Postprint of article in the *Journal of Systems and Software* 84 (4): 603–619 (2011)

XML-manipulating test case prioritization for XML-manipulating services[*†]

Lijun Mei[a], W.K. Chan[b‡], T.H. Tse[a], Robert G. Merkel[c]

[a] *Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong*
[b] *Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Hong Kong*
[c] *Faculty of Information Technology, Monash University, Clayton, Victoria, Australia*

A B S T R A C T

A web service may evolve autonomously, making peer web services in the same service composition uncertain as to whether the evolved behaviors are compatible with its original collaborative agreement. Although peer services may wish to conduct regression testing to verify the agreed collaboration, the source code of the former service may be inaccessible to them. Owing to the black-box nature of peer services, traditional code-based approaches to regression testing are inapplicable. In addition, traditional techniques assume that a regression test suite for verifying a web service is available. The location to store a regression test suite is also a problem. On the other hand, we note that the rich interface specifications of a web service provide peer services with a means to formulate black-box testing strategies. In this paper, we provide a strategy for black-box service-oriented testing. We also formulate new test case prioritization strategies using tags embedded in XML messages to reorder regression test cases, and reveal how the test cases use the interface specifications of web services. We experimentally evaluate the effectiveness of these black-box strategies in revealing regression faults in modified WS-BPEL programs. The results show that the new techniques can have a high chance of outperforming random ordering. Moreover, our experiment shows that prioritizing test cases based on WSDL tag coverage can achieve a smaller variance than that based on the number of tags in XML messages in regression test cases, even though their overall fault detection rates are similar.

Keywords:
Test case prioritization; Black-box regression testing; WS-BPEL; Service testing; Service-oriented testing

1. Introduction

The testing and analysis of web services have posed new foundational and practical challenges, such as the non-observability problem (Canfora and Di Penta, 2006; Mei et al., 2009d), the extensive presence of non-executable artifacts within and among web services (Mei et al., 2008a, 2009b), safeguards against malicious messages from external parties (Xu et al., 2005; Martin et al., 2007), ultra-late binding (Bartolini et al., 2008), and cross-organizational issues (Ye et al., 2009). Researchers have proposed diverse techniques to address the test case selection problem (Martin et al., 2007), the test adequacy problem (Mei et al., 2008b, 2009b), the test oracle problem (Tsai et al., 2005a; Chan et al., 2007), and the test case prioritization problem (Hou et al., 2008; Mei et al., 2009c, 2009d).

Regression testing is the *de facto* activity to address the testing problems caused by software evolution (Onoma et al., 1998). It aims to detect software faults by retesting modified software versions. However, many existing regression testing techniques (such as Harrold et al., 1993; Kim and Porter, 2002; Mei et al., 2009c; Rothermel et al., 2001) assume that the source code is available for monitoring (Mei et al., 2009d), and use the coverage information of executable artifacts (such as statement coverage achieved by individual test cases) to conduct regression testing. Nonetheless, the coverage information on an external service may not be visible to the service composition that utilizes this service. Moreover, even

† This research is supported in part by the General Research Fund of the Research Grant Council of Hong Kong (project no. 717308), a strategic research grant of City University of Hong Kong (project no. 7002464), and a discovery grant of the Australian Research Council (project no. DP0984760). A preliminary version of this paper was presented in QSIC 2009 (Mei et al., 2009c).

‡ Corresponding author. Tel.: +852 3442 9684.

E-mail addresses: ljmei@cs.hku.hk (L. Mei), wkchan@cs.cityu.edu.hk (W.K. Chan), thtse@cs.hku.hk (T.H. Tse), robert.merkel@benambra.org (R.G. Merkel).

though a technique may insert probing services to collect and compute coverage (Mei et al., 2008b, Section 5.3; Bartolini et al., 2009), the effect depends on whether the service being sampled is willing to provide such information accurately. Since such code coverage information cannot be assumed to be available, it is vital to consider alternative sources of information to facilitate effective regression testing.

Many web services (or *services* for short) use the *Web Services Description Language* (*WSDL*) (W3C, 2007a) to specify their functional interfaces and message parameters. They also use XML documents to represent the messages. To quantify the transfer of type-safe XML messages with external partners, the WSDL documents of a service further embed the types of such messages (Mei et al., 2008b).

WSDL documents are rich in interface information. Moreover, such documents are often observable by external services. Despite the richness of such documents, to the best of our knowledge, the use of WSDL documents to guide regression testing without assuming code availability has not been proposed or evaluated in the literature.

In general, different services developed by the same or multiple development teams may be modified independently of one another, and the evolution of services may not be fully known by every other service. With respect to a service maintained by a development team, a service maintained by another development team can be regarded as a service collaborator or a service consumer of the former service.

Let us consider a scenario that a service *A* (as a service consumer) would like to pair up with a service *B*, and yet the latter service may evolve over time or contain faults that lead to failures in some executions of their service collaborations. The service consumer *A* may want to execute some tests on the functions provided by *B* to ensure that *A*'s service composition has not been adversely affected (at least from *A*'s perspective). For instance, a company may want to make use of the electronic payment gateway provided by a bank to conduct payment transactions with the bank. Under this scenario, the internal service of the company is the service consumer of the payment gateway service of the bank. To the benefit of the company, the development team of the internal service would like to test its service collaboration with the payment gateway service comprehensively. In terms of testing, it typically means that many test cases will be used, which is costly to execute.

Furthermore, the program code of *B* (such as the payment gateway service of the bank in the above scenario) is generally inaccessible to *A* (the internal service of the company in the above scenario). Therefore, even though *A* may be able to discover and invoke a test suite to conduct regression testing on *B*, the above scenario makes impossible the test execution schedule that applies existing code-based regression testing techniques (Leung and White, 1989; Harrold et al., 1993) in general, and test case prioritization techniques (Rothermel et al., 2001) in particular, to improve the fault detection rate and achieve other goals.

The WSDL documents of services are accessible among peer services. It is well known, however, that black-box testing is not adequate and must be supplemented by white-box testing (Chen et al., 1998). *How well does the richness of information embedded in typical WSDL documents help alleviate this deficiency in service-oriented testing? Is it effective to use the black-box information in WSDL documents to guide regression testing to overcome the difficulties in testing services with hidden implementation details such as source code?* These questions motivate the study presented in this paper.

We observe that, in a regression test suite for service testing, existing test cases may record the associated XML messages that have been accepted or returned by a (previous) version of the target service. Because the associated WSDL documents capture the target service's functions and types of XML message, the *tags* defined in such documents and encoded in individual test cases can be filtered through all the WSDL documents. Moreover, we observe that a WSDL tag may occur several times in the same or different XML messages within a test case. For instance, to collect room booking information, multiple instances of room information often appear in the XML messages.

Following up on these observations, we propose two aspects in formulating test case prioritization techniques. The first aspect to make use of the tags in XML messages in relation to the WSDL documents of the service under test. We propose the use of WSDL tag coverage statistics and WSDL tag occurrence statistics. Based on these two statistics, we can cluster the test cases and iteratively select them from the sequence of clusters. The second aspect is to define the order of the sequence of clusters. There are many ways to do so, including simple orderings such as randomization, sorting, as well as more advanced sampling strategies. To facilitate further comparison of future research, we choose a simple strategy (namely, sorting according to the count statistics) so that researchers can easily compare it with their own strategies in the context of service regression testing.

Following these directions, we formulate four prioritization techniques as proofs of the concepts. We further conduct an empirical study on a suite (from Mei et al., 2009d) of WS-BPEL applications (OASIS, 2007) using both adequate test suites and random test suites to verify the effectiveness of our techniques. The results show that our techniques can have high chances of outperforming random ordering. Moreover, our experiment shows that prioritizing test cases based on WSDL tag coverage by regression test cases can achieve a smaller variance than that based on the number of WSDL tag occurrences resulting from regression test cases, even though their overall fault detection rates are similar. Our experiment also shows that the fault detection rates of our techniques on the subject applications can be less effective than white-box techniques, but this finding is *not* statistically significant.

Techniques for the construction of effective regression test suites are not within the scope of this paper. We appreciate that invalid test cases can be costly because they still require the execution of the service under test despite the lack of fruitful results. We assume that all the test cases in a given regression test suite are valid. Invalid regression test cases can be removed, for instance, using the information in the fault handler messages returned by the service, or using the WSDL documents of the services to validate the format of the test cases in advance. Once a test case has been identified to be invalid, it can be permanently removed from the regression test suite for that service. Thus, during the next regression testing of the service (without knowing in advance whether the service has evolved), the test case does not need to be considered in test case prioritization.

Existing techniques, such as Rothermel et al. (2001) and Mei et al. (2009d), assume that information on regression test cases is already available. In service-oriented applications, however, since the coordination program may use both in-house and external services to implement the functionality, the test suite for evaluating a service composition needs to be determined dynamically before each round of testing. We will, therefore, discuss how to model the entire testing procedure in this paper.

The main contribution of this paper with its preliminary version (Mei et al., 2009c) is threefold: (i) We propose a new set of black-box techniques to prioritize test cases for regression testing of services having observable and rich content interface. It eases the problem of autonomous evolution of individual services in service compositions, so that peer services can gain confidence on the service under test with lower cost in regression testing. Our technique is particularly useful when the source code of the service under test is not available or is too costly to obtain. (ii) We address the challenges in performing black-box regression testing for service-oriented applications, and develop a strategy to facilitate such testing. (iii) We report the first controlled experimental evaluation of the effectiveness of black-box regression testing in the context of service testing. Our empirical results indicate that the use of the information captured in WSDL documents (paired with regression test suites) is a promising way to lower the cost of quality assurance of workflow services. Our empirical results also indicate that the different partitions generated according to the different perspectives (white-box coverage or black-box coverage information) have different effects on the fault detection rates for different kinds of faults.

The rest of the paper is organized as follows: Section 2 introduces the foundations of test case prioritization. Section 3 introduces the preliminaries of our approach through a running service scenario. Section 4 discusses the challenges in regression testing of services, and presents new test case prioritization techniques for regression testing of services. Section 5 presents an experiment design and its results to evaluate our proposal. Section 6 presents the discussions, Section 7 reviews the related work, and finally, Section 8 concludes the paper and discusses future work.

2. Preliminaries

This section introduces the terminology of test case prioritization.

Test case prioritization (Rothermel et al., 2001) is a kind of regression testing technique (Li et al., 2007). Through the use of information from previous rounds of software evaluation, we can design techniques to rerun the test cases to achieve certain goals (such as to increase the fault detection rate of a test suite). We adopt the test case permutation problem from Rothermel et al. (2001) as follows:

Given: A test suite T; the set of permutations PT of T; and a function f from PT to real numbers. (For instance, f may calculate the fault detection rate of a permutation of T.)

Problem: Find $T' \in PT$ such that $\forall T'' \in PT$, $f(T') \geq f(T'')$.

The *Average Percentage of Faults Detected* (*APFD*) (Elbaum et al., 2002) measures the weighted average of the percentage of faults detected over the life of a test suite. APFD has been widely used in regression testing research. As Elbaum et al. (2002) point out, a higher APFD value implies a better fault detection rate. Let T be a test suite containing n test cases, and F be a set of m faults revealed by T. Let TF_i be the index of the first test case in a permutation T' of T that reveals fault i. The APFD value for test suite T' is defined as follows:

$$APFD = 1 - \frac{TF_1 + TF_2 + ... + TF_m}{nm} + \frac{1}{2n}$$

From this formula, we observe that APFD treats all faults equally. We further adopt an example from Mei et al. (2009c) to show how APFD measures the fault detection rates of different test suite permutations, as illustrated in Figure 1. The left-hand table shows the faults that test cases t_A to t_E can detect. T_1 $\langle t_B, t_A, t_D, t_C, t_E\rangle$ and T_2 $\langle t_C, t_D, t_E, t_A, t_B\rangle$ are two permutations of t_A to t_E. The APFD measures for T_1 and T_2 are given in Figures 1(a) and (b), respectively.

