Postprint of article in *Proceedings of the 36th Annual International Computer Software and Applications Conference* (*COMPSAC 2012*), IEEE Computer Society, Los Alamitos, CA (2012)

Preemptive Regression Test Scheduling Strategies: A New Testing Approach to Thriving on the Volatile Service Environments*

Lijun Mei
IBM Research – China
Beijing, China
meilijun@cn.ibm.com

Ke Zhai
The University of Hong Kong
Pokfulam, Hong Kong
kzhai@cs.hku.hk

Bo Jiang
Beihang University
Beijing, China
jiangbo@buaa.edu.cn

W. K. Chan**
City University of Hong Kong
Tat Chee Avenue, Hong Kong
wkchan@cityu.edu.hk

T. H. Tse
The University of Hong Kong
Pokfulam, Hong Kong
thtse@cs.hku.hk

***Abstract*—A workflow-based web service may use ultra-late binding to invoke external web services to concretize its implementation at run time. Nonetheless, such external services or the availability of recently used external services may evolve without prior notification, dynamically triggering the workflow-based service to bind to new replacement external services to continue the current execution. Any integration mismatch may cause a failure. In this paper, we propose *Preemptive Regression Testing* (PRT), a novel testing approach that addresses this adaptive issue. Whenever such a late-change on the service under regression test is detected, PRT preempts the currently executed regression test suite, searches for additional test cases as fixes, runs these fixes, and then resumes the execution of the regression test suite from the preemption point.**

Keywords—adaptive service composition, adaptive regression testing, preemptive regression testing, test case prioritization

I. INTRODUCTION

A workflow-based web service may use ultra-late binding to invoke other web services to concretize its implementation at run time. In general, such an implementation is known as a *service-based application* or a *service composition*. Nonetheless, the services that are invoked by the workflow-based web services, which we refer to as *external services*, may evolve without prior notification. The availability of recently used external services for the next service request is also uncertain. Both types of changes dynamically trigger the workflow-based service to bind to replacement external services so that the former service can continue with its execution. The testing of adaptive systems, such as a service composition led by a workflow-based web service, should assure that the workflow-based web service can successfully and dynamically switch its external services and successfully continue with the execution.

For example, during a round of regression test, a test case *t* having been executed on a service composition (such as version *u* of a workflow-based web service and version *v*1 of an external service) *may* require re-execution over the evolved version of the service composition (such as version *u* of the workflow-based web service and an evolved version *v*2 of the external service) so as to test the adaptability of the service composition.

Along any execution path of the workflow-based web service, there is a sequence of service invocation points, each of which leads the workflow-based web service to invoke an external service to provide a result. Moreover, every such service invocation may or may not bind to a new external service. For a "retest-all version" of regression testing, every test case of a regression test suite that goes through this execution trace may potentially be rerun to assure different sequences of binding configurations for the service invocation points. This process is intuitively heavy. On the other hand, testing each test case once irrespective of the possible number of such binding configuration sequences is inadequate.

To the best of our knowledge, existing test case prioritization techniques (such as [2][8][9][14][15][17][18][19][20]) do not take into account the changes in service binding. For instance, in a typical round of regression test, such techniques simply apply *all* the not-yet-executed test cases *once* to the service-based application. They are, therefore, inadequate in testing dynamic web services.

In this paper, we propose a novel approach — ***Preemptive Regression Testing*** (***PRT***) — for the regression testing of (dynamic) web services. We define a *dynamic web service* as a web service that can dynamically change its own processing logic or bind to and use new external services during the course of an execution. We refer to such a change during

** Correspondence author.

execution as a *late-change*. Whenever a late-change is detected, PRT ***preempts*** the currently executed regression test suite, ***searches*** for additional test cases as fixes, ***runs*** these fixes, and then ***resumes*** the execution of the suspended regression test suite from the preemption point. It continues to test the service with the prioritized test suite until all the test cases in the regression test suite have been applied to the service without further detecting any late-change. In this paper, we present three workflow-based strategies, each of which concretizes PRT. They use the workflow coverage of the web service as an indicator of whether a late-change may have occurred. They *adaptively* and *dynamically* prioritize and select test cases from a regression test suite until no change in the workflow coverage of the web service by the selected test cases is detected. We note, however, that PRT is general and not limited to the testing of workflow-based web services. Other PRT strategies following the preemption idea above can be similarly developed.

The basic idea of the three PRT strategies is as follows: Given a test case t of a prioritized regression test suite T that aims to verify a modified version u of a workflow-based web service, a PRT strategy executes u with t and compares the workflow coverage of u achieved by t with that of the last execution of u achieved by t. In case that no prior execution of u is available, the PRT strategy compares the current workflow coverage with that of the preceding version of u achieved by t.

If any change in the workflow coverage of u achieved by t is detected by the above comparison, the PRT strategy immediately preempts the current execution of T, searches T, and identifies a subset X ($\subseteq T$) as *fixes* that collectively covers the missed workflow coverage. The PRT strategy then executes X. Finally, it resumes from the preemption point to continue executing the remaining prioritized test cases in $T \setminus X$. However, unlike existing techniques, the execution of $T \setminus X$ is not the end of the PRT strategy.

In case that some workflow coverage missed by t is really due to the evolved external service binding (say, the use of a version $v2$), any test case in T executed before t can only assure u in the presence of a service other than $v2$ — none of such test cases, in fact, has assured u in the presence of $v2$. The PRT strategy thus re-executes *all* such test cases. The above procedure will continue until the entire T has been executed on u (in a round-robin manner) and no more change in workflow coverage is detected during the execution of the entire test suite T. This is because, by then, the strategy has no further evidence indicating that any external service used by the service-based application has changed.

Intuitively, if the service environment of a web service under test is violated, the PRT approach will conduct a "*long*" regression testing, which is different from existing regression testing techniques that may terminate too early, so that the service adaptation characteristics of the service composition cannot be thoroughly tested by the regression test suite.

