2009 Ninth International Conference on Quality Software

Tag-Based Techniques for Black-Box Test Case Prioritization for Service Testing*

Lijun Mei
The University of Hong Kong
Pokfulam, Hong Kong
ljmei@cs.hku.hk

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

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

Robert G. Merkel
Swinburne University of Technology
Melbourne, Australia
rmerkel@ict.swin.edu.au

***Abstract*—A web service may evolve autonomously, making peer web services in the same service composition uncertain as to whether the evolved behaviors may still be compatible to its originally collaborative agreement. Although peer services may wish to conduct regression testing to verify the original collaboration, the source code of the former service can be inaccessible to them. Traditional code-based regression testing strategies are inapplicable. The rich interface specifications of a web service, however, provide peer services with a means to formulate black-box testing strategies. In this paper, we 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 services. We evaluate experimentally their effectiveness on revealing regression faults in modified WS-BPEL programs. The results show that the new techniques can have a high probability of outperforming random ordering.**

***Keywords*—test case prioritization; black-box regression testing; WS-BPEL; service testing; encapsulation testing**

I. INTRODUCTION

The testing and analysis of web services has posed new foundational and practical challenges, such as the non-observability of when a web service may evolve [3][15], the extensive presence of non-executable artifacts within and among web services [16], safeguards against malicious messages from external parties [13], and ultra-late binding and cross-organizational issues [1]. Researchers have proposed diverse techniques to address the test case generation problem [13], the test adequacy problem [16], the test oracle problem [5][23], and the test case prioritization problem [18].

Regression testing is the de facto activity to address the testing problems due to software evolution. However, many existing regression testing techniques (e.g., [8][10][18][22]) assume that the source code is available [16], and use the coverage information of executable artifacts (e.g., statement coverage of test cases) to conduct regression testing. When such 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*) [25] to specify their functional interfaces and message parameters. They also use XML documents to represent the messages [24]. 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. Thus, WSDL documents are rich in interface information. We assume that such documents are observable to external services. Despite the richness of such documents, to our best knowledge, the use of WSDL documents to guide regression testing without assuming code availability have not been proposed or evaluated in the literature.

In general, different services developed by the same or multiple development teams may be modified autonomously by their maintainers, and the evolution of services may not be fully known by every other service. Let us consider a scenario that a service *A* would like to pair up with a service *B*, and yet the latter service may evolve over time. *A* may want to execute some tests on the service provided by *B* to ensure that *A*'s composition has not been adversely affected (at least from *A*'s perspective). The program code of *B* is generally inaccessible to *A*. Therefore, even though *A* may have a test suite to conduct regression testing on *B*, it cannot apply existing code-based regression testing techniques [8][11] in general, and test case prioritization techniques [22] in particular, to improve the fault detection rate and achieve other goals.

A test case prioritization technique aims to reorder test cases to achieve certain goals (such as detecting program faults earlier [7]). Because of the internal logic and states of a service are encapsulated by the service boundary, the advantages of existing test case prioritization techniques (such as improved fault detection rate [6][22]) cannot be realized effectively in service-oriented testing.

The WSDL documents of services are accessible among peer services. It is well known, however, that black-box testing is generally less effective than white-box testing. *Can the richness of WSDL documents help to narrow this gap? Is it effective to use the black-box information in WSDL documents to guide regression testing to overcome*

* This research is supported in part by the General Research Fund of the Research Grant Council of Hong Kong (project nos. 111107, 717308, and 717506), the Strategic Research Grant of City University of Hong Kong (project no. 7002464), and a discovery grant of the Australian Research Council (project no. DP0984760).

† Corresponding author.

1550-6002/09 $26.00 © 2009 IEEE
DOI 10.1109/QSIC.2009.12

the difficulties in testing services with hidden implementation details such as source code?

General test case prioritization and version-specific test case prioritization are two important kinds of technique in regression testing [22]. A technique of the former kind will sort a test suite for a program *P* with the intent of being useful over subsequent modified versions of *P*; whereas a technique of the latter kind will prioritize a test suite specific to a particular modified version of the program under test. In scenarios similar to the above, the source code of a service is generally inaccessible, limiting the applicability of version- specific techniques. General test case prioritization does not suffer from this problem. This paper studies general test case prioritization.