3. A scenario of service interaction

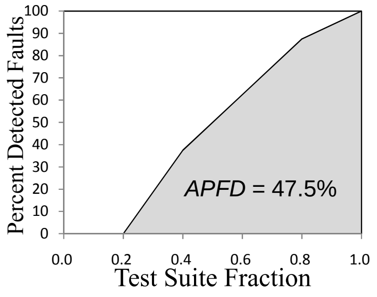
This section presents the scenario of a running service and introduces the preliminaries of our approach.

Let us consider a *HotelBooking* service adapted from the *TripHandling* project (IBM, 2006). Figure 2 depicts its control flow using an activity diagram in UML. A node represents an activity and a link represents a transition between two activities. We also annotate the nodes with

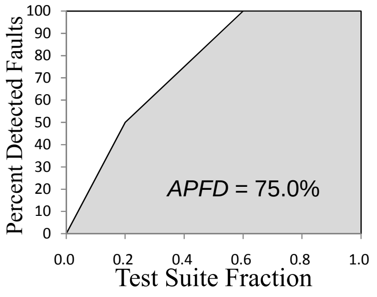
Test Case	Fault							
	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8
t_A	•		•					•
t_B								
t_C		•		•	•			•
t_D	•	•				•		
t_E			•				•	

Example on test suite and faults exposed

T_1: $\langle t_B, t_A, t_D, t_C, t_E\rangle$ T_2: $\langle t_C, t_D, t_E, t_A, t_B\rangle$



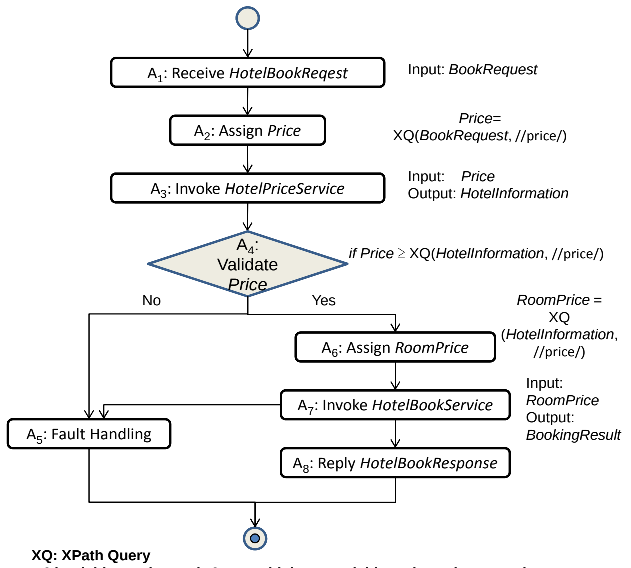
(a) *APFD* for test suite T_1



(b) *APFD* for test suite T_2

Figure 1. Example illustrating the measure (from Mei et al. (2009c, 2009d)).

information (such as an XPath) extracted from the program. We label the nodes as A_i for i = 1–8. The service aims to find a hotel room whose price is not more than a ceiling set by the user.



A1: Receive HotelBookReqest
Input: BookRequest
A2: Assign Price
Price= XQ(BookRequest, //price/)
A3: Invoke HotelPriceService
Input: Price
Output: HotelInformation
A4: Validate Price
if Price ≥ XQ(HotelInformation, //price/)
No
Yes
RoomPrice = XQ (HotelInformation, //price/)
A6: Assign RoomPrice
A7: Invoke HotelBookService
Input: RoomPrice
Output: BookingResult
A5: Fault Handling
A8: Reply HotelBookResponse

XQ: XPath Query
XQ(Variable, Exp): XPath Query with input variable and XPath expression.

Figure 2. Activity diagram for *HotelBooking*.

The service needs to handle the information in the reply messages from different hotels. An XML schema hotel for such messages is shown in Figure 3. A hotel has two types of room: doubleroom (line 3) and singleroom (line 4). Both doubleroom and singleroom are defined by the type room (lines 6–9). This XML schema is kept in a WSDL document. For a given service, there may be multiple WSDL documents (having relationships with one another) to define the set of service operations and message data types. For example, the schema of *hotel* and the schema of *room* may be defined in two different WSDL documents. For simplicity, however, we will simply show one XML schema in one WSDL document. Furthermore, we will use WSDL to refer to all the .wsdl and .xsd files (including local and remote files) that the master .wsdl of the web services points to.

```
<xsd:complexType name="hotel">
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="doubleroom" type="xsd:room"/>
    <xsd:element name="singleroom" type="xsd:room" />
</xsd:complexType>
<xsd:complexType name="room">
    <xsd:element name="roomno" type="xsd:int" />
    <xsd:element name="price" type="xsd:int"/>
</xsd:complexType>
```

Figure 3. XML schema hotel in WSDL document.

As we have highlighted in Section 1, in such cross-service scenarios, the internal model of a service (Figure 2) is unobservable by another service that wants to conduct regression (or conformance) test on the former service (to check whether the former service still conforms to the already established interoperability requirements with the latter service). Figure 4 depicts this problem.

Suppose the *TripHandling* service (the rightmost part of Figure 4) needs to invoke both the *HotelBooking* and *FlightBooking* services to handle the user's trip arrangement request. *TripHandling* may wish to run a regression test suite to assure the quality of *HotelBooking* inside its service composition. A test on such a service is usually done by sending request messages followed by receiving and handling response messages. Both the request and response messages are in XML format. These XML messages are visible to both *TripHandling* and *HotelBooking*, and the WSDL documents can be accessed publicly. However, the internal mechanism (see Figure 2) of *HotelBooking* service is hidden from *TripHandling*.

4. Test case prioritization

This section demonstrates how we adapt coverage-based test case prioritization strategy to formulate new prioritization techniques using WSDL information.

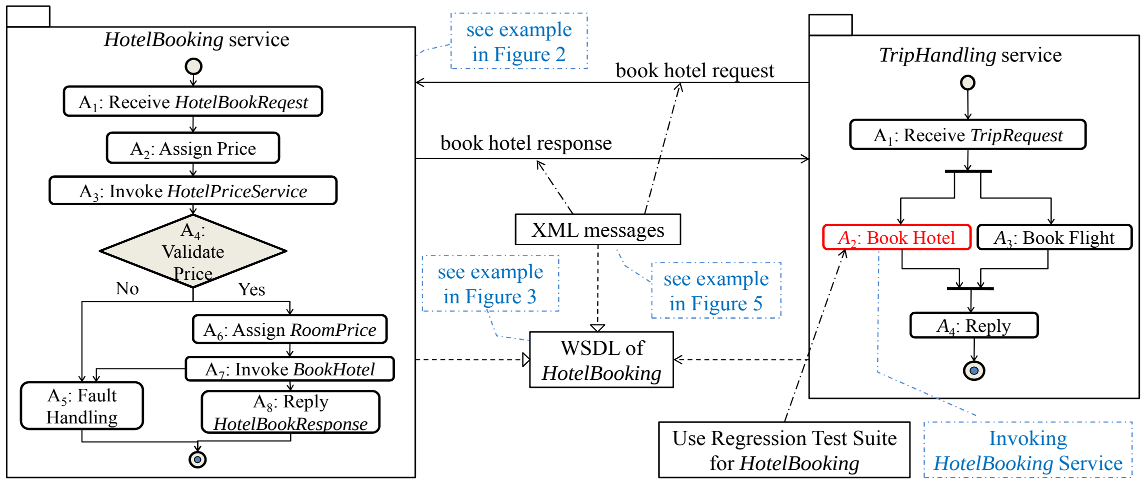


Figure 4. Service interaction scenario.

4.1. Motivating example

Let us consider seven test cases, in which the message has a price tag with the following values:

Test case 1 (t_1): *Price* = 200	Test case 2 (t_2): *Price* = 160
Test case 3 (t_3): *Price* = 150	Test case 4 (t_4): *Price* = 100
Test case 5 (t_5): *Price* = 50	Test case 6 (t_6): *Price* = 0
Test case 7 (t_7): *Price* = −100	

Figure 5 shows the XML messages for *HotelInformation* received in an XPath query (*HotelInformation*, //price/) by executing test cases t_1–t_7 on the *HotelBooking* service.

Our scenario further assumes that *HotelBookService* at A_7 fails in handling the fourth test case (t_4) because no available room satisfies this price (that is, the value kept by the message content *RoomPrice*). Therefore, the branch ⟨A_7, A_5⟩ in Figure 2 is covered by t_4. The activity coverage and workflow transition coverage of these test cases are summarized in Tables 1 and 2, respectively. We use a "•" to represent the item that has been covered by a test case.

We have the following observations from Tables 1 and 2: (i) The respective coverage scores of the activities by t_5–t_7 are identical; the same is true for the coverage of their transitions. (ii) The numbers of covered activities are identical for t_1–t_4; the same is true for their transitions. (iii) Suppose t_1 is first selected; using the "additional" coverage information (Rothermel et al., 2001), the remaining covered activities or transitions for t_4–t_7 are the same, while those for t_2 and t_3 are 0. In such a tie case, existing test case prioritization techniques such as

`<hotel> <name>Times</name> <doubleroom> <roomno>R101</roomno> <price>180</price> </doubleroom> <doubleroom> <roomno>R103</roomno> <price>150</price> </doubleroom> <singleroom> <roomno>R106</roomno> <price>120</Price> </singleroom> </hotel >`	`<hotel> <name>Times</name> <doubleroom> <roomno>R103</roomno> <price>150</price> </doubleroom> <singleroom> <roomno>R106</roomno> <price>120</Price> </singleroom> </hotel >`	`<hotel> <name>Times</name> <doubleroom></doubleroom> <singleroom> <roomno>R106</roomno> <price>120</Price> </singleroom> </hotel >`	`<hotel> <name>Times</name> <doubleroom></doubleroom> <singleroom> <roomno></roomno> <price>100</Price> </singleroom> </hotel >`
Test Case 1	Test Case 2	Test Case 3	Test Case 4
`<hotel> <name>Times</name> <doubleroom></doubleroom> <singleroom></singleroom> </hotel >`	`<hotel> </hotel >`	null	
Test Case 5	Test Case 6	Test Case 7	

Figure 5. XML messages for *HotelInformation* used in XQ(*HotelInformation*, //price/).

Rothermel et al. (2001) simply order them randomly to break the tie. As we have discussed in Section 1, such coverage data may not be observable by the service consumer in a service environment. Therefore, even though these statistics help prioritize test cases effectively, testers may find them impractical to obtain.

Table 1. Activity coverage by t_1–t_7.

Activity	t_1	t_2	t_3	t_4	t_5	t_6	t_7
A_1	•	•	•	•	•	•	•
A_2	•	•	•	•	•	•	•
A_3	•	•	•	•	•	•	•
A_4	•	•	•	•	•	•	•
A_5				•	•	•	•
A_6	•	•	•	•			
A_7	•	•	•	•			
A_8	•	•	•				
Total	7	7	7	7	5	5	5

Table 2. Transition coverage by t_1–t_7.

Transition	t_1	t_2	t_3	t_4	t_5	t_6	t_7
$\langle A_1, A_2\rangle$	•	•	•	•	•	•	•
$\langle A_2, A_3\rangle$	•	•	•	•	•	•	•
$\langle A_3, A_4\rangle$	•	•	•	•	•	•	•
$\langle A_4, A_5\rangle$					•	•	•
$\langle A_4, A_6\rangle$	•	•	•	•			
$\langle A_6, A_7\rangle$	•	•	•	•	•	•	•
$\langle A_7, A_8\rangle$	•	•	•				
$\langle A_7, A_5\rangle$				•			
Total	6	6	6	6	5	5	5

Every XML message resulting from a test case is of the form "<*tag*> ... </*tag*>", where <*tag*> is a WSDL tag. If a regression test case contains an XML message with a WSDL tag <*tag*>, we say that <*tag*> has been covered by the test case. Table 3 summarizes the coverage of the WSDL tags by test cases t_1–t_7 for the service interaction scenario. The WSDL tag coverage scores by t_5–t_6 are different. Similarly, the coverage by t_1 and t_2 are different from that of t_3 and t_4. If t_1 is first selected, since there is a reset procedure for coverage information, the remaining covered WSDL tags for t_3–t_7 are still different (Rothermel et al., 2001). These observations motivate us to study the use of WSDL tag coverage in prioritizing test cases.