We further note that existing test case prioritization strategies wait until the next round of regression test to find test cases based on the new coverage profiles. They are both unaware of the *missed* coverage for a particular service composition and unable to schedule target test cases to verify the present service composition in time. In our PRT strategies, a test case may be executed multiple times during the *same* round of regression testing. Furthermore, a regression test suite does not need to be completely executed before any test case is selected for re-execution. Hence, our PRT approach can be more lightweight than existing regression testing techniques in rescheduling test cases.

We conduct an empirical study using all the subjects from [14][17], and include a comparison between peer techniques [17][19] and new techniques built on top of our strategies. Our study confirms that our techniques are significantly more lightweight than existing techniques.

The main contribution of this paper is threefold: (i) We propose preemptive regression testing, which is a new approach in continuous regression testing to assuring service-based applications that address the challenges due to the presence of external services that may evolve or are of low availability. (ii) We concretize PRT to formulate three strategies. (iii) We present the first empirical study on the efficiency and effectiveness of techniques for continuous regression testing of services.

The rest of this paper is organized as follows: Section II gives a motivating example. Section III presents our strategies and our regression testing techniques. Section IV reports an empirical study, followed a review of related work in Section V. Finally, Section VI concludes the paper.

II. MOTIVATING EXAMPLE

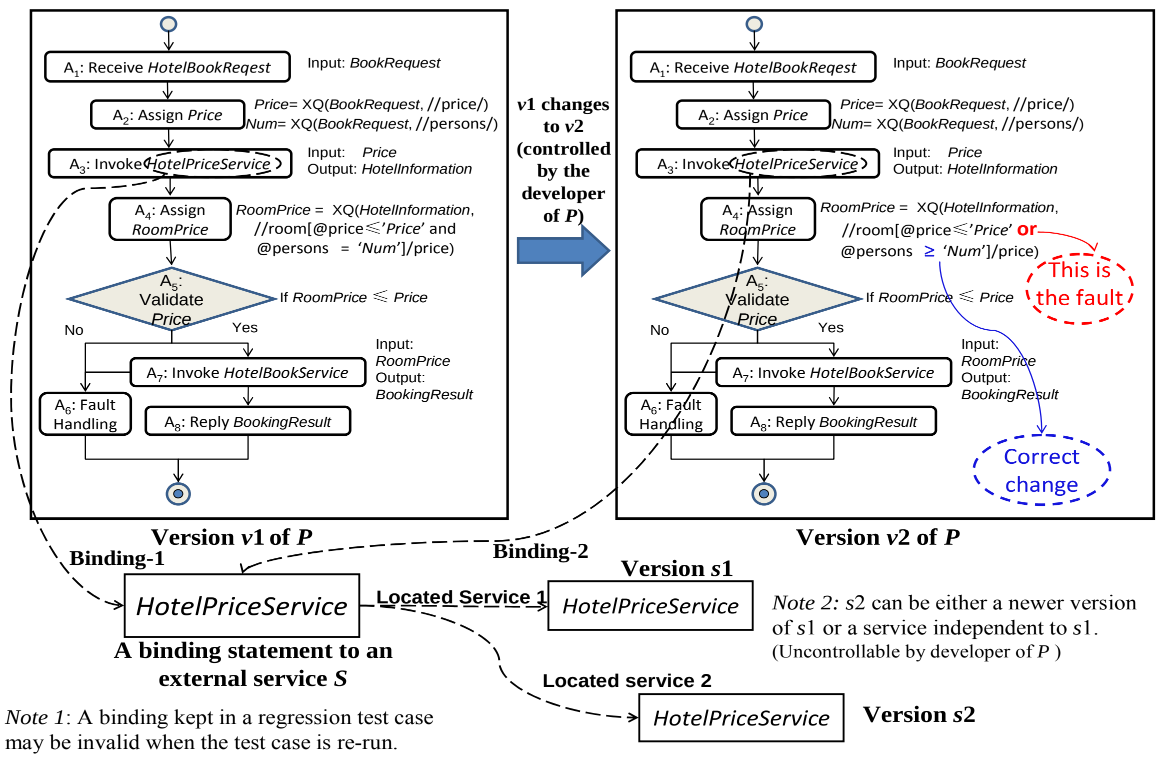
This section adapts an example from the *TripHandling* project [1] to illustrate the new challenges in regression testing of a web service such that its communicating external services may evolve during the execution of a prioritized regression test suite. For ease of presentation, we follow [14][17] to use an activity diagram to depict this web service, which we denote as P.

Fig. 1 shows a scenario where the developer of P modifies version $v1$ into version $v2$. Version $v1$ communicates with version $s1$ of an external hotel price enquiry service (denoted by S). Version $s1$ is, however, not controlled by the developer of P. The binding of version $s1$ of S to $v2$ of P is not guaranteed and may change dynamically. For example, the developer of S may modify the implementation and publish a new version $s2$ to replace $s1$. Alternatively, at run time, the quantity-of-service of $s1$ may not be good enough for $v2$, and hence $v2$ finds a replacement service S' and binds to it.

Our target for testing is to assure $v2$. We note that $v2$ has a dynamic adaptive ability; otherwise, it cannot achieve ultra-late binding to external services. 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 [24]. 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 XPath `//price/` and `//persons/` from *BookRequest* and stores them in the variables *Price* and



A1: Receive HotelBookReqest
Input: BookRequest
A2: Assign Price
Price= XQ(BookRequest, //price/)
Num= XQ(BookRequest, //persons/)
A3: Invoke HotelPriceService
Input: Price
Output: HotelInformation
A4: Assign RoomPrice
RoomPrice = XQ(HotelInformation, //room[@price≤'Price' and @persons = 'Num']/price)
A5: Validate Price
If RoomPrice ≤ Price
No
Yes
A7: Invoke HotelBookService
Input: RoomPrice
Output: BookingResult
A6: Fault Handling
A8: Reply BookingResult
Version v1 of P
v1 changes to v2 (controlled by the developer of P)
RoomPrice = XQ(HotelInformation, //room[@price≤'Price' or @persons ≥ 'Num']/price)
This is the fault
Correct change
Version v2 of P
Binding-1
Binding-2
HotelPriceService
A binding statement to an external service S
Located Service 1
Version s1
Note 2: s2 can be either a newer version of s1 or a service independent to s1. (Uncontrollable by developer of P)
Located service 2
Version s2
Note 1: A binding kept in a regression test case may be invalid when the test case is re-run.

