To appear in *Proceedings of the 18th International Conference on World Wide Web* (*WWW 2009*), ACM Press, New York, NY (2009)

Test Case Prioritization for Regression Testing of Service-Oriented Business Applications*†

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

Zhenyu Zhang
The University of Hong Kong
Pokfulam
Hong Kong
zyzhang@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

ABSTRACT

Regression testing assures the quality of modified service-oriented business applications against unintended changes. However, a typical regression test suite is large in size. Earlier execution of those test cases that may detect failures is attractive. Many existing prioritization techniques order test cases according to their respective coverage of program statements in a previous version of the application. On the other hand, industrial service-oriented business applications are typically written in orchestration languages such as WS-BPEL and integrated with workflow steps and web services via XPath and WSDL. Faults in these artifacts may cause the application to extract wrong data from messages, leading to failures in service compositions. Surprisingly, current regression testing research hardly considers these artifacts. We propose a multilevel coverage model to capture the business process, XPath, and WSDL from the perspective of regression testing. We develop a family of test case prioritization techniques atop the model. Empirical results show that our techniques can achieve significantly higher rates of fault detection than existing techniques.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—Testing tools; D.2.8 [**Software Engineering**]: Metrics—Product metrics

General Terms: Measurement, Reliability, Verification

Keywords

Test case prioritization, service orientation, XPath, WSDL

1. INTRODUCTION

Industrial leaders such as IBM, Microsoft, Oracle, and BEA promote the use of service-oriented business processes to build their enterprise applications. Process engineers may develop such applications using orchestration languages like the Web Service for Business Process Execution Language (WS-BPEL) [22] and Business Process Modeling Language (BPML). In a typical application, a business workflow (say, coded in BPEL) may use external web services to implement individual workflow steps. To transfer type-safe XML messages [22] among individual workflow steps and web services, process engineers write diverse specifications in Web Service Description Language (WSDL) [23] (dubbed *WSDL specifications*) to interpret different portions of the same or different XML documents for various workflow steps. Since a workflow step may use part of the content kept in an XML document, process engineers may define XPath expressions [25], which pairs with WSDL specifications, to extract the required contents from the document.

To cope with changing business requirements, process engineers may modify the service-oriented business process [11][14][26]. Testers should assure the quality of such revised applications. Regression testing, aimed at detecting potential faults caused by software changes, is the de facto approach [8][20]. It reruns test cases from existing test suites to ensure that no previously working function has failed as a result of the modification [8]. Although many researchers point out that frequent executions of regression test are crucial in successful application development [8][15], rerunning the regression test suite for large and complex systems may take days and even weeks, which is time-consuming. In service-oriented computing, a business process may invoke external web services (such as viewing an article in Economist.com), which may incur charges. To reduce costs, it is desirable to detect failures as soon as possible when executing the test suite. The use of effective regression testing techniques is, therefore, crucial.

Thus, test case prioritization [19] is important in regression testing [9][15]. It schedules the test cases in a regression test suite with a view to maximizing certain objectives (such as revealing faults earlier), which help reduce the time and cost required to maintain service-oriented business applications. Existing regression testing techniques for such applications focus on testing individual services [20] or workflow programs [6]. Surprisingly, to the best of our knowledge, the integration complexity raised by non-imperative artifacts such as XPath and WSDL among workflow steps has been inadequately addressed in regression testing research.

Let us consider a simple example. Suppose an application aims to implement an XPath query to select (from a list of available hotel rooms kept in an XML document) all "single rooms" priced less than $100. Suppose also that the XPath expression has been implemented erroneously as selecting *either* "single rooms" *or* rooms priced less

---

* This is a preprint of an Article accepted for publication in *Proceedings of the 18th International Conference on World Wide Web* (*WWW 2009*), ACM Press, New York, NY (2009) © 2009 International World Wide Web Conference Committee.

† This research is supported in part by GRF grants of the Research Grants Council of Hong Kong (project nos. 111107, 123207, 717308, and 717506).

‡ All correspondence should be addressed to Dr. W. K. Chan at Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Hong Kong. Tel: (+852) 2788 9684. Fax: (+852) 2788 8614. Email: wkchan@cs.cityu.edu.hk.

WWW'09, April 20–24, 2009, Madrid, Spain.
ACM 978-1-60558-487-4/09/04.

than $100. Using this incorrect XPath query, the application may select a "single room" priced at $100 or above. In general, an XPath query in a workflow step may introduce additional (*conceptual*) *branch decisions* (such as deciding whether a room can be selected), and thus affect the workflow logic.