However, t_1 and t_2 report the same coverage from Tables 1–3. Since there can be duplicated WSDL tags in an XML message (such as the two <doubleroom> tags in the reply message for test case 1 in Figure 5), we further count the number of occurrences of WSDL tags in Table 4. Using the number of WSDL tag occurrences, we can further differentiate between t_1 and t_2.

Table 3. WSDL tag coverage by t_1–t_7.

Element	t_1	t_2	t_3	t_4	t_5	t_6	t_7
hotel	•	•	•	•	•	•	
name	•	•	•	•	•		
doubleroom	•	•	•	•	•		
singleroom	•	•	•	•	•		
room (doubleroom)	•	•	•	•	•		
roomno (doubleroom)	•	•					
price (doubleroom)	•	•					
room (singleroom)	•	•	•	•	•		
roomno (singleroom)	•	•	•	•			
price (singleroom)	•	•	•	•			
Total	10	10	8	8	6	1	0

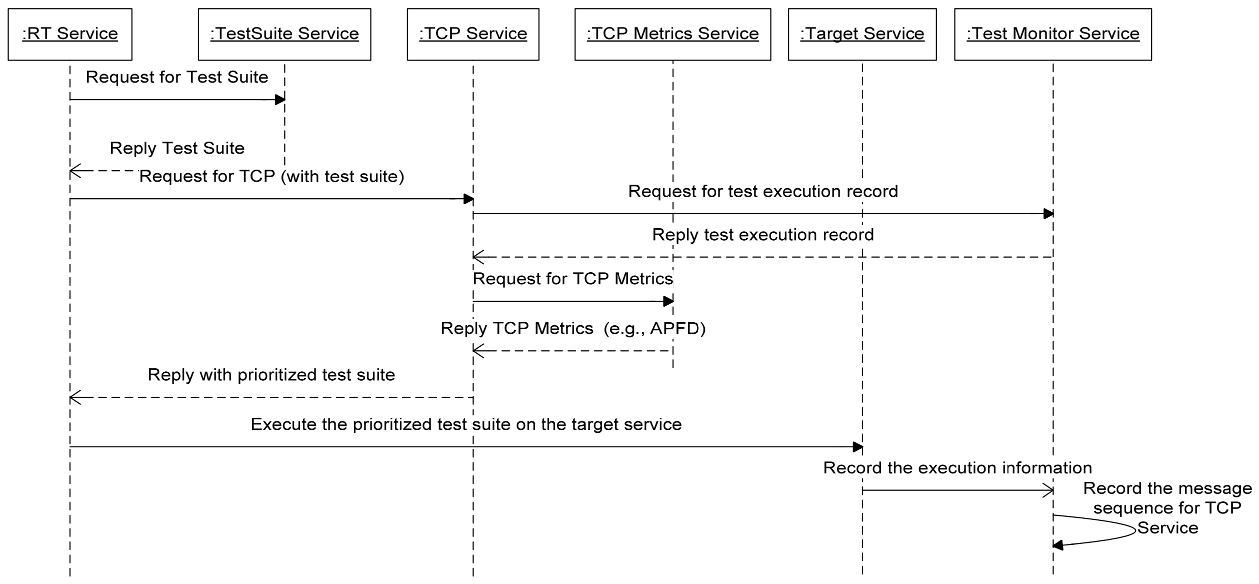

:RT Service
:TestSuite Service
:TCP Service
:TCP Metrics Service
:Target Service
:Test Monitor Service
Request for Test Suite
Reply Test Suite
Request for TCP (with test suite)
Request for test execution record
Reply test execution record
Request for TCP Metrics
Reply TCP Metrics (e.g., APFD)
Reply with prioritized test suite
Execute the prioritized test suite on the target service
Record the execution information
Record the message sequence for TCP Service

Figure 6. UML sequence diagram for black-box service-oriented testing.

As we have described in Section 1, peer services in a service composition may not have access to the source code of a target service. To know whether the target service still exhibits the desirable functions previously demonstrated in a service composition, peer services may run regression test suites on the target service. We explore the coverage of WSDL documents in our test case prioritization techniques with the intent to detect failures of the target services faster than random ordering. Moreover, in cases where white-box coverage information cannot be visible, our techniques can be used as replacements; otherwise, only random ordering can be adopted in such situations.

Table 4. WSDL tag occurrences resulting from t_1–t_7.

Element	t_1	t_2	t_3	t_4	t_5	t_6	t_7
hotel	**1**	**1**	**1**	**1**	**1**	**1**	0
name	**1**	**1**	**1**	**1**	**1**	0	0
doubleroom	**2**	**1**	**1**	**1**	**1**	0	0
singleroom	**1**	**1**	**1**	**1**	**1**	0	0
room (doubleroom)	**2**	**1**	**1**	**1**	**1**	0	0
roomno (doubleroom)	**2**	**1**	0	0	0	0	0
price (doubleroom)	**2**	**1**	0	0	0	0	0
room (singleroom)	**1**	**1**	**1**	**1**	**1**	0	0
roomno (singleroom)	**1**	**1**	**1**	**1**	0	0	0
price (singleroom)	**1**	**1**	**1**	**1**	0	0	0
Total	14	10	8	8	6	1	0

4.2. Black-box service-oriented testing

This section presents our black-box service-oriented testing strategy.

When considering testing scenarios of individual services, some test suites can be provided by third parties such as service consumers, or from public registers that record the invocations of services. On the other hand, test suites may not be provided in full to service consumers For example, some test cases may require a reset operation of the service contexts, and some may incur a charge. The utilization of test cases and test suites should therefore be more flexible and adaptive to facilitate the testing procedure.

The advantage of defining each major step of regression testing as a service is that such services can be dynamically changed and, furthermore, each service may be provided by a different service provider. The service representation of these steps provides service-oriented testers with a more flexible way to configure the regression testing procedure. For example, since the test suite for evaluating a service may be dynamically changed, we need to collect the latest version of a test suite from time to time, such as before starting a new round of regression test. Provision in the form of a service is therefore useful.

We first present formal definitions of the *roles* in black-box service-oriented testing.

Definition 1 (Test Case Service and Test Suite Service). A *test case service tcs* is a triple ⟨*url*, *t*, *s*⟩, where *url* is the location of the WSDL document of *tcs*, and *t* is the test case that can be used to evaluate service *s*. A test suite service *tss* is a triple ⟨*url*, *TS*, *s*⟩, where *url* is the location of *tss*, *TS* is a collection of test case services, each of which can evaluate service *s*.

Suppose we have a service *s'* that holds the test suites of another service *s*. We consider each test suite *T* in *s'* as a service. Therefore, *s'* can discover *T* by asking *s* about its test suite service by, for instance, finding the test suite service location in the interface specification of *s*.

Definition 2 (*TCP* Service and *TCP* Metrics Service). A *test case prioritization* (or *TCP*) *service tcps* is a triple ⟨*url*, *tcp*, *tcpms*⟩, where *url* is the location of the WSDL document of *tcps*, *tcp* is the test case prioritization technique used, and *tcpms* is the metrics service supporting *tcp*.

Definition 3 (Target Service). A *target service ts i*s a tuple ⟨*s*, *url*⟩, where *s* is the service that needs to be tested and *url* is the location of the WSDL document of *s*.

Definition 4 (Test Monitoring Service). A *test monitoring service tms* is a triple ⟨*url*, *s*, Ψ⟩, where *url* is the location of the WSDL document of *tms*, *s* is the service monitored by *tms*, and Ψ records the execution information of *s*, such as the sequence of exchange messages.

Definition 5 (RT Service). A *regression testing* (or *RT*) *service rts* is a tuple ⟨*url*, *ts*, *tss*, *tcps*, *tcpms*⟩, where *url* is the location of the WSDL document of *rts*, *ts* is the target service to be evaluated by using *rts*, *tss* is the test suite service that can be used to request for the test suite to evaluate *ts*, *tcps* is the test case prioritization service used to reorder test cases in *tss*, and *tcpms* is the prioritization metrics used by *tcps*.

Based on the above definitions of participating roles, we further propose the strategy of black-box service-oriented testing through the UML sequence diagram in Figure 6. We briefly explain the invocation sequence of this diagram. Regression Testing (RT) Service first collects a test suite from Test Suite Service, and then invokes Test Case Prioritization (TCP) Service to reorder this test suite. TCP Service further invokes TCP Metrics Service when calculating the weights of individual test cases in the test suite. TCP Service also needs to retrieve execution information on the test suite. After that, RT Service executes the reordered test suite on Target Service and, at the same time, the execution information (such as the message sequence) is recorded by Test Monitor Service.

In general, conducting a test on a service may involve multiple test monitors, multiple test cases, multiple TCP services, and multiple metric services. Ideally, the combination of such services can be changed dynamically and adaptively. It poses a workload demand on the underlying infrastructure to provide necessary computation and storage resources as well as the necessary bandwidth for different services to communicate efficiently. However, the variation of different combinations may pose very different workload demands. Putting these services in a cloud computing environment (Mei et al., 2008c) allows the target test (which is conceptually a service composition) to use the resources for support without bearing the cost of the underlying infrastructure when some resources are not used. For instance, if a particular test requires only one test monitor and demands to execute merely one percent of all applicable test cases, the provider of the service composition does not need to pay the charge for accommodating all test monitors and need not pay for the provision to keep or initialize the other 99 percent of applicable test cases. In the rest of the paper, we will present how to conduct test case prioritization for black-box services.

4.3. *Our prioritization techniques*

In this section, we first briefly review representative test case prioritization techniques for regression testing, and then introduce our adaptation. Among them, M1–M6 are adopted from Rothermel et al. (2001).

To study the effectiveness and tradeoffs of different techniques, we follow the styles of Elbaum et al. (2002) and Rothermel et al. (2001) in comparing other control techniques with ours. All the techniques and examples are listed in Table 5. Techniques M1–M6, representing *random*, *optimal*, *total-statement*, *additional-statement*, *total- branch*, and *additional-branch*, respectively, are taken from Rothermel et al. (2001).

Many WSDL documents define XML schemas used by services. Each XML schema contains a set of elements. Intuitively, the coverage information on these WSDL tags reveals the usage of the internal messages among activities.

Table 5. Prioritization techniques and examples.

Category	Name	Index	t_1	t_2	t_3	t_4	t_5	t_6	t_7
Benchmark	Random	M1	6	7	3	4	5	1	2
	Optimal	M2	–	–	–	–	–	–	–
Traditional White-Box	Total-Activity	M3	1	3	4	2	5	7	6
	Additional-Activity	M4	1	3	5	2	4	6	7
	Total-Transition	M5	2	3	4	1	5	6	7
	Additional-Transition	M6	1	6	4	2	3	7	5
Our Black-Box	Ascending-WSDL-TagCover	M7	5	7	6	4	3	2	1
	Descending-WSDL-TagCover	M8	1	2	3	7	4	5	6
	Ascending-WSDL-TagCount	M9	7	6	5	4	3	2	1
	Descending-WSDL-TagCount	M10	1	2	3	7	4	5	6

The coverage of all tags in a WSDL document may be easily satisfied (as illustrated, for instance, by t_1 or t_2 in Table 3). If a technique schedules test cases in the order of tag coverage of the WSDL document, then t_1 or t_2 will be selected first. Therefore, quantifying the tag coverage of WSDL documents provides a new way to explore the partitioning of test cases for service-oriented regression testing.

Intuitively, test cases with different WSDL tag coverage scores may indicate different types of messages (noting that although we have recorded the sequence of messages for a test case, the WSDL tags do not differentiate such sequences). We observe that a reply to a normal service invocation (which may include, say, a user profile indicating name, age, and other user information in XML format with a large number of WSDL tags) may provide more message contents than a reply to an abnormal service invocation (which often contains failure information only). WSDL tag coverage provides a feasible way to quantify the messages. The ordering generated according to such quantification may help achieve the following goal:

If testers think that the failures are triggered by normal service invocations, they can generate orderings in ascending number of WSDL tags in test cases. On the other hand, if testers think that the failures are more likely to be triggered by abnormal service invocations, they can generate orderings in descending number of WSDL tags in test cases. Based on the above analysis, we propose two techniques, namely M7 and M8.