Figure 1. Brief example illustrating dynamic regression testing for dynamic SOA programs.

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 *RoomPrice* using the price extracted via XPath //room[@price≤'Price' and @persons='Num'] /price/.

(e) A_5 verifies locally that the price in *HotelInformation* should not exceed the input *Price*.

(f) If the verification passes, A_7 executes *HotelBookService* to book a room, and A_8 returns the result to the customer.

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

Suppose we have five test cases t_1 to t_5 containing the price (*Price*) and the number of persons (*Num*) as parametric inputs as follows. We assume that only two types of rooms are available, namely, single rooms at a price of \$105 and family rooms (for 3 persons) at a price of \$150.

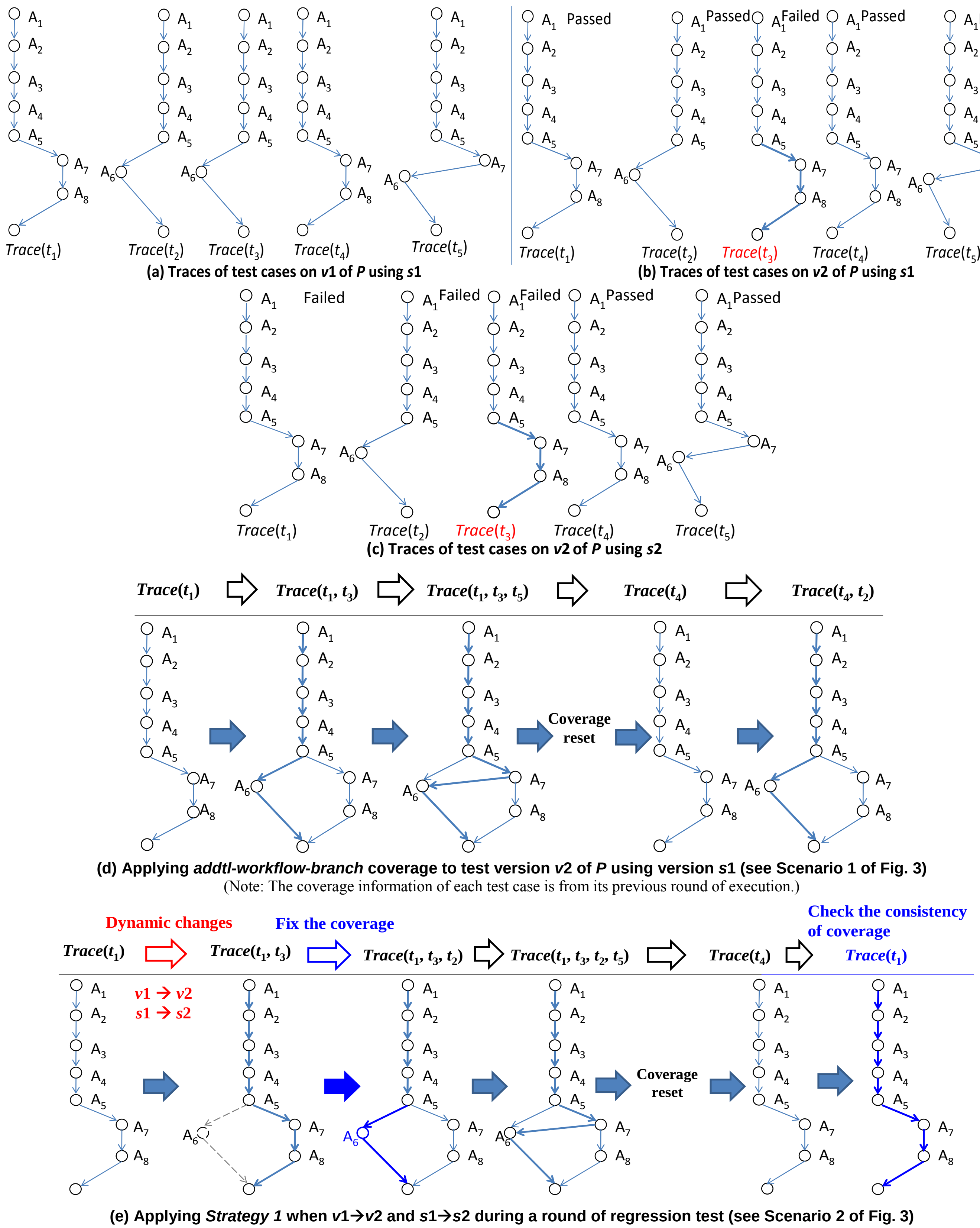
⟨*Price, Num*⟩	⟨*Price, Num*⟩
Test case t_1: ⟨200, 1⟩	Test case t_2: ⟨100, 5⟩
Test case t_3: ⟨125, 3⟩	Test case t_4: ⟨20, 2⟩
Test case t_5: ⟨−1, 1⟩	

Fig. 2(a) shows the execution traces of the five test cases over version *v*1 of *P* that uses version *s*1 of *S* as the hotel price enquiry service. Test case t_1 results in the successful booking of a single room. Test cases t_2 to t_5 results in unsuccessful bookings. The price validation process rejects t_2 and t_3. Since the minimum room price by *HotelQueryService* is \$40, the *HotelInformation* resulting from t_4 and t_5 both include a "no vacancy" notice, the input price, and a default number of persons (set at 99). Thus, although both t_4 and t_5 passes the price validation process, no room can be booked using the low price of t_4, while the price "–1" of t_5 will trigger a fault.