We observe that, in a regression test suite, existing test cases 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. Following up on these observations, we propose two directions in test case prioritization (based on XML tags of WSDL documents) to generate ordered regression test suites. We implement two prioritization techniques as proofs of the concepts under the two proposed directions.

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 of the services to validate 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. 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.

We further conduct an empirical study to verify the effectiveness of our techniques. The results show that they can be promising in their rates of fault detection.

The main contribution of this paper is twofold: (i) We propose a new set of black-box techniques to prioritize test cases for regression testing of services with observable and rich content interface. It addresses 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 report the first controlled experiment of its kind to evaluate the effectiveness of such techniques. Our empirical results indicate that the use of the information captured in WSDL documents (paired with regression test suites) can be promising in lowering the cost of the quality assurance of workflow services.

The rest of the paper is organized as follows: Section II reviews related work. Section III introduces the preliminaries of our approach through a running service scenario. Section IV discusses the challenges in regression testing of services, and presents new test case prioritization techniques for regression testing of services. Section V presents an experiment design and its results to evaluate our proposal. Section VI concludes the paper and discusses future work.

II. PRELIMINARIES

This section reviews related work and introduces the terminology in test case prioritization.

A. Related Work

Regression testing is a testing procedure conducted after modifications of a program [11]. It has been widely used in industry [19]. Leung and White [11] have pointed out that simply re-running all existing tests is not an ideal approach. Test case prioritization aims to reorder test cases to maximize a testing goal [22], and is one of the most important lines in regression testing research.

Many coverage-based prioritization techniques (e.g., [7][22]) have been proposed. A large number of these techniques prioritize test cases by means of code coverage achieved, such as the number of statements or branches covered by individual test cases [22]. 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 techniques used include fault-exposing potential of individual test cases [22], the series of regression history [10], and test costs and fault severities [7]. In the service environment, because of the limited knowledge of potential evolutions of a target service as well as the fault 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 [21] of a test suite have been studied experimentally for conventional testing. To our knowledge, however, these two aspects have not been examined in the context of service-oriented testing.

Martin et al. [13] 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. [23] 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. [16] use the mathematical definitions of XPath [27] as rewriting rules, and propose a data structure known as an *XPath Rewriting Graph* (*XRG*). They [17] 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. They have developed test case prioritization techniques [18] 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 be acquired.

B. Test Case Prioritization

Test case prioritization [22] is an important kind of regression testing technique [12]. Through the use of information from previous rounds of software evaluation, we can design techniques to rerun the test cases to achieve a certain goal (such as to increase the fault detection rate of a test suite). We adopt the test case permutation problem from [22] 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.)

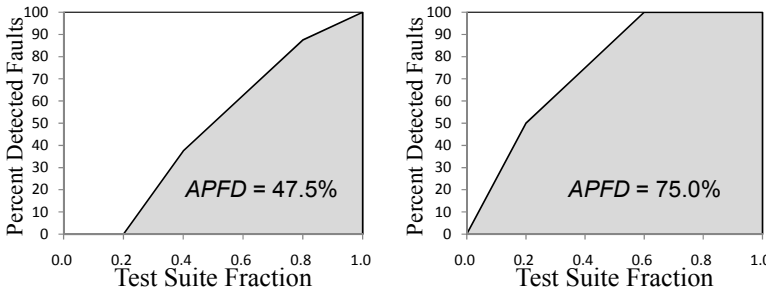
Problem: To find $T' \in PT$ such that $\forall T'' \in PT$, $f(T') \geq f(T'')$.

The *Average Percentage of Faults Detected* (*APFD*) [7] 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 the authors of [7] 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:

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

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



(a) *APFD* for test suite T_1 (b) *APFD* for test suite T_2

Figure 1. Example illustrating the *APFD* measure (from [18]).