M7: Ascending WSDL tag coverage prioritization (*Ascending-WSDL-TagCover*). This technique first partitions the test suite into groups where test cases in the same group have the same WSDL tag coverage, then sorts the groups in ascending order of the number of tags covered by a test case, and finally selects test cases iteratively from the ordered sequence of groups. (For each iteration, M7 selects one test case randomly from each group.)

M8: Descending WSDL tag coverage prioritization (*Descending-WSDL-TagCover*). This technique is the same as *Ascending-WSDL-TagCover*, except that it sorts the groups in descending order (instead of ascending order) of the WSDL tag coverage of each test case.

M7 and M8 examine the effect of WSDL tag coverage by a test case during test case prioritization for services. Both techniques include a grouping phase that partitions the test suite into groups. During the grouping phase, we count multiple occurrences of the same WSDL tag only once. In this way, two test cases in the same group have the same WSDL tag coverage (while their actual number of WSDL tags may be different). We then iteratively select test cases from each group. For example, possible prioritization orders for WSDL tag coverage by t_1–t_7 according to M7 and M8 may be $\langle t_7, t_6, t_5, t_3, t_4, t_2, t_1\rangle$ and $\langle t_2, t_1, t_3, t_4, t_5, t_6, t_7\rangle$, respectively.

In the context of black-box testing, the internal structure of a service-oriented program is not known. Thus, it is impossible to know the relative importance of different tags from the perspective of the technique. Without further information, M7 and M8 consider that the coverage of one tag is as important as the coverage of another. Such an assumption also applies to M9 and M10. Of course, in case there is any knowledge on the internal structure of the service-oriented program, we may differentiate among various tags according to their usages by the service-oriented program, and design appropriate techniques. However, this will be beyond the scope of black-box testing.

Next, we propose two techniques (M9 and M10) to prioritize test cases according to the number of occurrences of WSDL tags in each test case. The basic motivation of M9 and M10 is the same as that of M7 and M8, and hence we do not repeat the description here. However, M9 and M10 count the number of WSDL tag occurrences resulting from each test case, rather than the WSDL tag coverage by each test case. This handling can differentiate tie cases where multiple test cases have the same WSDL tag coverage but different WSDL tag occurrences.

M9: Ascending WSDL tag occurrence prioritization (*Ascending-WSDL-TagCount*). This technique first partitions a test suite into groups where test cases in the same group cover the same occurrences of WSDL tags, then sorts the groups in ascending order of the occurrence of tags covered by a test case, and finally selects test cases iteratively from groups. In each iteration, M9 selects one test case randomly from each group.

M10: Descending WSDL tag coverage prioritization (*Descending-WSDL-TagCount*). This technique is the same as *Ascending-WSDL-TagCount*, except that it sorts the groups in the descending order (instead of ascending order) of the occurrence of tags covered by a test case.

For instance, based on M9 and M10, possible prioritization orders for WSDL tag occurrence resulting from t_1–t_7 are ⟨t_7, t_6, t_5, t_4, t_3, t_2, t_1⟩ and ⟨t_1, t_2, t_4, t_3, t_5, t_6, t_7⟩, respectively.

Figure 7 further summarizes the difference between black-box testing techniques (M7–M10) and conventional (white-box) techniques (M3–M6). The figure shows that our black-box testing techniques only require interactive messages and the corresponding WSDL documents (as demonstrated in Section 3). In contrast, white-box testing techniques require the source programs of the services under test. Our techniques can be applied to services that may evolve over time. We assume that any given service provides a public test suite for consumers to verify its functionality and coordination. In case this is not provided, however, we may randomly create a test suite according to its public WSDL documentations (such as those stored in UDDI registries).

Moreover, to apply the techniques, we can reconstruct the document model (such as W3C, 2007a) based on the set of XML data. Our techniques can therefore be applied even when no WSDL is available for some application scenarios.

A valid test input with a new test result requires a test oracle to determine its correctness. We further assume that there is a test oracle. For instance, the developers or users of peer services may judge whether the returned test results are useful or meaningful.

Compared to our technique, a traditional function coverage prioritization strategy does not consider parametrical values or their dynamic types; nor does it determine the coverage achieved by individual test cases based on "tags". For instance, when polymorphic objects are used as parameters of a function, traditional techniques simply ignore this information, whereas such information has been considered in our techniques when covering

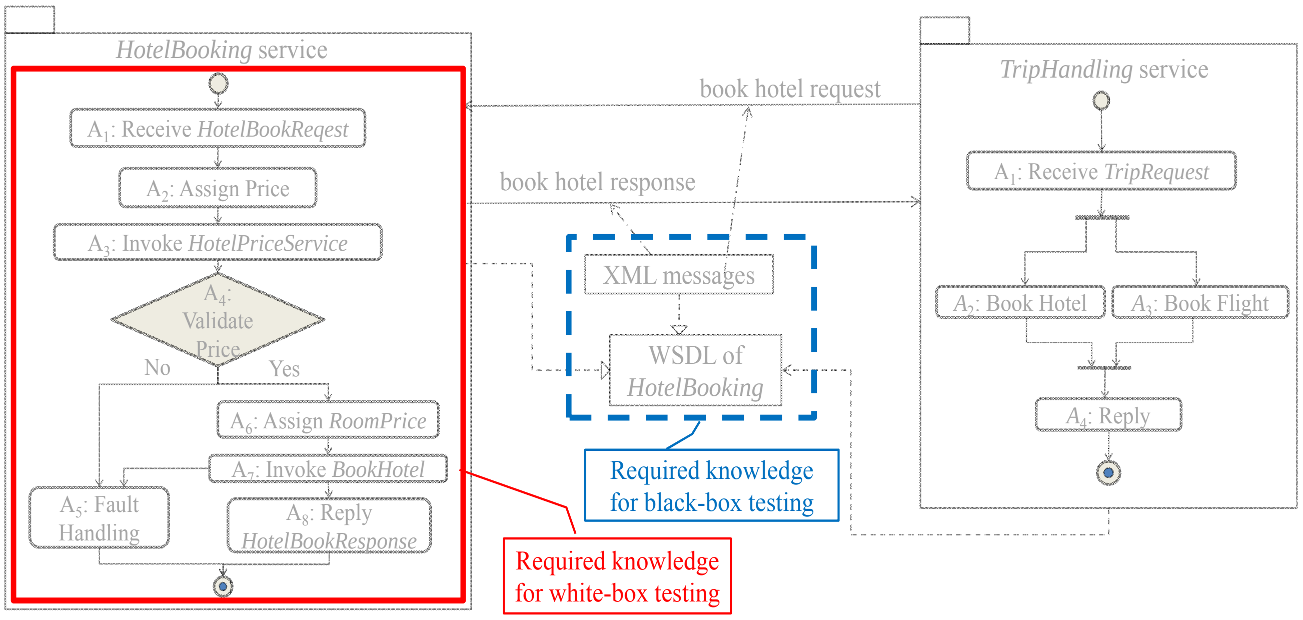


Figure 7. Different required knowledge for black- and white-box testing.

different tags of the XML schema captured in WSDL documents. Furthermore, while the collection of such information in traditional programs is costly, we observe that this information is available in services as a byproduct of their message interchange.

5. Experiment

This section reports on an experimental verification of our proposal.

5.1. *Experimental design*

5.1.1. *Subject programs, versions, and test suites*

We use a set of WS-BPEL applications to evaluate our techniques. The same set of artifacts has been used in the experiments in Mei et al. (2008b, 2009d). These applications have been used as benchmarks or examples in many WS-BPEL studies. Among them, the applications *atm*, *gymlocker*, *loanapproval*, *marketplace*, *purchase*, and *triphandling* are from BPEL repository (IBM, 2006), the application *buybook* is from Oracle BPEL Process Manager (Oracle Technology Network), and the application *dslservice* is from Web Services Invocation Framework (Apache Software Foundation, 2006).

Like many other studies that evaluate testing techniques, we seed known faults to measure the effectiveness of the prioritization techniques. Thus, we create a set of mutants for each benchmark program. Each mutant is a modified version of the original program with one fault seeded. A mutant is considered as an evolving version of an autonomous service. The fault-seeding procedure is similar to that of Hutchins et al. (1994). It has long been recognized (DeMillo et al., 1978) and verified experimentally (Offutt, 1992) that test cases that kill single-fault mutants are "very successful" in killing multiple-fault mutants as well. Jia and Harman (2009) point out the existence of "rare" combinations of faults that may mask one another and hence difficult to detect. However, test cases to kill such combinations are still under investigation by the said authors and hence we will not study them in the present paper.

Easily exposed faults are more likely to be detected and removed during program testing by developers, rather than allowed to persist until regression testing. Instead, we focus on relatively hard-to-detect faults when comparing the fault detection capabilities of various techniques. Hence, following Elbaum et al. (2000, 2002) and Jiang et al. (2009), we discard any faulty version if more than 20 percent of all test cases can detect failures due to the seeded fault. As such, 43 mutants are selected from the 60 mutants that are originally seeded. For any test experiment, one would always prefer a larger experiment that involves more subject programs and more faults. Some readers may therefore consider that the set of faults used in this experiment is not large enough. In our experience, however, simply executing a test case on one WS-BPEL program is already very tedious. Moreover, to the best of our knowledge, the scale of the experiment, both in terms of the number of subject programs and the number of faults, is already the largest among the experiments in published articles in service-oriented testing.

Table 6 shows the subject programs and their descriptive statistics. The descriptive statistics of the applicable modified versions are shown in the rightmost column of the table.

We constructed test cases randomly for each subject application. One thousand (1000) test cases were generated to form a test pool for each application. From each generated test pool, we randomly selected test cases to form a test suite. Selection continued iteratively until all the workflow activities, all the workflow transitions and all types of XML message had been covered by at least one test case. The procedure is similar to the test suite construction in Elbaum et al. (2002) and Mei et al. (2009c). We then applied the test suite to all the applicable faulty versions of the corresponding application. We successfully generated 100 test suites for every application. Table 7 shows the maximum, average, and minimum sizes of the test suites (where the size of a test suite refers to the number of test cases it contains).

Table 6. Subject programs and statistics.

Ref.	Application	Version	Element	LOC	WSDL Spec.	WSDL Tag	No. of Versions Used
A	Atm	8	94	180	3	12	5
B	Buybook	7	153	532	3	14	5
C	Dslservice	8	50	123	3	20	5
D	Gymlocker	7	23	52	1	8	5
E	Loanapproval	8	41	102	2	12	7
F	Marketplace	6	31	68	2	10	4
G	Purchase	7	41	125	2	10	4
H	Triphandling	9	94	170	4	20	8
	Total	60	527	1352	20	106	43

Table 7. Statistics of test suite sizes.

Ref. / Size	A	B	C	D	E	F	G	H	Avg.
Maximum	146	93	128	151	197	189	113	108	140.5
Average	95	43	56	80	155	103	82	80	86.3
Minimum	29	12	16	19	50	30	19	27	25.2

For every subject program and for every test suite thus constructed, we used each of the test case prioritization techniques (M1−M10) presented in Section 4 to prioritize the test cases. For every faulty version of the subject program and every corresponding prioritized test suite, we executed the test cases one by one according to their order in the test suite, and collected the test results.