Suppose a software engineer Jim decides to make the following changes to the precondition at node A_4 of version *v*1 of *P* in Fig. 1. 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 as indicated in version *v*2 of *P* in Fig. 1. Fig. 2(b) shows the traces of version *v*2 of *P* using version *s*1 of *S*, whereas Fig. 2(c) shows the traces of *v*2 of *P* using *s*2 of *S*.

Note that only the execution trace of t_3 is different among Fig. 2(a)–(c). 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. Moreover, suppose that t_1 and t_2 are failed test case on *s*2 due to the incorrect implementation of *s*2; whereas they are both passed test cases in Fig. 2(b).

In Fig. 2(d), we present a test case permutation generated by the existing *addtl-workflow-branch* test case prioritization technique, which is a traditional strategy adopted from *addtl-statement* coverage from [6][19]. The test suite detects a failure by the second test case (t_3). Suppose, further, that *s*1



(a) Traces of test cases on *v*1 of *P* using *s*1

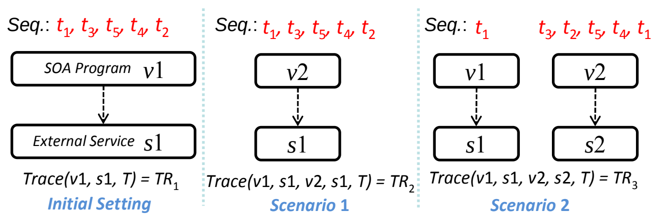
(b) Traces of test cases on *v*2 of *P* using *s*1

(c) Traces of test cases on *v*2 of *P* using *s*2

(d) Applying *addtl-workflow-branch* coverage to test version *v*2 of *P* using version *s*1 (see Scenario 1 of Fig. 3)
(Note: The coverage information of each test case is from its previous round of execution.)

(e) Applying *Strategy 1* when *v*1→*v*2 and *s*1→*s*2 during a round of regression test (see Scenario 2 of Fig. 3)
(Note: The coverage information of each test case is from its latest execution.)

Figure 2. Brief example illustrating the preemptive regression testing of service-based applications.


Seq.: t_1, t_3, t_5, t_4, t_2
SOA Program $v1$
External Service $s1$
Trace(v1, s1, T) = TR_1
Initial Setting
Seq.: t_1, t_3, t_5, t_4, t_2
$v2$
$s1$
Trace(v1, s1, v2, s1, T) = TR_2
Scenario 1
Seq.: t_1
t_3, t_2, t_5, t_4, t_1
$v1$
$v2$
$s1$
$s2$
Trace(v1, s1, v2, s2, T) = TR_3
Scenario 2

Scenario 1: SOA program P changes from $v1$ to $v2$ using $s1$

Scenario 2: Dynamic changes ($v1 \rightarrow v2$, $s1 \rightarrow s2$) during a round of regression test

Figure 3. Two scenarios of test case execution sequences.

evolves to $s2$ during the execution of the entire test suite. This test suite can detect failures by the first (t_1), the second (t_3), and the fifth (t_2) test cases. However, if $s1$ evolves to $s2$ after the execution of t_1 (as in scenario 2 of Fig. 3), at least one failure cannot be detected.

We further examine the scenarios in Figs. 2 and 3 carefully, and observe at least three problems:

- The final binding configuration of the application includes version $s2$. If the binding to $s1$ is replaced by that of $s2$ after some of the test cases have been executed (as in scenario 2 of Fig. 3), not all prioritized test cases will be executed on this configuration, defying the objective of test case prioritization to reorder test cases but not to discard any test case.
- To rerun the test suite, since the workflow coverage has been modified, a traditional prioritization algorithm can only be rerun after completing the entire previous execution of the prioritized regression test suite, which can be non-responsive to the change in binding.
- Should we suspend the execution of a test suite during re-prioritization?

Our PRT Technique. We now illustrate one technique that uses Strategy 1 (to be presented in Section III) to address the above problems due to dynamic changes after the execution of the first test case.

We observe from Fig. 2(e) that, although t_3 is targeted for covering A_6, it actually covers A_7 and A_8. PRT Strategy 1 is to select test cases from the ordered test suite to assure the correctness of A_6 immediately. In Fig. 2(e), we illustrate that the strategy selects t_2 as a replacement according to the given priority, and discovers 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 and t_3 are executed before the latest invocation of Strategy 1. Therefore, the technique reruns these two test cases (based on the same test case priority for the sake of simplicity) and discovers another failure when executing t_1. During the realization of Strategy 1, there is no need to suspend test case executions.

III. Preemptive Regression Testing

This section presents our ***Preemptive Regression Testing* (*PRT*)** approach and formulates three strategies for PRT.

A. *Preliminaries*

A web-based application is divided into two parts. The first part is a web service under test, denoted by P. We assume that P adopts a dynamic service-oriented architecture that can bind a service invocation point to different web services during the course of execution. Similar to a lot of regression testing research [8][11][17][18][19], our objective is to safeguard P from faulty modification of its implementation of the internal computation and dynamic service binding configurations. Testers may test P in a laboratory to collect the coverage data.

The second part is a set of services outside the service under test P. P needs to bind to and 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 over a given version u of P may involve the invocation of external services and obtaining their results.

In general, a web service P has no control over the evolution of external services, the availability of each external service, and the network conditions. It is unrealistic to assume that external services remain unchanged during any round of execution of a regression test suite in the real-world situation.

B. *PRT*

The idea of PRT is intuitive. Whenever a late-change is detected by service P, PRT preempts the currently executed regression test suite, searches for a subset of the regression test suite as a fix, runs the fix, and then resumes the execution of the regression test suite from the preemption point. PRT also continues to execute the remaining part of the prioritize test suite (possibly marking the test cases in the fix as executed) until all test cases have been executed without any preemption among them.

