G. Elbaum, and E. Marchetti, "Bringing white-box testing to service oriented architectures through a service oriented approach," *Journal of Systems and Software*, vol. 84, no. 4, pp. 655–668, 2011.

[5] C. Bartolini, A. Bertolino, E. Marchetti, and A. Polini, "Towards automated WSDL-based testing of web services," *Service-Oriented Computing* (*ICSOC '08*), pp. 524–529, 2008.

[6] K. Becker, J. Pruyne, S. Singhal, A. Lopes, and D. Milojicic, "Automatic determination of compatibility in evolving services," *International Journal of Web Services Research*, vol. 8, no. 1, pp. 21–40, 2011.

[7] F. Belli, A.T. Endo, M. Linschulte, and A. Simão, "A holistic approach to model-based testing of web service compositions," *Software: Practice and Experience*, vol. 44, no. 2, pp. 201–234, 2012.

[8] *Eclipse Environment Implementation of the Business Process Execution Language Engine* (*BPWS4J Engine 2.1*), http://en.pudn.com/downloads53/sourcecode/middleware/detail184250_en.html.

[9] R. Casado, M. Younas, and J. Tuya, "Multi-dimensional criteria for testing web services transactions," *Journal of Computer and System Sciences*, vol. 79, no. 7, pp. 1057–1076, 2013.

[10] W.K. Chan, S.C. Cheung, and K.R.P.H. Leung, "A metamorphic testing approach for online testing of service-oriented software applications," *International Journal of Web Services Research*, vol. 4, no. 2, pp. 60–80, 2007.

[11] L. Chen, Z. Wang, L. Xu, H. Lu, and B. Xu, "Test case prioritization for web service regression testing," *Proceedings of the 5th IEEE International Symposium on Service Oriented System Engineering* (*SOSE '10*), pp. 173–178, 2010.

[12] T.Y. Chen and P.-L. Poon, "On the effectiveness of classification trees for test case construction," *Information and Software Technology*, vol. 40, no. 13, pp. 765–775, 1998.

[13] L.F. de Almeida, Jr. and S.R. Vergilio, "Exploring perturbation based testing for web services," *Proceedings of the IEEE International Conference on Web Services* (*ICWS '06*), pp. 717–726, 2006.

[14] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *IEEE Transactions on Software Engineering*, vol. 32, no. 9, pp. 733–752, 2006.

[15] M. Grochtmann and K. Grimm, "Classification trees for partition testing," *Software Testing, Verification and Reliability*, vol. 3, no. 2, pp. 63–82, 1993.

[16] S.-S. Hou, L. Zhang, T. Xie, and J.-S. Sun, "Quota-constrained test-case prioritization for regression testing of service-centric systems," *Proceedings of the IEEE International Conference on Software Maintenance* (*ICSM '08*), pp. 257–266, 2008.

[17] B. Jiang, Z. Zhang, W.K. Chan, and T.H. Tse, "Adaptive random test case prioritization," *Proceedings of the 24th IEEE / ACM International Conference on Automated Software Engineering* (*ASE '09*), pp. 233–244, 2009.

[18] M.B. Juric, *A Hands-on Introduction to BPEL, Part 2: Advanced BPEL*, Oracle Technology Networks, http://www.oracle.com/technetwork/articles/matjaz-bpel2-082861.html.

[19] B. Li, D. Qiu, H. Leung, and D. Wang, "Automatic test case selection for regression testing of composite service based on extensible BPEL flow graph," *Journal of Systems and Software*, vol. 85, no. 6, pp. 1300–1324, 2012.

[20] Z. Li, W. Sun, Z.B. Jiang, and X. Zhang, "BPEL4WS unit testing: framework and implementation," *Proceedings of the IEEE International Conference on Web Services* (*ICWS '05*), pp. 103–110, 2005.

[21] H. Liu, Z. Li, J. Zhu, and H. Tan, "Business process regression testing," *Proceedings of the 5th International Conference on Service-Oriented Computing* (*ICSOC '07*), pp. 157–168, 2007.