5.1.2. *Effectiveness measure*

To compare the effectiveness (in terms of fault detection rates) among M1−M10, we use the Average Percentage of Faults Detected (APFD) as the metric, which

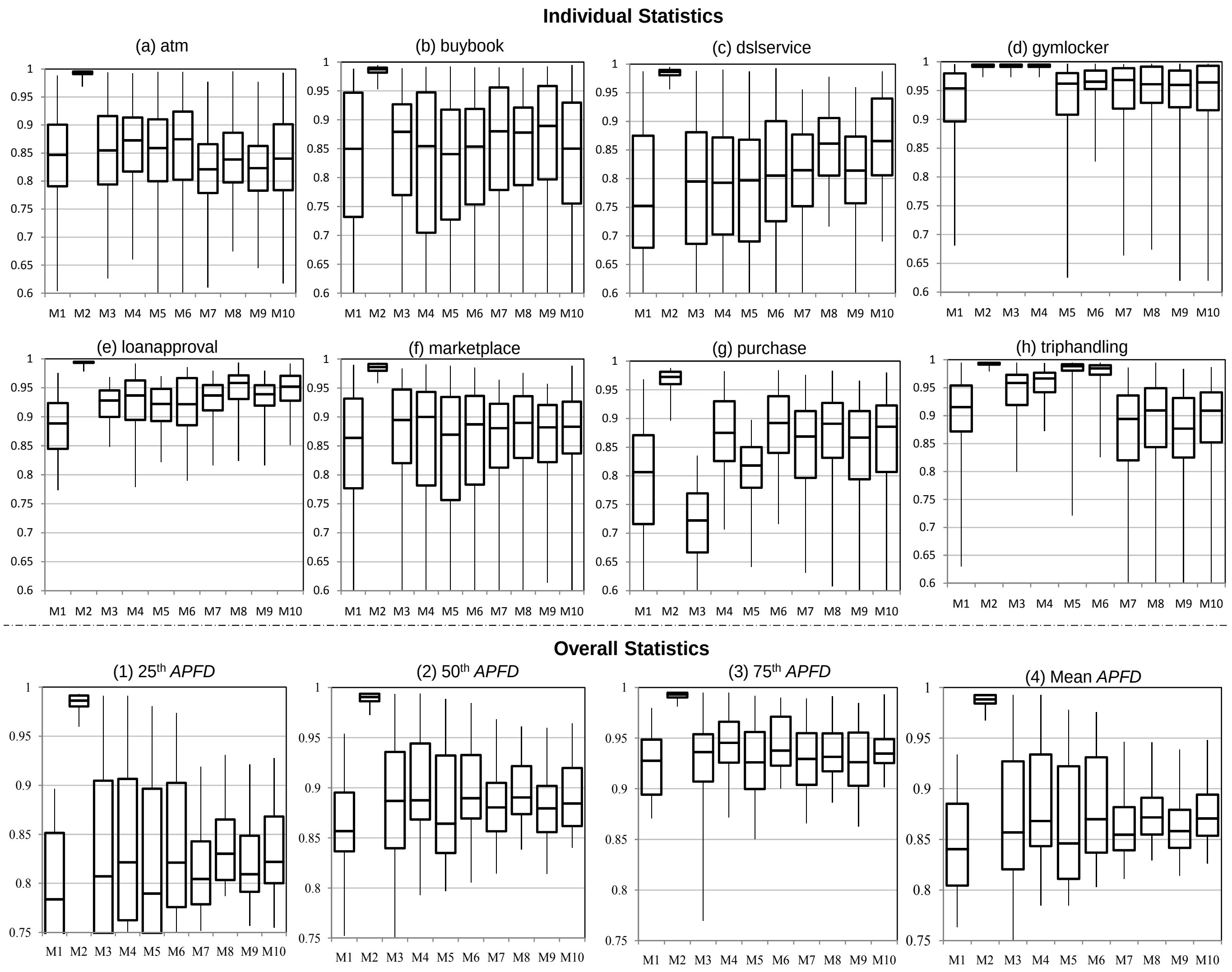


Figure 8. Comparisons of APFD measures among M1–M10 using coverage-adequate test suites.

(The *x*-axes show the techniques and the *y*-axes show the APFD values.)

measures the weighted average of the percentage of faults detected over the life of a test suite. The APFD metric has been introduced in Section 2.

5.2. Data analysis

5.2.1. Overall effectiveness

In this section, we analyze the data to evaluate the effectiveness of different techniques. We apply techniques M1–M10 on each application and calculate the APFD values. We repeat this procedure 100 times using the generated test suites. The results are collected and summarized in the box plots in Figure 8.

These box plots are drawn using the PTS box-and-whisker plot chart utility available through Microsoft Excel. Each box plot shows the 25th, 50th, and 75th percentiles of a technique. The result for each application is given in Figures 8(a)–(h). The overall 25th, 50th, and 75th percentiles as well as the overall mean of all subject programs are shown in Figures 8(1)–(4).

We observe from Figures 8(1)–(4) that M1 and M2 show the worst and the best performances, respectively, in terms of APFD values. This result is consistent with previous studies such as Rothermel et al. (2001).

We first use the 25th percentile APFD to compare M1–M10. Figure 8(1) shows that random prioritization (M1) achieves a mean value of 0.788. The minimum and maximum mean APFD achieved for the white-box techniques (M3–M6) are 0.817 and 0.839, respectively. On the other hand, the mean APFD for M8 is 0.844, which is higher than the corresponding values for M3–M6.

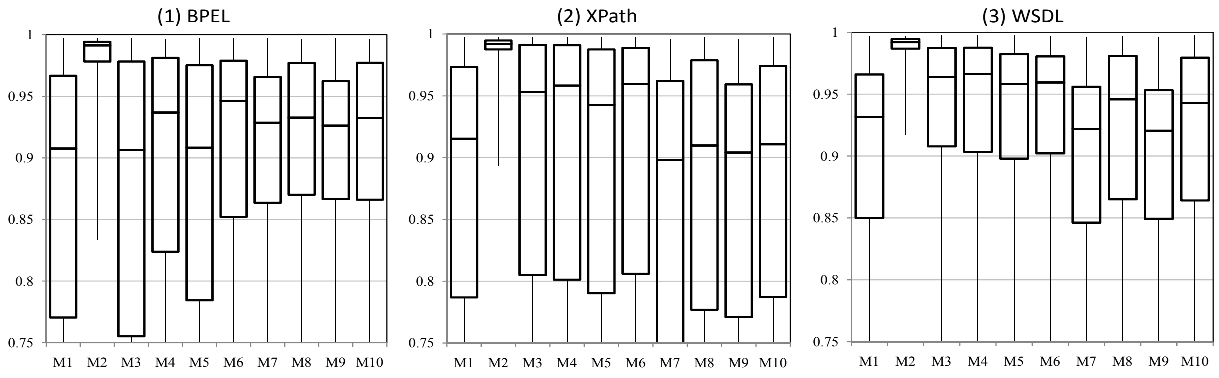


Figure 9. Categorized comparisons of APFD measures among M1–M10.

(The *x*-axes show the techniques and the *y*-axes show the APFD values.)

M7–M10 demonstrate much smaller variances than M1 and M3–M6.

Then, we compare M1–M10 using the median (that is, the 50th percentile) APFD value in Figure 8(2). M4 shows better results than all the other techniques from all perspectives except for the minimum APFD value.

Next, we compare M1–M10 using the 75th percentile APFD value in Figure 8(3). The variance of M10 is much smaller than that of any other technique. M3 and M5 have similar APFD values (which we call *performance* for ease of reference). The performance of M9 is similar to that of M7. M10 is a bit better than M8. This observation is not too surprising because of the lack of knowledge of the detailed program code for M7–M10.

Finally, we compare the mean APFD values. The mean APFD for M8 is 0.879, which is higher than that for M3 or M5 but lower than that for M4 or M6. This observation indicates that our black-box testing techniques (particularly M8) can achieve similar (or even better) APFD results when compared with the white-box testing techniques (M3–M6). The results of M9 and M10 are similar to those of M7 and M8, respectively.

Figure 8(4) shows that M8 and M10 are better than all the other techniques (except the optimal) at the 25th percentile of mean APFD results. Moreover, M3–M6 are also better than M1 at the 75th percentile. On average, as reported in Figures 8(1)–(4), our black-box testing techniques M7–M10 are close to white-box testing techniques (M3–M6) for the mean APFD values and at the 25th and 50th percentiles of APFD results. However, when considering the 75th percentile values in Figures 8(1)–(4), we observe that the performance of M7–M10 is worse than that of M3–M6.

Overall, M7–M10 are more effective at early fault detection than random ordering. This indicates that black-box test case prioritization techniques (instead of M1) are a promising method to use for service testing. As reported in Figures 8(1)–(4), the overall performance between M8 and M10 and that between M7 and M9 are similar. But we observe that, in some cases such as Figures 8(b) and (f), M8 is better than M10 while, in some other cases such as Figure 8(c), M10 is better than M8.

We further compare the overall performances of each technique in terms of the minimum, mean, and maximum APFD values in Table 8. We observe that the mean APFD values of M3–M10 are close to one another, and all are higher than that of M1. When considering the minimum APFD values, we find that M7–M10 all achieve higher values, which indicate that our techniques have smaller variances than M3–M6. An interesting observation is that M6 reports the same minimum APFD value as M1. This shows that sometimes the use of M6 may not contribute to an increase of the fault detection rate.

We also briefly compare M1–M10 from individual benchmark applications. We find that our technique M8 can even be better than M3–M6 in some benchmark applications (such as Figures 8(c) and (e)). Among all these subject applications, M7 and M8 are significantly better than M1 in five cases out of eight (Figures 8(b), (c), and (e)–(g)), close to M1 in two (Figures 8(d) and (h)), and only a little worse than M1 in one case (Figures 8(a)).

Table 8. Further comparisons of APFD measures among M1–M10.

Tech. / APFD	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10
Max.	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Mean.	0.84	0.99	0.86	0.88	0.87	0.88	0.87	0.88	0.87	0.88
Min.	0.04	0.90	0.28	0.25	0.30	0.04	0.31	0.38	0.43	0.44

We further observe that the results of M7 and M9 are similar, and those of M8 and M10 are also similar. Figure 8(3) shows that M10 has a much smaller variance than M1–M9 in all (in terms of the 75th percentile APFD values). This result indicates that M7–M10 have a high chance in outperforming random ordering. Thus, our techniques can be more effective than random ordering for black-box service-oriented testing.

5.2.2. *Effectiveness on categorized faults*

In this section, we analyze the capability of detecting different kinds of faults. We categorize faults into three categories, namely, BPEL faults, XPath faults, and WSDL faults, and collect APFD results for each category.

The results, as shown in Figure 9, are interesting. Before conducting the experiment, we expected that our black-box testing techniques might be better than white-box techniques in detecting WSDL faults. The actual results show, however, that our techniques have variances smaller than white-box techniques in detecting BPEL faults, but have variances larger than white-box techniques in detecting XPath and WSDL faults. Through further investigation, we find that some BPEL faults cannot be revealed using one kind of XML message input (which apply to a large proportion of a test suite) but can easily be revealed by another kind of XML message (which may apply to a small proportion of a test suite). In any case, our techniques can detect BPEL faults earlier (in terms of the 25th percentile APFD values) than M1, M4, and M5.

Next, let us analyze the performance in detecting XPath faults. Since an XPath fault may lead to a change in activity and transition coverage, it is not surprising that white-box techniques can achieve better results.

Finally, when comparing Figure 9(3) with Figures 9(1) and (2), we find that WSDL faults are easier to detect than BPEL and XPath faults. Figure 9(3) also indicates that the performance of test cases using M7–M10 may not as good as the use of M3–M6, and therefore the chance of achieving a higher fault detection rate is lower than M3–M6.

In summary, we observe the following through the analysis of the categorized fault detection rates: (i) proper partitioning of test cases can affect the fault detection rate, and (ii) when attempting to detect faults in an artifact of a program, if there are multiple kinds of artifacts in the program, it may be more effective to use other kinds of artifacts to partition test cases.

We also observe from Figures 8 and 9 that, overall speaking, the two (ascending and descending) sorting strategies provide observable differences in terms of medium APFD, and that the use of the descending strategy is better than using the ascending strategy.

5.2.3. *Hypothesis testing*

One may wonder whether the differences between our WSDL-related techniques and conventional techniques are significant. To answer this question, we conduct hypothesis testing to study the differences among the above techniques. We follow Li et al. (2007) and perform *one-way ANalysis Of VAriance* (*ANOVA*) to find out whether the means of APFD distributions for different techniques differ significantly. Since M2 (the optimal technique) is much better than the other techniques (see Figure 8), we skip the comparison with M2 in the hypothesis testing.