There are many aspects that a PRT technique can use:

A late-change can be detected at the URL level or from the message header of some service message. However, if they fail to provide clues, testers may still need techniques to help them assure the web services under test. An approximate condition is that such a late-change will lead to a change in the implementation of the external service with respect to the service under test. The behavior of the new implementation may alter the execution flow of the service under test. Such change in execution flow can be viewed as a rough indicator for late-changes. Based on this insight, we formulate three PRT strategies to be presented in the next subsection.

A preemption can immediately occur or put in a priority queue and handled similar to how an operating system handles software interrupts. A search for fix can be expanded to the generation of new test cases. Alternatively, it may source test cases that are not originally in the regression test suite. The execution of a fix and the marking of test cases that serve as fixes can also be handled similarly to how a preemption can be handled. The execution of the whole test suite can continue as the stopping criterion can be further modified, such as considering all preemption points rather than merely the last one, or none of them.

C. *Our Coverage-Based PRT Strategies*

In this section, we present three PRT strategies.

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.

In addition, executing such a replacement test case may discover additional coverage items that have been missed as well. In this case, Strategy 1 will suspend its current round, invoke a new round of Strategy 1, wait for this newly invoked round to complete, remove from *F* of the current round those coverage items already covered by the recursively invoked rounds of Strategy 1, and then resume the execution of the current round 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 (see *addtl-statement* coverage in [6] for details).

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 executed by Strategy 1, it will invoke Strategy 2.

Compared with existing strategies, our strategies require additional storage so that we can mark the end of each iteration of a strategy. In the 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 space requirement is light.

Determining the workflow coverage after the execution of every test case will cause additional overhead. However, the computation of workflow coverage changes can be done by comparing the current test trace with the previous one, which is fast. The runtime slowdown factor on the workflow process can be small because a workflow process tends to be small and the major delay appears to be the time needed to wait for the results from external services.

Intuitively, Strategy 1 only requires fixing the coverage, while Strategy 2 requires reprioritization of the not-yet-executed test cases. Therefore, Strategy 2 involves more slowdown overhead than Strategy 1. When the number of test cases per iteration increases (where the term "iteration" has the same meaning as that in *addtl-statement* coverage), the reprioritization definitely takes more time to complete. However, the procedure does not need to be conducted on the whole test suite. Therefore, the reprioritization only depends on the number of test cases executed in each iteration, regardless of the size of the whole regression test suite.

Although our strategies require computation and storage costs, such costs are less than those of retesting all the test cases in a test suite, which involves invoking external services, arranging resources, and even human interactions.

D. *Implementation*

This section describes the application of our strategies to build three test case prioritization techniques (listed as M1–M3 in Table 1).

1) *Our PRT Techniques*

We apply our three strategies to the existing additional-branch technique [19] (also known as *addtl-workflow-branch* coverage in [17]) to build three new evolution-aware techniques (M1─M3 in Table 1). Each new technique has a stopping criterion: For a web service *P* and a regression test suite *T* for *P*, the technique will stop applying test cases to *P* if every test case in *T* results in no further change in the workflow coverage of *P*.

TABLE 1. CATEGORIES OF TEST CASE PRIORITIZATION TECHNIQUES

Name	Reference
Addtl-Workflow-Branch-Fix	M1
Addtl-Workflow-Branch-Reschedule	M2
Addtl-Workflow-Branch-FixReschedule	M3

M1 (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 the *addtl-workflow-branch* coverage [17] to generate a sequence of test cases. *Phase 2: runtime adjustment*. Right after the execution of a test case, it runs Strategy 1 and then continues to apply the given 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.

M2 (Addtl-Workflow-Branch-Reschedule). This technique consists of two phases: *Phase 1: preparation*. This phase is the same as Phase 1 of M1. *Phase 2: runtime adjustment*. It is the same as Phase 2 of M1, except that it runs Strategy 2 rather than Strategy 1.

M1 only deals with test cases that miss to cover some items that have been covered in the last executions of the test cases, whereas M2 only deals with test cases that cover more items than those covered in the last executions of the test cases. The following technique strikes a balance between M1 and M2 by using Strategy 3.

M3 (Addtl-Workflow-Branch-FixReschedule). This technique also consists of two phases. *Phase 1: preparation*. This phase is the same as Phase 1 of M1. *Phase 2: runtime adjustment*. It is the same as Phase 2 of M1, except that it runs Strategy 3 instead of Strategy 1.

In some regression testing techniques (such as [10]), new test cases need to be introduced into the original test suite to check the revised code. Such consideration is not in the scope of the present paper. However, it is not difficult to adapt our PRT approach to use the new test cases. Here is one possible solution: We first categorize the new test cases into a few new iterations. After executing the test cases of an existing iteration, we will then execute the new iterations.

IV. Evaluation

This section evaluates our PRT techniques.

A. Experimental Setup

We chose a set of eight subject programs to evaluate our strategies, as listed in Table 2. They were representative service-based applications developed in WS-BPEL. This set of applications was also used in previous empirical studies reported in [14][17].

We generated 100 test suites for each application. The statistics of these test suites are shown in Table 3. It presents the maximum, average, and minimum numbers of test suites for each benchmark application.

Table 2. Subject Programs and Their Descriptive Statistics

Subject Ref	Subject 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 3. Statistics of Test Suite Sizes

Size \ Subject	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

Strictly following the methodology in [6], we generated 60 modified versions [17], as shown in Table 2. The fault in any modified version could be detected by some test case in every test suite. 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 obtained the implementation tool of Mei et al. [17], configured it, and used it 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 application (rather than the modified versions). For each application, 1000 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 the test suite over the workflow branches, XRG branches, or WSDL elements. This construction process is also adopted in [6][19]. We successfully generated 100 test suites for each application.