[22] L. Mei, Y. Cai, C. Jia, B. Jiang, and W.K. Chan, "Prioritizing structurally complex test pairs for validating WS-BPEL evolutions," *Proceedings of the IEEE International Conference on Web Services* (*ICWS '13*), pp. 147–154, 2013.

[23] L. Mei, Y. Cai, C. Jia, B. Jiang, and W.K. Chan, "Test pair selection for test case prioritization in regression testing for WS-BPEL programs," *International Journal of Web Services Research*, vol. 10, no. 1, pp. 73–102, 2013.

[24] L. Mei, W.K. Chan, and T.H. Tse, "Data flow testing of service-oriented workflow applications," *Proceedings of the 30th International Conference on Software Engineering* (*ICSE '08*), pp. 371–380, 2008.

[25] L. Mei, W.K. Chan, T.H. Tse, and F.-C. Kuo, "An empirical study of the use of Frankl-Weyuker data flow testing criteria to test BPEL web services," *Proceedings of the 33rd Annual International Computer Software and Applications Conference* (*COMPSAC '09*), vol. 1, pp. 81–88, 2009.

[26] L. Mei, W.K. Chan, T.H. Tse, and R.G. Merkel, "XML-manipulating test case prioritization for XML-manipulating services," *Journal of Systems and Software*, vol. 84, no. 4, pp. 603–619, 2011.

[27] L. Mei, K. Zhai, B. Jiang, W.K. Chan, and T.H. Tse, "Preemptive regression test scheduling strategies: a new testing approach to thriving on the volatile service environments," *Proceedings of the 36th Annual International Computer Software and Applications Conference* (*COMPSAC '12*), pp. 72–81, 2012.

[28] L. Mei, Z. Zhang, W.K. Chan, and T.H. Tse, "Test case prioritization for regression testing of service-oriented business applications," *Proceedings of the 18th International Conference on World Wide Web* (*WWW '09*), pp. 901–910, 2009.

[29] C.D. Nguyen, A. Marchetto, and P. Tonella, "Test case prioritization for audit testing of evolving web services using information retrieval techniques," *Proceedings of the 2011 IEEE International Conference on Web Services* (*ICWS '11*), pp. 636–643, 2011.

[30] Y. Ni, S.-S. Hou, L. Zhang, J. Zhu, Z.J. Li, Q. Lan, H. Mei, and J.-S. Sun, "Effective message-sequence generation for testing BPEL programs," *IEEE Transactions on Services Computing*, vol. 6, no. 1, pp. 7–19, 2013.

[31] M.E. Ruth and S. Tu, "Towards automating regression test selection for web services," *Proceedings of the 16th International Conference on World Wide Web* (*WWW '07*), pp. 1265–1266, 2007.

[32] A. Sharma, T.D. Hellmann, and F. Maurer, "Testing of web services: a systematic mapping," *Proceedings of the IEEE 8th World Congress on Services* (*SERVICES '12*), pp. 346–352, 2012.

[33] C.-A. Sun, G. Wang, B. Mu, H. Liu, Z. Wang, and T.Y. Chen, "A metamorphic relation-based approach to testing web services without oracles," *International Journal of Web Services Research*, vol. 9, no. 1, pp. 51–73, 2012.

[34] A. Tarhini, H. Fouchal, and N. Mansour, "Regression testing web services-based applications," *Proceedings of the IEEE International Conference on Computer Systems and Applications* (*AICCSA '06*), pp. 163–170, 2006.

[35] D. Wang, B. Li, and J. Cai, "Regression testing of composite service: an XBFG-based approach," *Proceedings of the IEEE Congress on Services Part II* (*SERVICES-2*), pp. 112–119, 2008.

[36] *Web Services Business Process Execution Language Version 2.0: OASIS Standard*, Organization for the Advancement of Structured Information Standards (OASIS), 2007, http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf.

[37] *Web Services Description Language* (*WSDL*) *Version 2.0 Part 1: Core Language*, W3C, 2007, http://www.w3.org/TR/wsdl20/.

[38] *XML Path Language* (*XPath*) *2.0: W3C Recommendation*, W3C, 2007, http://www.w3.org/TR/xpath20/.