The null hypothesis is that the means of APFD values for M1 and M3–M10 are equal. To decide whether to accept or reject the null hypothesis, we set the significance level to 5%. If the p-value is smaller than 5%, the differ-

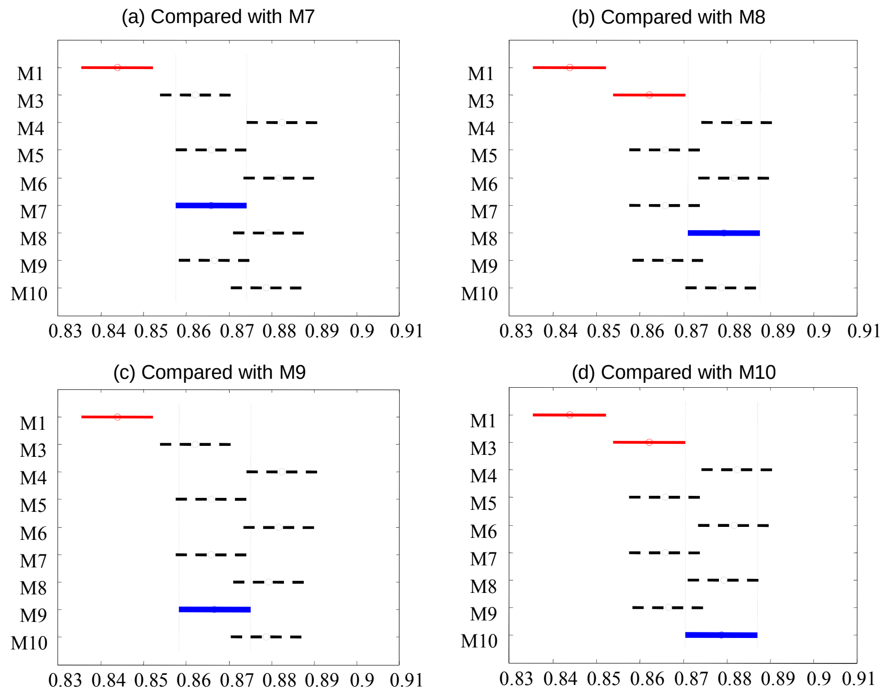


Figure 10. Multiple comparisons among M1 and M3-M10 using coverage-adequate suites.

(The *y*-axes show the techniques and the *x*-axes show the APFD values.)

ence between the techniques is deemed to be statistically significant. For each of the subject programs, ANOVA returns a p-value much less than 0.05, which successfully rejects the null hypothesis at a significance level of 5%. Since there is no outliner or exception case in the ANOVA results, we will not list the detailed statistics.

Following Rothermel et al. (2001), we further conduct multiple comparisons (Li et al., 2007; Jiang et al., 2009) to study whether the means of test case prioritization techniques differ significantly from one another at a significance level of 5%. We present the multiple comparison results in Figure 10, styled after Jiang et al. (2009) and generated by MATLAB using the default setting of alpha value = 0.05 and ctype = "hsd" (for Tukey's honestly significant difference criterion). The thick (blue) lines represent the target technique to be compared with other techniques. The thin (red) lines represent the techniques whose means differ significantly from the target technique, while the dashed (black) lines represents techniques comparable to the target technique.

First, we compare M7–M8 with M1. We observe that the pairs (M1, M7) and (M1, M8) are significantly different. This observation shows that our black-box testing techniques are better than random ordering in terms of the overall mean APFD. On the other hand, when we compare M7–M8 with M3–M6, we do not observe a significant difference apart from the pair (M3, M8). This indicates that the overall performance of M3–M6 is close to that of M7–M8. This result also verifies the observation that we have discussed in Figures 8(1)–(4) above.

When comparing M7 with M8, the result shows that recursively selecting test cases from the highest coverage to the lowest coverage of WSDL tags is more likely to achieve a higher fault detection rate. However, we do not find a significant difference between M7 and M8 using the overall APFD values.

Next, we compare M7−M8 with M9−M10. We do not observe significant differences between them. We can observe that the performances of M7 and M9 are similar, and those of M8 and M10 are similar. The differences between M9−M10 and M1−M6 are similar to those between M7−M8 and M1−M6. This shows different ways of counting the number of WSDL tags (namely, whether multiple occurrences of a WSDL tag should be counted as one) does not produce very different results on the benchmark applications we have used.

5.2.4. *Comparisons of randomly created test suites*

In the experimental setting presented in Section 5.1, we follow some specified criterion for the adequacy of a test suite (say, each workflow transition must have been covered by at least one test case). The test suite construction process randomly adds a test case to the test suite (initially empty) until the criterion has been met. Nevertheless, it is sometimes infeasible to create such test suites for a black-box service whose internal structure is not available. In such situations, we randomly add test cases to a test suite (initially empty) until a predefined size has been met. In this way, we have created 100 test suites having the same sizes as those in Table 7, and examine the fault detection rates of M1–M10 on these test suites. We further note that if the service is a black-box service whose internal structure is not known, it is generally impractical to apply M3–M6 in reordering test cases. However, we only use M3–M6 as benchmarks.

The result for each application is given in Figures 11(a)–(h). The overall 25th, 50th, and 75th percentiles as well as the overall means of all subject programs are shown in Figures 11(1)–(4). M1 and M2 show the worst and the best performances in all these figures.

We first use the 25th percentile APFD values to compare M1–M10 in Figure 11(1). All of M3–M10 demonstrate better results than M1. The white-box techniques have results similar to the black-box techniques. However, in terms of the 25th value (the bottom transversal line of the box) and the 75th value (the top transversal line of the box) in the box plots, the black-box techniques show slightly better results than the white-box techniques. This shows that our black-box techniques are close to (and even a little better than) white-box techniques in the worst orderings generated.

Then, we use the median APFD (that is, the 50th percentile) to compare M1–M10 in Figure 11(2). The white-box techniques (except M3) have results similar to those of M7–M10. Compared with Figure 8(2), we observe that the variances reported by M3–M6 in Figure 11(2) are smaller than those in Figure 8(2).

Next, we use the 75th percentile APFD values to compare M1–M10 in Figure 11(3). M3–M6 all show slightly better results than M7–M10. M6 shows the best performance among M3–M10 from many perspectives, such as the highest 25th percentile, median, and 75th percentile, as well as the smallest variance. These observations demonstrate that M8 and M10 are close in performance to the white-box testing techniques, while M7 and M9 are slightly worse than the white-box techniques.

Finally, we compare the mean APFD values for M1–M10 in Figures 8(4) and 11(4). In Figure 11(4), the differences in mean APFD values between M7 and M8 and between M9 and M10 are similar to their differences in Figure 8(4). We can, however, observe larger variances of M7–M10 in Figure 11(4) than in Figure 8(4). Since the randomly created test suites may be coverage-inadequate, this observation indicates that the performances (in terms of mean APFD values) of M7−M10 may have more differences on coverage-inadequate test suites than on coverage-adequate test suites.

When comparing M7 and M9 with M8 and M10 in Figures 11(1)−(4), we observe that the use of the *descending* strategies (M8 and M10) may be more effective than the *ascending* strategies (M7 and M9).

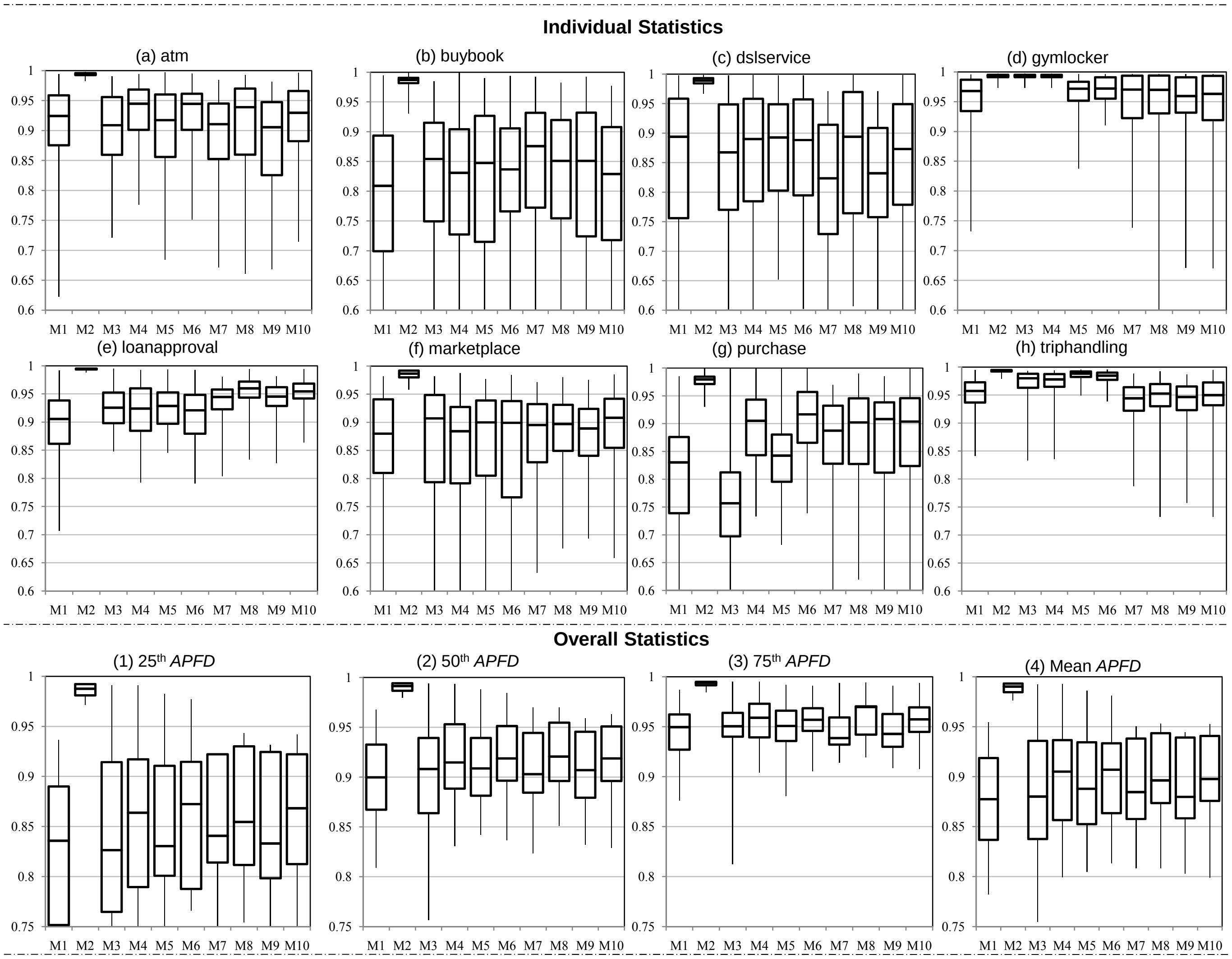


Figure 11. Comparisons APFD measures among M1–M10 using randomly created test suites.

(The *x*-axes show the techniques and the *y*-axes show the APFD values.)

M3–M6 have larger variances in Figure 11(4) than in Figure 8(4). Among them, M4–M6 show better results than M7–M10 in terms of the median values in the box plots. However, the results of M7–M10 are much better than random prioritization. M8 and M10 are slightly better than M3–M6 in terms of the 25th percentile values in the box plots. This indicates that our black-box techniques are useful.

We also compare the performance of each application in Figures 8 and 11 to gain more insight. In summary, random ordering achieves slightly better results in Figure 11 than in Figure 8. This shows that random prioritization may be slightly affected by the construction procedure of the regression test suites. Nevertheless, random ordering still cannot be better than black-box techniques in most cases.

We observe through the comparison of Figures 11(1)–(4) with Figures 8(1)–(4) that our black-box techniques outperform random prioritization using either type of test suite. When comparing the performances of M3–M10 between Figures 11(1)–(4) and Figures 8(1)–(4), we find that the variances of these techniques in Figure 11 are larger than those in Figure 8. The relative performances between the white-box techniques M3–M6 and the black-box techniques M7–M10 remain similar. Since the construction procedures of both adequate-coverage test suites and random test suites consist of random test case selection, the results may thus differ from each other within a small range on all the benchmark applications.

We also observe from Figure 11 that arranging the test suite in descending order achieves slightly higher medium APFD values than ascending order.