Furthermore, different XML messages that conform to the same WSDL specification may contain different sets of XML elements (including tags and attribute names). We refer to an XML element defined by at least one XML schema in a WSDL specification as a *WSDL element*. Incorrectly defining a WSDL element or failing to provide a definition may result in an integration error.

Faults may reside on the non-imperative artifacts (such as XPath and WSDL) in a service-oriented business application. To the best of our knowledge, however, prioritization techniques to effectively find test cases to reveal such implementation problems earlier during maintenance has not been studied. This paper tackles the problem.

Following our previous work [12], we model an XPath query (in the presence of a WSDL specification) as an XPath Rewriting Graph (XRG). An XRG represents potential scenarios of content selections from XML messages. Each content selection scenario is captured as an XRG branch (see Section 2.2.2). We note that XRG branches for different XML messages that the XPath expression is querying on may be different. To account for the WSDL artifact, we say that a test case *t* has covered a WSDL element *e* if *t* contains an XML message *m* as input, or *t* causes the application to generate an XML message *m*, such that *m* has *e* as its entity tag. In a changing business application, every artifact (workflow, XPath, or WSDL) may be modified. As a result, fault(s) may be introduced to the artifacts. The use of workflow coverage data to prioritize test cases may be effective for detecting faults in the workflow program, such as wrong predicates. However, such prioritizations may be ineffective for handling faults in other artifacts. More examples will be given in Section 3.

We propose a multilevel coverage model to capture the coverage requirements of these artifacts. Level 1 covers only the workflow, level 2 covers both workflow and XPath, and level 3 covers workflow, XPath, and WSDL. Through the level-by-level use of coverage data for test cases, we propose a new family of test case prioritization techniques.

To handle multiple types of artifact in the family of test case prioritization techniques, we use two strategies. The first strategy is to treat different artifacts homogenously, which is akin to enlarging the coverage space from pure workflow-oriented coverage space to a space linked up to the coverage space of other artifact types. We call it a *summation* strategy. On the other hand, we appreciate that such a homogenous treatment of artifacts may not reflect the different roles of these artifacts in a service-oriented business application. For instance, from the perspective of process engineers who write such applications, a workflow program is more important than XPath expressions or WSDL specifications. Therefore, we propose another strategy called a *refinement* strategy. This strategy would refer to another type of artifact (such as WSDL) only if using the artifacts already referred to (such as workflow and XPath) cannot help a prioritization technique to select a test case.

We develop a family of techniques using the above model and strategies. With the inclusion of more artifacts, our techniques can intuitively be more effective in detecting faults residing across various artifacts. Our experiment further shows that the family of techniques is effective to reveal regression faults in modified programs, and the techniques at a higher level is generally more effective than those at a lower level.

The main contribution of this paper is threefold. (i) Through a multilevel coverage model, we propose a family of test case prioritization techniques that consider imperative and non-imperative artifacts (including workflow, XPath, and WSDL) in service-oriented business applications. (ii) We analyze the proposed prioritization techniques and present a hierarchy to capture their relations. To our best knowledge, this is the first logical hierarchy to relate test case prioritization techniques in the literature. (iii) We report an experimental study to verify the effectiveness of our proposal.

The rest of the paper is organized as follows: Section 2 gives the preliminaries. Section 3 shows a motivating example to discuss the challenges. Section 4 presents our prioritization techniques. Section 5 presents an experiment to validate our proposal, followed by discussions and related work in Sections 6 and 7, respectively. Finally, Section 8 concludes the paper.

2. PRELIMINARIES

2.1 Test Case Prioritization

Test case prioritization [5][19] is an important kind of regression testing technique [9][18]. With the information gained in the previous software evaluation, we may design techniques to run the test cases to achieve a certain goal in the regression testing. For example, proper test case prioritization techniques increase the fault detection rate of a test suite and the chance of executing test cases with higher rates of fault detection earlier [5]. We adapt the test case permutation problem from [19] as follows:

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

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

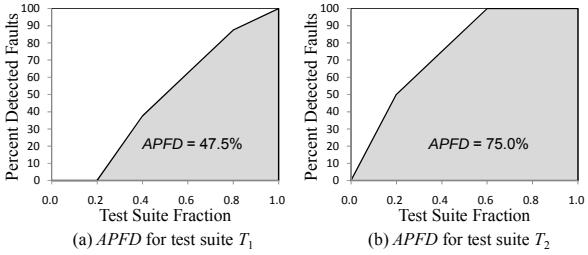
The metric of *Average Percentage of Faults Detected* (*APFD*) [5] is widely adopted in evaluating test case prioritization techniques [6][19]. A higher *APFD* value indicates faster (or better) fault detection rate [5]. Let T be a test suite containing n test cases, F be a set of m faults revealed by T, and TF_i be the first test case index in ordering T' of T that reveals fault i. The following equation gives the *APFD* value for ordering T' [5].