[39] W. Xu, J. Offutt, and J. Luo, "Testing web services by XML perturbation," *Proceedings of the 16th International Symposium on Software Reliability Engineering* (*ISSRE '05*), pp. 257–266, 2005.

[40] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[41] K. Zhai, B. Jiang, and W.K. Chan, "Prioritizing test cases for regression testing of location-based services: metrics, techniques and case study," *IEEE Transactions on Services Computing*, vol. 7, no. 1, pp. 54–67, 2014.

[42] K. Zhai, B. Jiang, W.K. Chan, and T.H. Tse, "Taking advantage of service selection: a study on the testing of location-based web services through test case prioritization," *Proceedings of the IEEE International Conference on Web Services* (*ICWS '10*), pp. 211–218, 2010.

[43] J. Zhang, "An approach to facilitate reliability testing of Web services components," *Proceedings of the 15th International Symposium on Software Reliability Engineering* (*ISSRE '04*), pp. 210–218, 2004.

[44] Y. Zheng, J. Zhou, and P. Krause, "An automatic test case generation framework for web services," *Journal of Software*, vol. 2, no. 3, pp. 64–77, 2007.

[45] H. Zhu, "A framework for service-oriented testing of web services," *Proceedings of the 30th Annual International Computer Software and Applications Conference* (*COMPSAC '06*), vol. 2, pp. 145–150, 2006.

[46] H. Zhu and Y. Zhang, "Collaborative testing of web services," *IEEE Transactions on Services Computing*, vol. 5, no. 1, pp. 116–130, 2012.

[47] Y. Zou, C. Feng, Z. Chen, X. Zhang, and Z. Zhao, "A hybrid coverage criterion for dynamic web testing," http://software.nju.edu. cn/zychen/paper/2013SEKEa.pdf.

Lijun Mei received the PhD degree from The University of Hong Kong. He is a staff researcher at IBM Research—China. His research interest includes addressing the issues of program testing and testing management in the business environment. He has conducted extensive research in testing service-based applications.

W.K. Chan is an assistant professor at the City University of Hong Kong. His current main research interests include program analysis and testing for concurrent software and systems. He is on the editorial board of the *Journal of Systems and Software*. He has published extensively in highly reputable venues such as the *ACM Transactions on Software Engineering and Methodology*, *IEEE Transactions on Software Engineering*, *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Services Computing*, *Communications of the ACM*, *Computer*, ICSE, FSE, ISSTA, ASE, WWW, ICWS, and ICDCS. He is a regular PC member of the ICWS series of conferences, and has served as the innovation showcase chair of ICWS/SCC 2009–2010.

T.H. Tse received the PhD degree from the London School of Economics in 1988 and was a visiting fellow at the University of Oxford in 1990 and 1992. He is a professor in computer science at The University of Hong Kong. His current research interest is in program testing, debugging, and analysis. He is the steering committee chair of QSIC and an editorial board member of the *Journal of Systems and Software*, *Software Testing, Verification and Reliability*, *Software: Practice and Experience*, and the *Journal of Universal Computer Science*. He also served on the search committee for the editor-in-chief of the *IEEE Transactions on Software Engineering* in 2013. He is a fellow of the British Computer Society, a fellow of the Institute for the Management of Information Systems, a fellow of the Institute of Mathematics and its Applications, and a fellow of the Hong Kong Institution of Engineers. He was awarded an MBE by The Queen of the United Kingdom.

Bo Jiang received the PhD from The University of Hong Kong. He is an assistant professor at Beihang University. His research interests include the reliability and testing of mobile applications, program debugging, adaptive testing, and regression testing.

Ke Zhai received the bachelor’s and master’s degrees from Northeastern University, Shenyang, China, in 2007 and 2009, respectively. He is currently a technology analyst in Goldman Sachs and working toward the PhD degree at The University of Hong Kong. His research interests include concurrent bug detection and the testing of service-based applications. His work has been published in the *IEEE Transactions on Services Computing*, ISSTA, and ICWS.