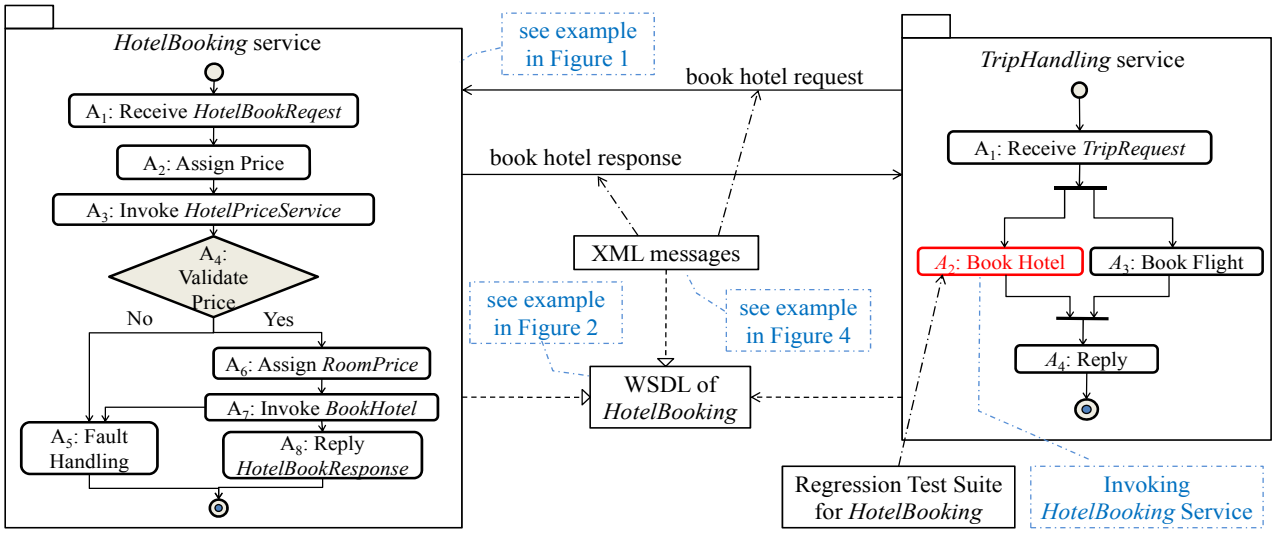


Figure 2. Service interaction scenario.

We also adopt an example from [18] 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.

III. Scenario of Running Service

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 [2]. Figure 3 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 information (such as an XPath) extracted from the program. We label the nodes as A_i for i = 1 to 8. The service aims to find a hotel room whose price is not more than a ceiling set by the user.

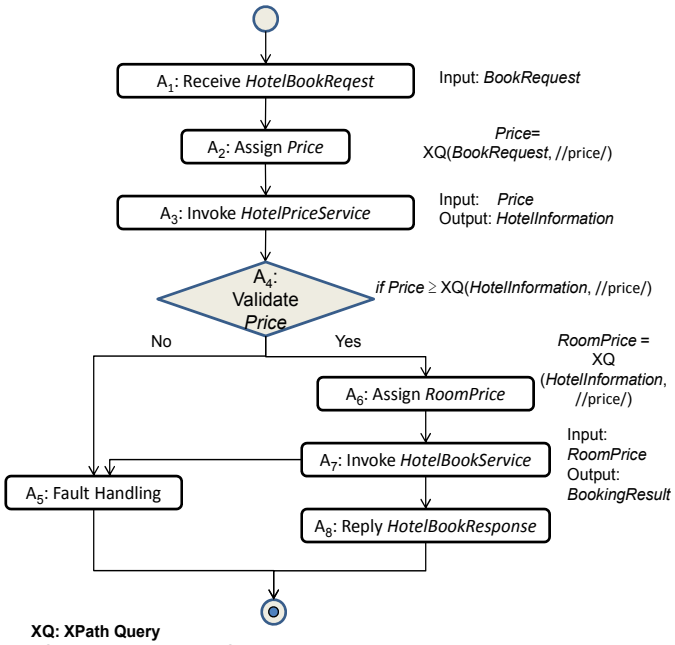


Figure 3. Activity diagram for *HotelBooking*.

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 4. 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 8–10). This XML schema is kept in a WSDL document.

As we have highlighted in Section I, in such cross-service scenarios, the internal model of a service as shown in Figure 3 is unobservable by an external 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 2 depicts this problem.

```
1    <xsd:complexType name="hotel">
2        <xsd:element name="name" type="xsd:string"/>
3        <xsd:element name="doubleroom" type="xsd:room"/>
4        <xsd:element name="singleroom" type="xsd:room" />
5    </xsd:complexType>
6    <xsd:complexType name="room">
7        <xsd:element name="roomno" type="xsd:int" />
8        <xsd:element name="price" type="xsd:int"/>
9    </xsd:complexType>
```

Figure 4. XML schema hotel in WSDL document.