To simulate scenarios in a real dynamic service environment, our setting is that web service modifications, external service evolutions, and test case prioritization and selection may occur concurrently during any round of execution of a test suite. For ease of reference, each of them is called a *change*. We define a *change window* to refer to the time interval between two changes. For ease of comparison, we directly use the number of test case executions to represent the time spent. We set the change windows to $x * |T|$, where $x = 0.2$, 0.4, 0.6, and 0.8, and $|T|$ is the size of the test suite T. (We note that when $x = 1.0$, the entire test suite will be completely executed before any evolution occurs. In this case, our initialized techniques will degenerate to traditional techniques.) For each test suite T, we randomly chose among the modified versions and generated a sequence of versions to simulate a sequence of changes. We set each sequence to consist of 50 changes.

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.

B. Measurement Metrics

A fault in a service composition is only detected when the faulty service composition is being dynamically bound and a test case that can reveal the fault is successfully scheduled by a technique to execute over the faulty service composition. Once the service composition has evolved, the test case may only detect a fault due to another service composition rather than this one.

The first effectiveness measure that we will use is, therefore, the number of test cases successfully scheduled by a technique such that each test case detects a failure from a service composition. For ease of reference, we simply refer this metric to as *the number of fault-revealing test cases*. Using this metric allows us to measure the precision of a scheduling technique. A higher metric value indicates a higher precision.

Many existing test case prioritization experiments use the *Average Percentage of Faults Detected* (*APFD*) [6], which only takes into account the first test case that can detect a fault, regardless of the actual round of regression test that the test case is executed. As we have illustrated in

Section I, service compositions may evolve. The ability of an algorithm to schedule a test case to detect the presence of faults in one version inadequately represents the ability to schedule the same test case to detect such presence in another version of the same composition.

We also measure the number of test cases that a technique needs to reschedule. We refer to this metric to as *the number of reordered test cases*. Note that, in practice, reordering test cases is not merely giving a new index to every test case. Rather, any change in a test schedule means that testers need to make new arrangements for resources, internal (human) users, and service partners accordingly. Reducing the number of reordered test cases is crucial.

C. *Data Analysis*

This section analyzes the results of the empirical study.

1) *Analysis on Precision*

Table 4 presents the mean results of the total number of fault-revealing test cases produced by each technique for the eight subjects within each change window. The cells that indicate noticeable advantages over benchmark techniques are typeset in bold.

First, we find that as the size of a change window increases from 0.2 * |*T*| to 0.8 * |*T*|, the effectiveness (in terms of precision) of our techniques generally increase. This finding is consistent with the expectation on our strategies: As a size of a change window increases, the probability of detecting a change in workflow coverage achieved by at least one test case will, on average, increase. Every such detection will trigger our rescheduling strategy (which is the core difference between our dynamic strategy and traditional static counterparts) to find and apply test cases to verify the service composition within the corresponding change window period.

Second, we observe the M2 is less effective than M1 and M3. The result may indicate that Strategy 1 can be more effective than Strategy 2.

Finally, the difference between M1 and M3 is small. The result indicates that adding Strategy 2 on top of Strategy 1 has no noticeable effect.

TABLE 4. PRECISION COMPARISON

Technique	**Change Window between Test Cases**			
	0.2*|*T*|	**0.4*|*T*|**	**0.6*|*T*|**	**0.8*|*T*|**
M1	105.4	213.5	324.1	434.9
M2	103.0	207.3	314.4	419.3
M3	105.6	212.9	324.3	435.8

2) *Analysis on Efficiency*

Since the test cases used in each technique are the same throughout the empirical study, in order to compare the efficiency, we study the number of reordered test cases incurred by each technique.

Random ordering simply selects a test case from the whole test suite randomly, and hence there is no additional reordering cost. Table 5 shows the results of M1 to M3, in which we normalize each result by the mean number of reordered test cases achieved by disabling any strategy (which essentially renders M1–M3 into the same technique, referred to as *Disabled* in the rest of this paper) when the change window is 0.2 * |*T*|. We have typeset in bold those cells that correspond to fewer reordering test cases than those of *Disabled*.

TABLE 5. EFFICIENCY COMPARISON

Technique	**Change Window between Test Cases**			
	0.2*|*T*|	**0.4*|*T*|**	**0.6*|*T*|**	**0.8*|*T*|**
Disabled	1.000	2.000	3.000	4.000
M1	0.950	1.417	1.775	2.105
M2	0.137	0.174	0.216	0.251
M3	1.238	1.916	2.477	2.980

Table 5 shows that M1−M3 significantly reorder fewer test cases than *Disabled* in almost all the cells. The only exception is M3 when the change window size is 0.2 * |T|. The result shows that, in general, our strategies are more *lightweight* than *Disabled*. Overall speaking, they save around 48% of test cases reordering incurred by *Disabled*.

We also observe that, as the size of a change window increases, M1−M3 reorder increasingly smaller ratios of test cases with respect to *Disabled*. This result is encouragingly.

We further observe that M2 is particularly efficient. From Table 4, on average, it only incurs 9.0% of the total number of test cases needed to be reordered by *Disabled*. This saving is significant.

3) *Hypothesis Testing*

We apply hypothesis testing to the raw data for Table 4 to identify the differences among different techniques. We only show the hypothesis testing results for the change window of 0.8*|*T*| in Table 6. The results of 0.2*|*T*|, 0.4*|*T*|, and 0.6*|*T*| are similar. We omit then owing to page limit. The hypothesis testing results for the change windows of other sizes are similar and consistent with the results in Table 6.

TABLE 6. STUDENT'S *t*-TEST RESULTS FOR COMPARISON
(USING THE CHANGE WINDOW OF 0.8 * |*T*|)

	Our Techniques		
	M1	M2	M3
Disabled	**0.03**	0.20	**0.02**

Student's *t*-test assesses whether the means of two groups are statistically different from each other. If the significance is less than 0.05, the difference is statistically significant. We summarize the results in Table 6. The cells that indicate significant differences from benchmark techniques are typeset in bold.