5.2.5. *Hypothesis testing of randomly created test suites*

We continue to present the multiple comparison (Carmer and Swanson, 1971) results of M1 and M3–M10

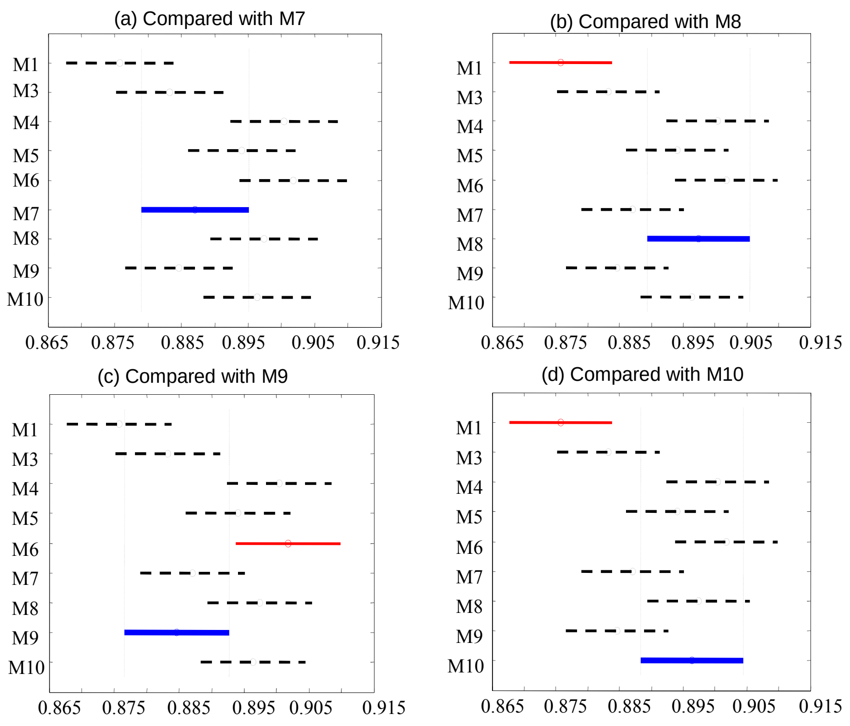


Figure 12. Multiple comparisons among M1 and M3-M10 using randomly created test suites.

(The *y*-axes show the techniques and the *x*-axes show the APFD values.)

in Figure 12, also generated using MATLAB. Similar to Figure 10, the thick (blue) lines represent the target technique to be compared with other techniques. The thin (red) lines represent the techniques whose means differ significantly from the target technique, and the dashed (black) lines represents techniques comparable to the target technique.

We observe that both M8 and M10 significantly outperform random prioritization. Although M7 and M9 are better than M1, the differences are not statistically significant. M6 achieves significantly better results than M9. However, apart from M6 and M9, there is no significant difference between the white-box techniques and the black-box techniques. Furthermore, M4–M6 all significantly outperform M1. For the sake of brevity, we do not show them in Figure 12.

Compared with the hypothesis testing results in Figure 10, the differences of M1 and M3 from the black-box techniques are less significant. In any case, the results show that our techniques can have a high chance of outperforming random ordering, and two of the black-box techniques are close to the white-box techniques.

5.3. *Threats to validity*

Threats to construct validity arise if the measuring instruments do not adequately capture the concepts they are supposed to measure. In the experiment, we use the fault detection rate to measure the effectiveness of M1–M10. We choose the APFD metric, which has been widely adopted in test case prioritization (such as Elbaum et al., 2002) to measure the fault detection rate of prioritization techniques. There are other metrics proposed for evaluating the prioritization techniques (such as Li et al., 2007). Different metrics serve to measure different aspects of a testing technique. In our experiment, we assume that the test oracle of each regression test case is reliable. In practice, however, some of the failures miss to be identified. In such cases, the APFD values presented in this paper are different from the actual APFD values. However, in our controlled experiment setting, the same test case is applied by each of the techniques to the same modified version of each subject. Moreover, APFD treats all faults equally. We categorize all the faults into three categories, namely, BPEL faults, XPath faults, and WSDL faults, and compute the APFD results for each category to gain more insight.

Although we have tried our best to search for publicly available benchmark programs, we have not found such programs with documentation of real-life faults. We have therefore used seeded faults in the experiment.

Threats to internal validity are the influences that can affect the dependency of experimental variables involved. When executing a test case on a service composition, the contexts of the involved services may affect the outcome of the test case, making the evaluation result inconclusive. We have proposed a framework (Mei, 2009a) to address this problem. In this paper, our experiment tool resets the run-time contexts of services to the required values for each test case.

External validity is concerned with whether the results are applicable to the general situation. Eight WS-BPEL programs have been included in the experiment. Each program can be considered as a service. These services were not large in size and may not be truly representative of the many different types of web service, which may also be written in diverse programming languages. Moreover, the same WSDL documents may not be applicable to programs of different scales. We will conduct other studies to verify the techniques further. We also plan to collaborate with the industry to design benchmark service-oriented programs in larger distributed computing environments, and to evaluate our proposed prioritization techniques to gain more insight.

In our experiment, we have not used multi-fault or other single-fault service-oriented applications to verify our techniques. Although using more faulty versions may strengthen how an experiment addresses the external threats to validity, it involves more effort in evaluating each technique. In our experience in service-oriented testing, in order to conduct a test experiment, researchers require the setting up of the underlying platform such as web servers, BPEL engines, and database servers. It takes non-trivial effort to enable these platform applications to support test experiments. For instance, sometimes the BPEL engine may simply hang and does not respond as usual when processing messages. In such a scenario, researchers are required to divert effort not only to run their tests, but also to diagnose and fix/bypass the faults in such a testing environment. Compared with the effort to run test experiments on more traditional subjects (such as the Siemens suite (Elbaum et al., 2000)), the effort to run experiments that use service-oriented programs as subjects is significantly more immense. To balance between our available human resources and the completeness of the experiment, we choose not to include other faulty versions in the current study.

Another threat to validity is the correctness of our tools. We have used Java to develop our tools for program instrumentation and test case prioritization, used PTS box chart utility to draw the box plots, and used MATLAB to do the statistical analysis. To minimize errors, we have carefully tested our tools to assure their correctness. However, we are unable to conduct a thorough test of PTS box chart utility and MATLAB.

We use a random test case generator to construct random test cases for the required test suites. As shown in many previous studies, the use of test suites fulfilling different testing criteria (such as branch coverage) may result in different fault detection capabilities. This will also affect the APFD values when comparing different test case prioritization techniques. We use the same number of test sets for each subject version. The use of different numbers of test sets for different versions may affect the results.

6. Discussion

We have assumed in our model that the interface specification of a service specifies how the test suite services and test case services will be called. However, this constraint may be relaxed in many ways, such as using method invocation, centralized services, and UDDI registries. For example, a service provider may provide an entry in its WSDL documents in the UDDI registries to illustrate how its test suite service can be accessed.

We have identified the key roles in black-box service-oriented testing in Section 4.2, and have modeled these roles by services. Based on such modeling, many interesting approaches can be considered, such as how to represent the flow of regression testing using a format (e.g., WSDL) that can be released to public registries, and how our proposed strategy can be dynamically adapted according to the feedback from regression testing after a round of test suite evaluation of the target services.

In our problem setting, we consider that the internal structure of a workflow service may not be available, and thus develop techniques to address regression testing challenges when only exchanged messages between individual workflow steps and services are available. However, if more information is known when test cases are executed (such as the coverage information of the internal structures of services), we can further derive new techniques by adapting our proposed techniques. For example, we may use the new information to prioritize the tie cases resulting from our black-box testing techniques. In addition, we may also apply our black-box testing techniques to prioritize the tie cases resulting from white-box testing techniques if the relevant white-box information is available.

To the best of our knowledge, existing test case prioritization techniques on service-oriented programs either have their own assumptions (such as resource constraints in Hou et al., 2008), or are not black-box test case prioritization techniques (see, for instance, Mei et al., 2009d). Hence, we only compare our black-box testing techniques with white-box techniques and two control techniques (random and optimal).

The experimental results have shown that our black-box techniques can have a high chance of outperforming random ordering, and two of our techniques are close to white-box techniques. The results have also indicated that our black-box techniques are close to (and even a little better than) white-box techniques in the worst orderings generated for either adequate-coverage test suites or random test suites. These observations suggest that our techniques may serve as viable choices when conducting regression testing for service-oriented programs.

7. Related work

Regression testing is a testing procedure conducted after modifications of a program (Leung and White, 1989). It has been widely used in the industry (Onoma et al., 1998). Leung and White (1989) have pointed out that simply rerunning all existing tests is not an ideal approach. Test case prioritization aims to reorder test cases to maximize a testing goal (Rothermel et al., 2001), and is one of the most important lines in regression testing research.

Many coverage-based prioritization techniques, such as Elbaum et al. (2002) and Rothermel et al. (2001), have been proposed. A large number of these techniques prioritize test cases by means of the code coverage achieved, such as the number of statements or branches covered by individual test cases (Rothermel et al., 2001). Furthermore, although there are header files and preprocessing macros in C programs, existing code-based prioritization techniques that use C programs as subjects do not explore such information. Specific examples of criteria used include fault-exposing potential of individual test cases (Rothermel et al., 2001), the series of regression history (Kim and Porter, 2002), and test costs and fault severities (Elbaum et al., 2002). In the service-oriented environment, because of the limited knowledge of potential evolutions of a target service as well as the classifications of previously identified faults in fault-based techniques, it is not clear how history-based or cost-based techniques can be applied effectively. The effects of compositions and granularity (Rothermel et al., 2002) of a test suite have been studied experimentally in conventional testing. To the best of our knowledge, however, these two aspects have not been examined in the context of service-oriented testing.

Hou et al. (2008) consider the issue of service invocation quota in a regression testing technique. Specifically, they observe that some web services may not be invoked without limit. They use such quota constraints as the metric to guide the prioritization of test cases. In our problem setting, we do not have any quota constraint. We evaluate how well the number of XML tags is used as a metric to reorder test cases. Zhai et al. (2010) also study the impact of service invocation in regression testing techniques. Rather than setting up a quota to constrain the number of invocations of a service, they integrate the notion of service-oriented architecture to their techniques. They observe that, to bind a service, a set of candidate services satisfying the required quality-of-service constraint may be discovered. Nonetheless, a typical service selection process will discard all but one of such services. In verifying the set of candidate services, they propose to discard a service permanently once it is found to be faulty. Consequently, their technique can reduce the average number of service invocations and improve the identification rate of faulty services. Their approach does not optimize the fault detection rate of individual services. On the other hand, our work does not use service-oriented architecture as its core, but focuses on improving the fault detection rate for a given service under test. It will be interesting to explore the integration of these two dimensions.

Our work is also related to the area of testing third-party web services. Brenner et al. (2007) study different general testing approaches that can be used compatibly to verify such web services. Zhai et al. (2010) focus on eliminating third-party web services as early as possible from subsequent service selection considerations. Their focus is not on the testing of such third-party web services. Bartolini et al. (2009, 2010) propose a framework to collect and report the coverage statistics achieved by a test on a service. We focus on using the information captured in messages to guide the testing process. None of these studies examines this information.

Next, we review related work in the area of service-oriented computing. Martin et al. (2007) outline a framework that generates and executes web service requests, and collects the corresponding responses from web services. They propose to examine the robustness aspect of services by perturbing such request-response pairs. They have not studied test case prioritization. Tsai et al. (2005a) recommend using an adaptive group testing technique to address the challenges in testing service-oriented applications when a large number of web services are available. They rank test cases according to a voting mechanism on input-output pairs. Neither the source code of the service nor the structure of WSDL is utilized.

Mei et al. (2008b) use the mathematical definitions of XPath (W3C, 2007b) as rewriting rules, and propose a data structure known as an *XPath Rewriting Graph* (*XRG*). They further develop the notion of XRG patterns to capture how different XRGs are related even though they may refer to different XML schemas or tags (Mei et al., 2009b). They have developed test case prioritization techniques (Mei et al., 2009c) on top of their XRG structure. However, they have not studied whether WSDL can be used in a standalone manner for regression testing of services when the source code is unavailable or too costly to acquire.