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

We provide an example to show how *APFD* measures the fault detection rate of different test suite ordering.

Example on test suite and faults exposed

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			•				•	



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

Figure 1. Example illustrating the *APFD* measure.

A program may have multiple faults. A test case sometimes can detect zero, one, or more faults, however, it can hardly find out all faults. Suppose the faults that test cases t_A to t_E can detect are shown in Figure 1. Let the two permutations for t_A to t_E be T_1 ⟨t_B, t_A, t_D, t_C, t_E⟩ and T_2 ⟨t_C, t_D, t_E, t_A, t_B⟩. The *APFD* measures on T_1 and T_2 are also given in Figure 1.

Other metrics [9] can also be used to measure these techniques. Owing to space limit, we will report such results in future work.

2.2 XPath and XPath Query Model

2.2.1 XPath

We adopt the definition of XPath expression in [13]. An *XPath expression* is defined using the following grammar:

$$q \rightarrow n \mid * \mid . \mid q / q \mid q // q \mid q[q]$$

The operators include the following: $n \in \Sigma$ is any label, * denotes a label wildcard, and . (the dot operator) denotes the current node. The constructions / and // mean child and descendant navigations, while the square brackets [ ] enclose a predicate. The symbols in Σ represent the element labels and attribute labels that can occur in XML documents. The set of all trees are denoted by T_Σ, and each tree represents an XML document satisfying the XML schema Ω. We also use Ω to represent the set of labels that can occur in the XML schema Ω. For a tree $t \in T_\Sigma$, an *XPath query* $q(t)$ is a query on t using an XPath expression q, and returns a set of nodes of t. $NODES(t)$ and $EDGES(t)$ denote the sets of nodes and edges, respectively. $LABEL(x)$ is the label on node x, $LABEL(x) \in \Sigma$. The transitive closure of $EDGES(t)$ is denoted by $EDGES^+(t)$, and the reflexive and transitive closure of $EDGES(t)$ is denoted by $EDGES^*(t)$.

Reference [13] gives the following definitions to represent a decidable fragment of XPath in Figure 2. According to [13], this fragment has provided representative XPath syntaxes and is sufficient to be the basis of studying XPath.

			Rule
n(x)	=	{y | (x, y)∈*EDGES*(t), *LABEL*(y) = n}	… 1
*(x)	=	{y | (x, y) ∈*EDGES*(t)}	… 2
.(x)	=	{x}	… 3
$(q_1/q_2)(x)$	=	{z | y∈q_1(x), z∈q_2(y)}	… 4
$(q_1//q_2)(x)$	=	{z|y∈q_1(x), (y, u)∈*EDGES**(t), z∈q_2(u)}	… 5
$(q_1[q_2])(x)$	=	{y | y∈q_1(x), q_2(y)≠Ø}	… 6

Figure 2. Syntax of a decidable fragment of XPath [13].

2.2.2 XPath query model

XPath queries are used to locate contents from an XML document. We have proposed in [12] an *XPath Rewriting Graph* (*XRG*) to represent an XPath with a document model Ω of XML documents. We revisit XRG here to facilitate the description of our techniques. An XRG is built on the definitions of XPath syntactic constructs [13]. We treat these definitions (Figure 2) as "left-to-right" rewriting rules, and through a series of rewriting [3], transform an XPath into a normal form or a fixed point. An XRG also records the intermediate rewriting steps, and links every two consecutive steps in the graph.

To capture the notion of rewriting [3], there are two types of node in an XRG, namely *rewriting node* $\langle q, L^c, rule\rangle$ and *rewritten node* $\langle q, L^c, L^n, S\rangle$. q is an XPath expression. L^c and L^n are sets of nodes (L^c, $L^n \subseteq NODES(\Omega)$). They represent the sets of tags relevant to q. L^c is the set of nodes located by the previous rewriting step. L^n is the set of nodes that can be located by q starting from some node in L^c. S is a set-theoretic representation of the result of q. Besides, *rule* denotes the rewriting rule used to generate the sub-terms in this node. Initially, L^c is assigned to $\{ROOT\}$, where *ROOT* is the unique root node of Ω.