Suppose the *TripHandling* service (the rightmost part of Figure 2) 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 3) of *HotelBooking* service is hidden from *TripHandling*.

IV. Test Case Prioritization

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

A. Motivating example

Let us consider six 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* = 150
Test case 3 (t_3):	*Price* = 100	Test case 4 (t_4):	*Price* = 50
Test case 5 (t_5):	*Price* = 0	Test case 6 (t_6):	*Price* = –100

Let us discuss activity A_4. Suppose Figure 5 shows the XML messages for *HotelInformation* received in an XPath query (*HotelInformation*, //price/) by executing test cases t_1 to t_6 on the *HotelBooking* service.

Our scene further assumes that *HotelBookService* at A_7 fails in handling the third test case (t_3) because no available room satisfies this price (i.e., the value kept by the message content *RoomPrice*). Therefore, the branch $\langle A_7, A_5 \rangle$ in Figure 3 is covered by t_3. The activity coverage and workflow transition coverage of these test cases are summarized in Table 1 and Table 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 of the activities for t_4 to t_6 are identical; the same is true for the coverage of their transitions. (ii) The numbers of covered activities are the same for t_1 to t_3; the same is true for their transitions. (iii) Suppose t_1 is first selected; using the additional coverage information [22], the remaining covered activities or transitions for t_3 to t_6 are the same, while that for t_3 is 0. In such a tied case, existing test case prioritization techniques such as [22] simply order them randomly to break the ties.

<hotel> <name>Times</name> <doubleroom> <roomno>R101</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
<hotel> <name>Times</name> <doubleroom></doubleroom> <singleroom></singleroom> </hotel >	<hotel> </hotel >	null
Test Case 4	Test Case 5	Test Case 6

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

TABLE 1. ACTIVITY COVERAGE OF t_1 TO t_6.

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

TABLE 2. TRANSITION COVERAGE OF t_1 TO t_6.

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

We further show a few XML messages (Figure 5) for the hotel schema (Figure 4). For ease of presentation, we summarize the element coverage of the XML schema for t_1 to t_6 in Table 3.

TABLE 3. WSDL TAG COVERAGE FOR t_1 TO t_6.

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

We refer to an XML element z defined in any XML schema as a *WSDL tag*. If a regression test case includes an XML message that contains an XML element z, we say that z has been *covered* by the test case. In Table 3, the coverage of WSDL tags for t_4, t_5, and t_6 are different; the numbers of WSDL tags in t_2 and t_3 are different from that in t_1. If t_1 is first selected, since there is a reset procedure for coverage information, the remaining covered WSDL tags for t_3, t_4, t_5, and t_6 are still different [22]. These observations motivate us to study the use of WSDL tag coverage in prioritizing test cases.

As we have described in Section I, peer services in a service composition may not access the source code of a target service. To know whether the target service still exhibits the desirable functions as previously demonstrated, 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.

B. Our Prioritization Techniques

In this section, we first briefly review representative test case prioritization techniques for regression testing, and then introduce our adaptation.

TABLE 4. PRIORITIZATION TECHNIQUES AND EXAMPLES.

Category	Name	Index	t_1	t_2	t_3	t_4	t_5	t_6
Benchmark	Random [22]	M1	6	3	4	5	1	2
	Optimal [22]	M2	–	–	–	–	–	–
Traditional White-Box	Total-Activity [22]	M3	1	3	2	4	6	5
	Additional-Activity [22]	M4	1	4	2	5	3	6
	Total-Transition [22]	M5	2	3	1	4	5	6
	Additional-Transition [22]	M6	1	6	3	2	4	5
Our Black-Box	Ascending-WSDL-Tag	M7	5	6	4	3	2	1
	Descending-WSDL-Tag	M8	1	2	6	3	4	5

To study the effectiveness and tradeoffs of different techniques, we follow the style of [7][22] and compare other control techniques with ours. All the techniques and examples are listed in Table 4. Techniques M1–M6,

representing *random*, *optimal*, *total-statement*, *additional-statement*, *total-branch*, and *additional-branch*, respectively, are taken from [22].