In the area of test oracles, Tsai et al. (2005b, 2008) observe that many web services may produce the same output for the same input. They propose to use whether the outputs of such web services agree with one another as a means to identify web services with desirable outputs by the use of a variant of the majority vote strategy. Bai et al. (2007) and Bai and Kenett (2009) further propose a framework to support this group testing proposal by enforcing check-in and check-out features of web service registries, and study how to prioritize risky (also known as harmful) test cases to be executed earlier. Dai et al. (2007) and Di Penta et al. (2007) propose to add contracts to service descriptions to serve as a kind of correctness criterion (or test oracle) to check test results. Chan et al. (2005, 2007) use metamorphic relations of applications to test stateless scientific web services. Our experiment uses the expected results captured for the regression test cases. Our work has not considered the strengths of test oracles used to identify failures.

There are recent studies on fault-tolerant web services (Aghdaie and Tamir, 2009) and (self-)adaptive web services. They are not considered in our experiment. Nonetheless, we believe that our techniques can be applied to validate such web services.

8. Conclusion

In a service composition, a member service may evolve without any prior agreement from peer services. Other services that relate to this member service may need to conduct regression testing to verify whether the functions of the latter service conform to their established inter-operability requirements. Even though the former services want to conduct regression testing, the latter service has been encapsulated with only an observable interface, making it difficult for the former services to apply existing code-based test case prioritization techniques to achieve certain prioritization goals.

This paper studies whether the use of WSDL information may facilitate effective regression testing of services. To the best of our knowledge, it is among the pioneering work that formulates black-box regression testing to verify web services. The paper also proposes a set of roles to support black-box service-oriented regression testing and outlines a scenario to illustrate how to collaborate these roles. We have also discussed how a cloud computing infrastructure may conceptually be helpful to service-oriented testing. Specifically, by using a cloud computing concept, a test does not need not to bear the cost of monitoring and communicating with those services that have been excluded from the current test.

With regard to regression testing techniques, this paper has proposed to compute the WSDL tag coverage from the input and output messages associated with regression test suites, and formulated four black-box test case prioritization techniques to verify the concepts. We have evaluated our techniques, compared them with both traditional techniques and random ordering using a suite of WS-BPEL applications in a controlled experimental setting. The empirical results have indicated that, overall, our black-box testing techniques are only slightly less effective than white-box techniques, but not to the level of statistical significance, and that they can significantly outperform random ordering in achieving higher rates of fault detection.

Since WSDL is only an interface specification, it may be blind to certain regression fault types. For future work, one may enhance the proposed techniques by integrating with other black-box techniques. Moreover, when testing can be done in a field environment, it can be considered as a behavior monitoring approach. This is particularly viable when testing in a cloud computing infrastructure. Such kind of infrastructure and dynamic testing environment, as outlined in the paper, warrants further research.

References

Aghdaie, N., Tamir, Y., 2009. CoRAL: a transparent fault-tolerant web service. Journal of Systems and Software 82 (1), 131–143.

Apache Software Foundation, 2006. Web Services Invocation Framework: DSL Provider Sample Application. Available at http://ws.apache.org/wsif/wsif_samples/.

Bai, X., Cao, Z., Chen, Y., 2007. Design of a trustworthy service broker and dependence-based progressive group testing. International Journal of Simulation and Process Modelling 3 (1/2), 66–79.

Bai, X., Kenett, R.S., 2009. Risk-based adaptive group testing of semantic web services. In: Proceedings of the 33rd Annual International Computer Software and Applications Conference (COMPSAC 2009), vol. 2. IEEE Computer Society, Los Alamitos, CA, pp. 485–490.

Bartolini, C., Bertolino, A., Elbaum, S.G., Marchetti, E., 2009. 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). ACM, New York, NY, pp. 161–170.

Bartolini, C., Bertolino, A., Elbaum, S.G., Marchetti, E., 2010. Bringing white-box testing to service oriented architectures through a service oriented approach. Journal of Systems and Software. doi: 10.1016/j.jss.2010.10.024.

Bartolini, C., Bertolino, A., Marchetti, E., Polini, A., 2008. Towards automated WSDL-based testing of web services. In: Service-Oriented Computing (ICSOC 2008), Lecture Notes in Computer Science, vol. 5364. Springer, Berlin, Germany, pp. 524–529.

Brenner, D., Atkinson, C., Hummel, O., Stoll, D., 2007. Strategies for the run-time testing of third party web services. In: Proceedings of the IEEE International Conference on Service-Oriented Computing and Applications (SOCA 2007). IEEE Computer Society, Los Alamitos, CA, pp. 114–121.

Canfora, G., Di Penta, M., 2006. SOA: testing and self-checking. In: Proceedings of the International Workshop on Web Services: Modeling and Testing (WS-MaTe 2006). Palermo, Italy, pp. 3–12.

Chan, W.K., Cheung, S.C., Leung, K.R.P.H., 2005. 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, pp. 470–476.

Chan, W.K., Cheung, S.C., Leung, K.R.P.H., 2007. A metamorphic testing approach for online testing of service-oriented software applications. International Journal of Web Services Research 4 (2), 60–80.

Chen, H.Y., Tse, T.H., Chan, F.T., Chen, T.Y., 1998. In black and white: an integrated approach to class-level testing of object-oriented programs. ACM Transactions on Software Engineering and Methodology 7 (3), 250–295.

Dai, G., Bai, X., Wang, Y., Dai, F., 2007. Contract-based testing for web services. In: Proceedings of the 31st Annual International Computer Software and Applications

Conference (COMPSAC 2007). IEEE Computer Society, Los Alamitos, CA, pp. 517–526.

DeMillo, R.A., Lipton, R.J., Sayward, F.G., 1978. Hints on test data selection: help for the practicing programmer. IEEE Computer 11 (4), 34–41.

Di Penta, M., Bruno, M., Esposito, G., Mazza, V., Canfora, G., 2007. Web services regression testing. In: Test and Analysis of Web Services, Springer, Berlin, Germany, pp. 205–234.

Elbaum, S.G., Malishevsky, A.G., Rothermel, G., 2000. Prioritizing test cases for regression testing. In: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2000). ACM, New York, NY, pp. 102–112.

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

Harrold, M.J., Gupta, R., Soffa, M.L., 1993. A methodology for controlling the size of a test suite. ACM Transactions on Software Engineering and Methodology 2 (3), 270–285.

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

Hutchins, M., Foster, H., Goradia, T., Ostrand, T., 1994. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In: Proceedings of the 16th International Conference on Software Engineering (ICSE 1994). IEEE Computer Society, Los Alamitos, CA, pp. 191–200.

IBM, 2006. BPEL Repository. Available at http://www.alphaworks.ibm.com/tech/bpelrepository.

Jia, Y., Harman, M., 2009. Higher order mutation testing. Information and Software Technology 51 (10), 1379–1393.

Jiang, B., Zhang, Z., Chan, W.K., Tse, T.H., 2009. Adaptive random test case prioritization. In: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009). IEEE Computer Society, Los Alamitos, CA, pp. 233–244.

Kim, J.-M., Porter, A., 2002. 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). ACM, New York, NY, pp. 119–129.

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

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

Martin, E., Basu, S., Xie, T., 2007. Automated testing and response analysis of web services. In: Proceedings of the IEEE International Conference on Web Services (ICWS 2007). IEEE Computer Society, Los Alamitos, CA, pp. 647–654.

Mei, L., 2009a. 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). IEEE Computer Society, Los Alamitos, CA, pp. 371–374.

Mei, L., Chan, W.K., Tse, T.H., 2008a. An adaptive service selection approach to service composition. In: Proceedings of the IEEE International Conference on Web Services (ICWS 2008). IEEE Computer Society, Los Alamitos, CA, pp. 70–77.

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

Mei, L., Chan, W.K., Tse, T.H., 2008c. A tale of clouds: paradigm comparisons and some thoughts on research issues. In: Proceedings of the 2008 IEEE Asia-Pacific Services Computing Conference (APSCC 2008). IEEE Computer Society, Los Alamitos, CA, pp. 464–469.

Mei, L., Chan, W.K., Tse, T.H., 2009b. 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). ACM, New York, NY, pp. 151–160.

Mei, L., Chan, W.K., Tse, T.H., Merkel, R.G., 2009c. Tag-based techniques for black-box test case prioritization for service testing. In: Proceedings of the 9th International Conference on Quality Software (QSIC 2009). IEEE Computer Society, Los Alamitos, CA, pp. 21–30.

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

OASIS, 2007. Web Services Business Process Execution Language Version 2.0: OASIS Standard. Available at http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html.

Offutt, A.J., 1992. Investigations of the software testing coupling effect. ACM Transactions on Software Engineering and Methodology 1 (1), 5–20.

Onoma, A.K., Tsai, W.-T., Poonawala, M., Suganuma, H., 1998. Regression testing in an industrial environment. Communications of the ACM 41 (5), 81–86.

Oracle Technology Network. Oracle BPEL Process Manager. Available at http://www.oracle.com/technology/products/ias/bpel/. (Last access on June 29, 2009.)

Rothermel, G., Elbaum, S.G., Malishevsky, A., Kallakuri, P., Davia, B., 2002. The impact of test suite granularity on the cost-effectiveness of regression testing. In: Proceedings of the 24th International Conference on Software Engineering (ICSE 2002). ACM, New York, NY, pp. 130–140.

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

Tsai, W.-T., Chen, Y., Paul, R., Huang, H., Zhou, X., Wei, X., 2005a. Adaptive testing, oracle generation, and test case ranking for web services. In: Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC 2005), vol. 1. IEEE Computer Society, Los Alamitos, CA, pp. 101–106.

Tsai, W.-T., Chen, Y., Zhou, X., Bai, X., 2005b. Web service group testing with windowing mechanisms. In: Proceedings of the IEEE International Symposium on Service-Oriented System Engineering (SOSE 2005). IEEE Computer Society, Los Alamitos, CA, pp. 221–226.

Tsai, W.-T., Zhou, X., Chen, Y., Bai, X., 2008. On testing and evaluating service-oriented software. IEEE Computer 41 (8), 40–46.

W3C, 2007a. Web Services Description Language (WSDL) 2.0. Available at http://www.w3.org/TR/wsdl20.

W3C, 2007b. XML Path Language (XPath) 2.0 W3C Recommendation. Available at http://www.w3.org/TR/xpath20/.

Xu, W., Offutt, J., Luo, J., 2005. Testing web services by XML perturbation. In: Proceedings of the 16th International Symposium on Software Reliability Engineering (ISSRE 2005). IEEE Computer Society, Los Alamitos, CA, pp. 257–266.

Ye, C., Cheung, S.C., Chan, W.K., Xu, C., 2009. Atomicity analysis of service composition across organizations. IEEE Transactions on Software Engineering 35 (1), 2–28.

Zhai, K., Jiang, B., Chan, W.K., Tse, T.H., 2010. 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). IEEE Computer Society, Los Alamitos, CA, pp. 211–218.

Lijun Mei obtained his PhD degree in computer science from The University of Hong Kong. He also received his BEng and MSc degrees in computer science from Tsinghua University, Beijing, China. His research interests include testing and analyzing of software engineering issues, particularly service-oriented applications. He has published in various journals and conferences, including ICSE 2008, WWW 2009 and FSE 2009. He has been selected for an IBM PhD fellowship and an HKSAR Government Scholarship. He is currently a staff researcher at IBM China Research Laboratory.

W.K. Chan is an assistant professor at City University of Hong Kong. He received his PhD degree from The University of Hong Kong. His main research interest is in software engineering issues in program testing and analysis, and service composition. He is on the editorial board of Journal of Systems and Software

T.H. Tse is a professor in computer science at The University of Hong Kong. He received his PhD from the London School of Economics and was twice a visiting fellow at the University of Oxford. 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 Journal of Universal Computer Science. 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 decorated with an MBE by The Queen of the United Kingdom.

Robert G. Merkel is a Lecturer in software engineering at Monash University, Melbourne, Australia. He received his PhD from Swinburne University of Technology. His research interests include random testing, the theory of software testing, and the testing of embedded systems.