Table 6 shows that M1 and M3 are statistically different from *Disabled*. However, we cannot find significant differences between M2 and *Disabled*. Rejecting the null hypothesis only indicates that the means of the two groups are statistically different from each other. We further examine Table 4 to determine which technique is better.

In short, Table 4 and the hypothesis testing result indicate that the *fix* strategy can be more effective than the *reschedule* strategy. At the same time, Table 5 shows that M2 is more efficient. Our empirical analysis concludes that a *clear* tradeoff between effectiveness and efficiency exists in at least a class of dynamic test case scheduling strategies (represented by our strategies) that supports service regression testing in the open environment. Furthermore, M1–M3 are more lightweight than *Disabled*.

4) *Threats to Validity*

This section discusses the threats to validity of the experiment.

Construct validity relates to the metrics used to evaluate the effectiveness of test case prioritization techniques. In the experiment, we propose two metrics to evaluate our techniques from the perspective of continuous regression testing. Using other metrics may give different results. We have explained the rationales of the two metrics. A risk of the current experiment is that it has not measured the APFD values of the techniques under study. We have explained that the faults in various versions of external services are to simulate problematic requests and responses from the environment of the web service under test. Such faults in the simulated artifacts cannot be meaningfully located by the testers of the web service under test. Hence, we do not proceed to measure APFD.

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 follow [13] to use a tool to reset the contexts to the same values every time before rerunning any test case. This approach is also advocated by agile software development. Another threat in our experiment is the scheduler used in determining the evolution sequence. The use of another scheduler may affect the results. To reduce bias, we use a random scheduler to generate of such schedules. However, the random scheduler may produce quite many diverse schedules. To address this threat, we have used a fairly large number of test suites (100 test suites per subject) in each of the four change window setting. As indicated by Table 3, the average number of test cases per test suite is about 88. We believe that we have collected a sufficiently large pool of data to measure the central tendency in terms of precision and efficiency.

External validity refers to whether the experiment can be generalized. The current subject programs are not large, even though they have been used in [14][17]. The use of service-based applications with other binding characteristics may produce different results. We only use four change windows in the empirical study. The interpolation and extrapolation of the data points to change windows of other sizes may not be applicable. Our subjects are based on WS-BPEL, and the results of other types of artifacts are still unclear. We have only used three particular instances of PRT techniques. The results thus obtained should be interpreted with care before evaluating PRT in general. Our experiment has not compared with other techniques except "*Disabled*", which is basically the traditional additional test case prioritization strategy. Comparisons with other strategies as baselines can help make PRT more mature.

V. Related Work

This section reviews other work related to our proposal.

The project most relevant to this proposal is the work in Mei et al. [17]. It proposed to prioritize test cases based on the workflow coverage achieved by a test suite over a preceding version of a modified application. It built its techniques on top of the earlier data flow testing work [14], which aims to reveal the potential integration of messages and code through XML Schemas and XPaths. Like classical test case prioritization techniques [18][19], the techniques in Mei et al. [17] are unaware of any evolution of external services and do not reschedule test cases that have not been executed to assure a modified application in the potential presence of newer versions of external services.

Many existing techniques for unit and integration testing of service-oriented programs have been proposed. Bartolini et al. [3] discussed potential ways to apply data flow testing to service composition. They also proposed a framework to facilitate the collection of coverage summaries of test executions of service-oriented programs. Mei et al. modeled the combination of XPath and WSDL as an XRG, and developed data flow testing techniques to verify services that manipulates XML messages [14] and services that interact through XML messages [15].

Hou et al. [8] also observed the need to test service-oriented applications that invoke external services. They added invocation quotas to constrain the number of requests for specific web services, and then developed techniques to prioritize test cases to maximize the test requirement coverage under such quota constraints. They have not observed that an external service may evolve during a round of regression test of the modified application. Ruth and Tu [20] and Chen et al. [5] conducted impact analysis on web services. They aimed to identify revised fragments of code in a service by comparing the flow graph of the new version with that of the previous version. Chen et al. [5] also prioritize test cases based on the weights thus identified. Mei et al. [16] also propose an interface-based test case prioritization technique. However, they have not considered the evolution of external services in the course of testing.

Li et al. [12] studied the generation of control-flow test cases for the unit testing of BPEL programs. Fu et al. [7] considered the role of XPath when studying the formal verification of web services. They translated services into Promela, and translated an XPath into a Promela procedural routine using self-proposed variables and code to simulate XPath operations in the web service environment. Chan et al. [4] proposed to use metamorphic testing to alleviate the test oracle issues for stateless web services. Zhai et al. [25] further used the dynamic features of service selection to reduce the service invocation cost. Zhu and Zhang [26] proposed a framework that integrates different test components wrapped as a web service to realize testing techniques using service-oriented approaches.

Finally, we review related regression testing techniques. Leung and White [10] pointed out that simply rerunning all existing test cases in a regression test suite is far from ideal. Many existing techniques for regression test selection (such as [18]) and test case prioritization (such as [19]) selected test cases that are related to the modified edges of the control flow graphs of the revised applications. Kim and Porter [9] proposed to use the history information of different program versions to prioritize test cases. Our techniques are aware of the potential changes in the environment of the application under test and can select the same test case multiple times before every test case in the test suite has been selected.

VI. Conclusion

A service using the dynamic service-oriented architecture can bind to different external services and communicate with the latter dynamically at run time. The testing of such a service should address adaptability characteristics. We call such a dynamical change in binding as a late-change. Many existing techniques on regression testing are unaware of such dynamic evolution of binding with respect to the service under test.