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.

The coverage of all elements in a WSDL document may be easily satisfied (as illustrated, for instance, by t_1 in Table 3). If a technique schedules test cases in the order of total element coverage of the WSDL document, then t_1 will be selected first.

Based on the above analysis, we propose two techniques, namely M7 and M8.

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

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

M7 and M8 target to examine the effect of the number of WSDL tags in a test case during test case prioritization for services. Both M7 and M8 include a grouping phase that divides the test suite into groups. Two test cases in the same group contain the same number of WSDL tags (while the actual WSDL tags may be different). We then iteratively select test cases from each group. Take t_1 to t_6 as an example. Possible WSDL coverage prioritization orders for M7 and M8 are $\langle t_6, t_5, t_4, t_3, t_1, t_2\rangle$ and $\langle t_1, t_2, t_4, t_5, t_6, t_3\rangle$, respectively.

Figure 6 further summarizes the difference between black-box testing techniques (M7 and M8) 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 III). 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 (e.g., [25]) 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.

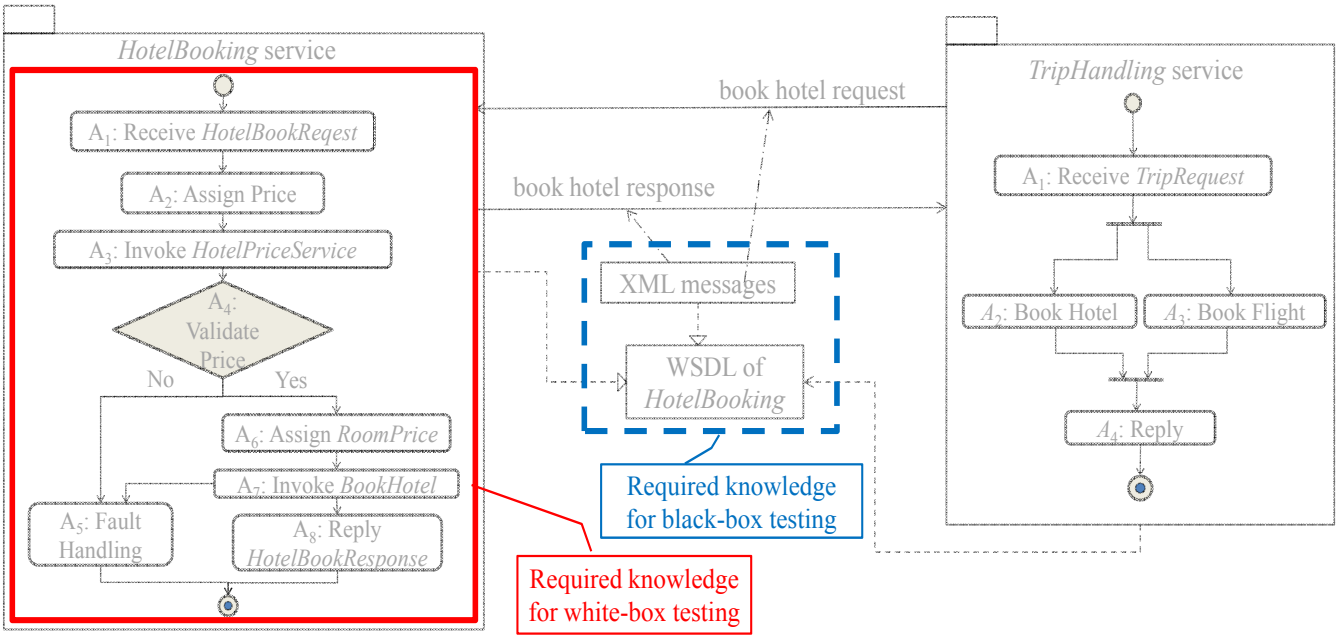


Figure 6. The different required knowledge for black- and white-box testing.

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 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.

V. EXPERIMENT

This section reports on an experimental verification of our proposal.