Let us show an example of an XRG. Suppose, during the reservation of a hotel room (see the example in Section 3), the booking information (in XML format) is kept in a BPEL variable *HotelInformation*. Figure 3 shows a simplified XML schema *hotel* for *HotelInformation*. (We have omitted relevant details from the schema to ease the discussion of the example in Section 3.) A room has three attributes (lines 7–9): *roomno*, *price*, and *persons* (indicating the maximum number of persons allowed).

Consider an XPath query on *HotelInformation*, denoted by *XQ*(*HotelInformation*, *q*), where *q* is //room[@price≤'Price' and @persons='Num']/price/. Informally, *q* finds a room within the requested price that can accommodate the requested number of persons. The corresponding XRG is shown in Figure 4**.**

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

Figure 3. Part of WSDL document: XML schema of *hotel*.

We use the algorithm *Compute_XRG* from [12] to construct XRGs. We show the first rewriting step to illustrate how an XRG is computed. *XQ*(*HotelInformation*, *q*) is first identified by Rule 5 (q_1 = * and q_2 = room[precondition]/price/*), where precondition is "@price≤'Price' and @persons='Num'". Rewriting node R_1 is thus generated. Next, the algorithm recursively processes three sub-terms: //, q_1, and q_2. The *middle* sub-term // matches Rule 5 (note that // is the same as .//.), and so R_3 is generated. The *left* sub-term * matches Rule 2, and hence rewritten node R_2 is generated. The *right* sub-term q_2 matches Rule 4, and rewriting node R_4 is generated. The remaining rewriting steps are similar.

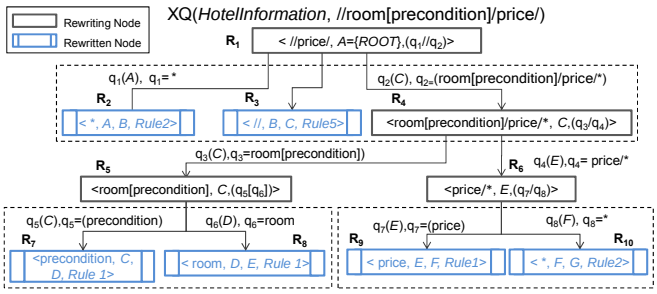


Figure 4. Example of *XPath Rewriting Graph* (*XRG*).

Following [12], we can obtain a *conceptual path* that models a logical computation of an XPath via an *inorder traversal* of the XRG with all the rewriting nodes dropped (as illustrated in Figure 8). Such a path contains *implicit predicates* that decide on the legitimate branch (called *XRG branch*) to be taken. For example, if no element in the XML document can be selected for the set B in R_2, B would be empty. This will result in no more applicable rewriting. A succeeding rewritten node will be appeared on a conceptual path only if its preceding rewritten node provides a non-empty set of L^c. Therefore, a branch can be modeled by whether L^c on a node is empty or not.

3. MOTIVATING EXAMPLE

We adapt the business process *HotelBooking* from the *TripHandling* project [21] to introduce the challenges in a typical service-oriented business application. *HotelBooking* offers the hotel booking service. Since showing the actual BPEL code in XML format is quite lengthy, we follow [12] to use an UML activity diagram to depict this business process to ease the illustration