Preemptive Regression Testing (PRT) is a new approach proposed in this paper. It detects late-changes during the execution of a regression test suite, preempts the execution, selects test cases from a regression test suite as fixes, runs the fixes, and then resumes the suspended execution of the regression test suite. It repeats the process until no test execution preemption between any two test cases of the whole test suite occurs. To demonstrate PRT, we have formulated three strategies that detect the changes in workflow coverage achieved by the regression test suite over the modified web service. We have also reported an empirical study. The results have shown that our techniques are more efficient, and demonstrated a clear tradeoff between effectiveness and efficiency among the PRT strategies.

It will be interesting to explore PRT further by formulating other strategies with higher fault-detection effectiveness and lower slowdown overheads. It will also be interesting to investigate how to make PRT scalable and develop a framework for developing different PRT techniques.

Acknowledgment

This research is supported in part by the General Research Fund of the Research Grant Council of Hong Kong (project nos. 111410 and 717811), and a basic research funding of Beihang University (project no. YWF-12-LXGY-008).

References

[1] *alphaWorks Technology: BPEL Repository*. IBM, 2006. Available at http://www.ibm.com/developerworks/webservices/library/ws-awbpelrepos/.

[2] C. Bartolini, A. Bertolino, S. G. Elbaum, and E. Marchetti. Whitening SOA testing. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*ESEC 2009/FSE-17*), pages 161–170. ACM, New York, NY, 2009.

[3] C. Bartolini, A. Bertolino, E. Marchetti, and I. Parissis. Data flow-based validation of web services compositions: perspectives and examples. In *Architecting Dependable Systems V*, volume 5135 of Lecture Notes in Computer Science, pages 298–325. Springer, Berlin, Germany, 2008.

[4] W. K. Chan, S. C. Cheung, and K. R. P. H. Leung. Towards a metamorphic testing methodology for service-oriented software applications. In *The 1st International Conference on Services Engineering* (*SEIW 2005*), *Proceedings of the 5th International Conference on Quality Software* (*QSIC 2005*), IEEE Computer Society, Los Alamitos, CA, 470–476, 2005.

[5] L. Chen, Z. Wang, L. Xu, H. Lu, and B. Xu. Test case prioritization for web service regression testing. In *Proceedings of the 5th IEEE International Symposium on Service Oriented System Engineering* (*SOSE 2010*), pages 173–178. IEEE Computer Society, Los Alamitos, CA, 2010.

[6] S. G. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: a family of empirical studies. *IEEE Transactions on Software Engineering*, 28 (2): 159–182, 2002.

[7] X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *Proceedings of the 13th International Conference on World Wide Web* (*WWW 2004*), pages 621–630. ACM, New York, NY, 2004.

[8] S.-S. Hou, L. Zhang, T. Xie, and J.-S. Sun. Quota-constrained test-case prioritization for regression testing of service-centric systems. In *Proceedings of the IEEE International Conference on Software Maintenance* (*ICSM 2008*), pages 257–266. IEEE Computer Society, Los Alamitos, CA, 2008.

[9] J.-M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the 24th International Conference on Software Engineering* (*ICSE 2002*), pages 119–129. ACM, New York, NY, 2002.

[10] H. K. N. Leung and L. J. White. Insights into regression testing. In *Proceedings of the IEEE International Conference on Software Maintenance* (*ICSM 1989*), pages 60–69. IEEE Computer Society, Los Alamitos, CA, 1989.

[11] Z. Li, M. Harman, and R. M. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 33 (4): 225–237, 2007.

[12] Z. Li, W. Sun, Z. B. Jiang, and X. Zhang. BPEL4WS unit testing: framework and implementation. In *Proceedings of the IEEE International Conference on Web Services* (*ICWS 2005*), pages 103–110. IEEE Computer Society, Los Alamitos, CA, 2005.

[13] L. Mei. A context-aware orchestrating and choreographic test framework for service-oriented applications. In *Doctoral Symposium, Proceedings of the 31st International Conference on Software Engineering* (*ICSE 2009*), pages 371–374. IEEE Computer Society, Los Alamitos, CA, 2009.

[14] L. Mei, W. K. Chan, and T. H. Tse. Data flow testing of service-oriented workflow applications. In *Proceedings of the 30th International Conference on Software Engineering* (*ICSE 2008*), pages 371–380. ACM, New York, NY, 2008.

[15] L. Mei, W. K. Chan, and T. H. Tse. Data flow testing of service choreography. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*ESEC 2009/FSE-17*), pages 151–160. ACM, New York, NY, 2009.

[16] 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*, 84 (4): 603–619, 2011.

[17] L. Mei, Z. Zhang, W. K. Chan, and T. H. Tse. Test case prioritization for regression testing of service-oriented business applications. In *Proceedings of the 18th International Conference on World Wide Web* (*WWW 2009*), pages 901–910. ACM, New York, NY, 2009.

[18] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22 (8): 529–551, 1996.

[19] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27 (10): 929–948, 2001.

[20] M. E. Ruth and S. Tu. Towards automating regression test selection for web services. In *Proceedings of the 16th International Conference on World Wide Web* (*WWW 2007*), pages 1265–1266. ACM, New York, NY, 2007.

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

[22] *Web Services Choreography Description Language Version 1.0*. World Wide Web Consortium, Cambridge, MA, 2005. Available at http://www.w3.org/TR/ws-cdl-10.

[23] *Web Services Description Language* (*WSDL*) Version 2.0 Part 1: Core Language. World Wide Web Consortium, Cambridge, MA, 2007. Available at http://www.w3.org/TR/ wsdl20/.

[24] *XML Path Language* (*XPath*) 2.0: W3C Recommendation. World Wide Web Consortium, Cambridge, MA, 2007. Available at http://www.w3.org/TR/xpath20/.

[25] 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. In *Proceedings of the IEEE International Conference on Web Services* (*ICWS 2010*), pages 211–218. IEEE Computer Society, Los Alamitos, CA, 2010.

[26] H. Zhu and Y. Zhang. Collaborative testing of web services. *IEEE Transactions on Services Computing*, 5 (1): 116–130, 2012.