A. Experimental Design

1) Subject Programs, Versions, and Test Suites

We used a set of WS-BPEL applications to evaluate our techniques. The same set of artifacts have been used in the experiments in [16][18]. These applications have been used as benchmarks or examples in many WS-BPEL studies. We create a series of modified versions for each benchmark program. Every modified version is changed on top of the original version. A modified version is considered as an evolving version of an autonomous service. Like many other studies that evaluate testing techniques, we seed known faults to measure the effectiveness of the prioritization techniques. Each modified version contains a single fault.

TABLE 5. SUBJECT PROGRAMS AND STATISTICS.

Ref.	Applications	Version	Element	LOC	WSDL Spec.	WSDL Tag	No. of Versions Used
A	atm [2]	8	94	180	3	12	5
B	buybook [20]	7	153	532	3	14	5
C	dslservice [26]	8	50	123	3	20	5
D	Gymlocker [2]	7	23	52	1	8	5
E	loanapproval [2]	8	41	102	2	12	7
F	marketplace [2]	6	31	68	2	10	4
G	purchase [2]	7	41	125	2	10	4
H	triphandling [2]	9	94	170	4	20	8
	Total	60	527	1352	20	106	43

Table 5 shows the subject programs and their descriptive statistics. We followed [6] and discarded any modified version if more than 20 percent of test cases could detect failures due to its fault. (The fault seeding procedure is similar to [9].) 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 [7][18]. 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 6 shows the maximum, average, and minimum sizes of the test suites.

TABLE 6. STATISTICS OF TEST SUITE SIZES.

Size \ Ref.	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–M8) presented in Section IV to prioritize the test cases. For every faulty version of the a subject program and every corresponding prioritized test suite, we executed the test cases one by one according to their order in the test suit, and collected the test results.

2) Effectiveness Measure

To compare the effectiveness (in terms of fault detection rates) among M1–M8, we use the *Average Percentage of Faults Detected* (*APFD*) as the metric [7], which 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 II (A).

B. Data Analysis

In this section, we analyze the data to evaluate the effectiveness of different techniques. We apply techniques M1–M8 on each application and calculate the value of *APFD*. We repeat this procedure 100 times using the generated test suites. The results are collected and summarized in the box-plots in Figure 7.

Each box-plot shows the 25th, 50th, and 75th percentiles of a technique. The result for each application is given in Figures 7(a)–(h). The overall 25th, 50th, and 75th percentiles as well as the overall mean of all subject programs are shown in Figures 7(1)–(4).

We first use the 25th percentile to compare M1–M8, followed by examining the 75th percentile. As shown in Figure 7(1), random prioritization (M1) achieves a mean *APFD* of 0.788. The maximum and minimum mean *APFD* achieved for the white-box techniques (M3–M6) are 0.817 and 0.839, respectively. On the other hand, the mean *APFD*

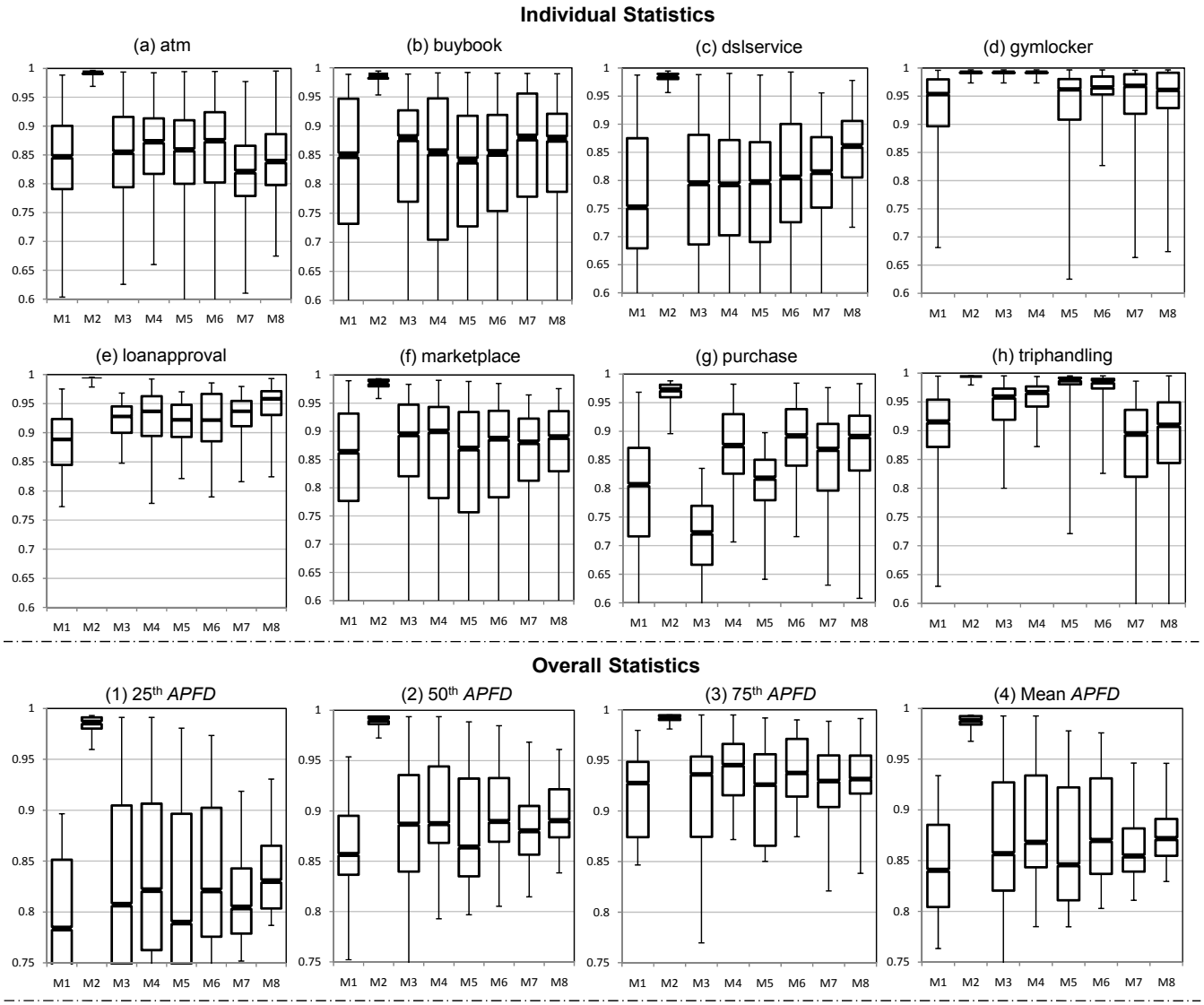


Figure 7. Comparisons of M1–M8 using *APFD* measures (x-axis is metrics; y-axis is APFD value).

for M8 is 0.844, which is higher than the corresponding values for M3–M6.

The mean *APFD* for M7 is 0.821, 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). When considering the 75th value in Figures 7(1)–(4), we observe that the performance of M7 and M8 is worse than that of M3 to M6. This observation is not too surprising because of the lack of knowledge of the detailed program code for M7 and M8.

Figure 7(4) shows that M8 is better than all other techniques (except the optimal) at both the 25th and 50th percentiles of *APFD* results. Moreover, M3–M6 are also better than M1 at the 75th percentile. Similar conclusions can be drawn from Figures 7(1)–(3).

We find that our technique M8 can even be better than M3–M6 in some benchmark applications (such as Figure 7(c) and (e)). Among all these subject applications, M7 and M8 are significantly better than M1 in five cases out of eight (Figure 7(b), (c), (e), (f), and (g)), close to M1 in two (Figure 7(d) and (h)), and only a little worse than M1 in one case (Figure 7(a)). This result indicates that M7 and M8 have a high probability of outperforming random ordering.

On average (as reported in Figure 7(4)), our black-box testing techniques M7 and M8 are close to white-box testing techniques (M3–M6) for the mean *APFD* and at the 25th, 50th, and 75th percentiles of *APFD* results.

In the experiment, M7 and M8 are more effective at early fault detection than random ordering. This indicates that black-box test case prioritization is a promising method for services testing.

One may wonder whether the differences between our WSDL-related techniques and conventional techniques are significant. To answer this question, we compare the differences among M1, M3–M6, and M7–M8 (which we call between-group comparisons), and the differences between M7 and M8 (within-group comparison). Since rejecting the null hypothesis can only tell whether the performance of M3–M8 is significantly different, we follow [12] to conduct a multiple-comparison procedure to further explore where the differences lie. Like [12], the Least Significant Difference (LSD) method [4][12] is employed in the multiple-comparison to compare techniques. We set the significance level to 0.05.

TABLE 7. MULTIPLE COMPARISONS USING *LSD*.

Category	Technique (x, y)	Sig. < 0.05		Sig. > 0.05	
		x–y > 0	x–y < 0	x–y > 0	x–y < 0
Between-Group	M1, M7	**2**	**5**	0	1
	M1, M8	**0**	**4**	2	2
	M3, M7	3	2	1	2
	M3, M8	2	3	2	1
	M4, M7	3	2	1	2
	M4, M8	3	2	1	2
	M5, M7	**2**	**5**	0	1
	M5, M8	**1**	**5**	1	1
	M6, M7	3	1	1	3
	M6, M8	2	4	2	0
Within-Group	M7, M8	**0**	**4**	2	2

Table 7 presents pairwise comparison results of all subject applications between groups (between M1 and M7–M8 and between M3–M6 and M7–M8), and within the group (M7–M8). We categorize the results of the comparisons into two classes (< 0.05 and > 0.05) in terms of the significance of the mean difference. Each value in Table 7 is the total number of applications in the groups.

In the between-group category, 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*. However, we do not observe a significant difference among the following pairs: (M3, M7), (M3, M8), (M4, M7), and (M4, M8) (considering that the difference in values between any two subgroups under "Sig. < 0.05" is no more than one). This indicates that the overall performance of M3 and M4 are close to that of M7 and M8. We further find that M7 and M8 are both better than M5 (2 vs. 5, and 1 vs. 5), and the results between M6 and M7–M8 are similar.

In the within-group category, we find that M8 is significantly better than M7 (0 vs. 4). This 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. This result also verifies the observation that we have discussed in Figures 7(1)–(4) above.

C. Threats to Validity

Threats to construct validity arise if measurement 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–M8. We choose the *APFD* metric, which has been widely adopted in many test case prioritization work (such as [6][7]), to measure the fault detection rate of prioritization techniques.

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 [14] to address this problem. In this paper, our experiment tool resets the runtime 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.

VI. CONCLUSION

In a service composition, a member service may constantly evolve. 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 interoperability 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 test case prioritization techniques.

This paper studies whether the use of WSDL information may facilitate effective regression testing of services. We have proposed to compute the WSDL tag coverage from the input and output messages associated with regression test suites. We have proposed two black-box test case prioritization techniques, and have used WS-BPEL programs in a controlled experiment to evaluate them. The empirical results show that our techniques can be effective. However, since WSDL is only an interface specification, it may be blind to certain regression fault types. For future work, we plan to enhance the proposed techniques by integrating with other black-box techniques.

REFERENCES

[1] C. Bartolini, A. Bertolino, E. Marchetti, and A. Polini. Towards automated WSDL-based testing of web services. In *Service-Oriented Computing* (*ICSOC 2008*), volume 5364 of Lecture Notes in Computer Science, pages 524–529. Springer, Berlin, Germany, 2008.

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

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

[4] S. G. Carmer and M. R. Swanson. Detection of differences between means: a Monte Carlo study of five pairwise multiple comparison procedures. *Agronomy Journal*, 63: 940–945, 1971.

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

[6] H. Do, G. Rothermel, and A. Kinneer. Empirical studies of test case prioritization in a JUnit testing environment. In *Proceedings of the 15th International Symposium on Software Reliability Engineering* (*ISSRE 2004*), pages 113–124. IEEE Computer Society Press, Los Alamitos, CA, 2004.

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

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

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

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

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

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

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

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

[15] L. Mei, W. K. Chan, and T. H. Tse. An adaptive service selection approach to service composition. In *Proceedings of the IEEE International Conference on Web Services* (*ICWS 2008*), pages 70–77. IEEE Computer Society Press, Los Alamitos, CA, 2008.

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

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

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

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

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

[21] G. Rothermel, S. G. Elbaum, A. Malishevsky, P. Kallakuri, and B. Davia. 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*), pages 130–140. ACM Press, New York, NY, 2002.

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

[23] W.-T. Tsai, Y. Chen, R. Paul, H. Huang, X. Zhou, and X. Wei. 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*), volume 1, pages 101–106. IEEE Computer Society Press, Los Alamitos, CA, 2005.

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

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

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

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