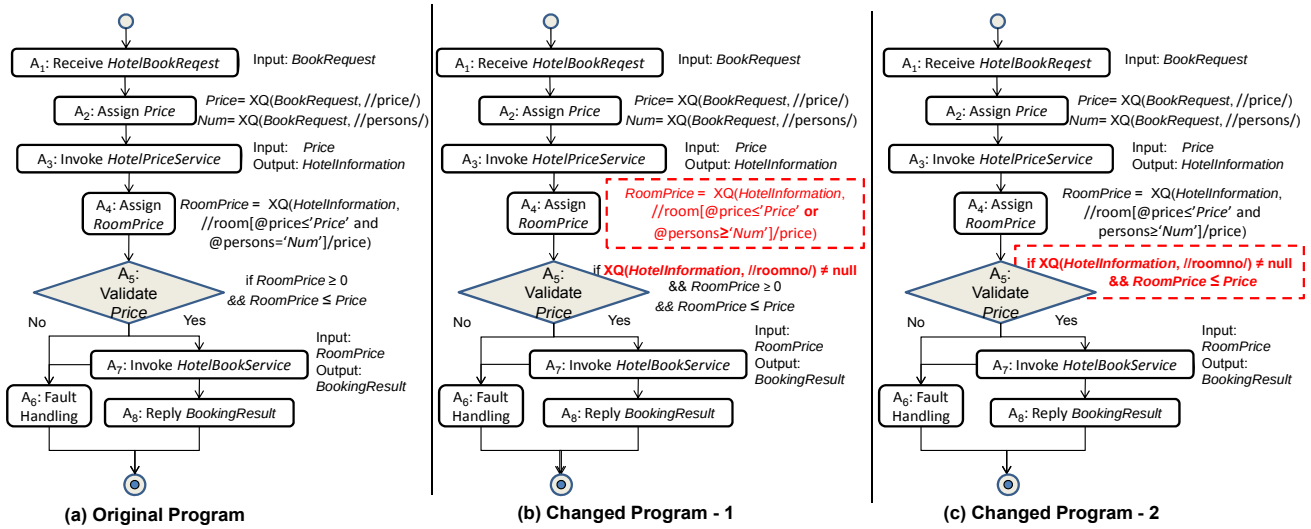


Figure 5. Activity diagram of a WS-BPEL application.

(Figure 5(a)). We also present two changes that may result in integration failures in Figures 5(b) and 5(c).

We use a node to represent a workflow node, and a link to represent a transition between two activities. We also annotate the nodes with information extracted from the program, such as the input-output parameters of the activities and XPath. The nodes are numbered as A_i (for i from 1 to 8) to ease the illustration. The process *HotelBooking* in Figure 5(a) is described as follows:

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

(b) A_2 extracts the inputted room price and number of persons via XPath //price/ and //persons/ from *BookRequest*, and stores them in the variables *Price* and *Num*, respectively.

(c) A_3 invokes the service *HotelPriceService* to find available hotel rooms with prices within budget (not exceeding *Price*), and keeps the result in *HotelInformation* (defined in Figure 3).

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

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

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

(g) If *RoomPrice* is erroneous or *HotelBookService* (A_7) produces a failure, A_6 will invoke a fault handler, i.e., $\langle A_7, A_6 \rangle$ is executed.

For ease of understanding, we summarize the artifacts and their relationships in UML class diagram notation (as shown in Figure 6). The description has been given in Section 1.

Suppose a process engineer Rick decides that the precondition at node A_4 in Figure 5(a) should be changed to that at node A_4 in Figure 5(b). (He attempts to allow customers to select any room that can provide accommodation for the requested number of persons.) However, he wrongly changes the precondition in the XPath (namely, changing "and" to "or"). While he intends to provide customers more choices, the process does not support his intention (for instance, the process is designed to immediately proceed to book rooms, rather than providing choices for customers to select). Further, suppose another engineer May wants to correct this fault. She plans to change node A_4 in Figure 5(b) back to that in Figure 5(a). However, she considers that the precondition at node A_5 is redundant (i.e., no need to require RoomPrice ≥ 0). Therefore, she changes the node A_5 in Figure 5(b) to become the node A_5 in Figure 5(c), and forgets to handle a potential scenario (Price < 0). Her change thus introduces a regression fault into the original program.

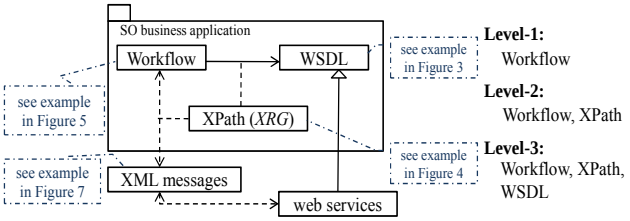


Figure 6. Key artifacts and their relationships in typical service-oriented business application.

We use a set of test cases (t_1 to t_6) to illustrate the challenges in test case prioritization. The inputs to WS-BPEL applications are XML documents. We simply use the price value of the variable *Price* to stand for the variable to save space. Due to page limit, the XML schema that defines *BookRequest* is not shown. Let us discuss A_4. Figure 7 shows the messages used at A_4 for t_1 to t_6.

When executing t_1 to t_6 on the program in Figure 5(b), t_1 extracts a right room price; t_4 to t_6 extract no price value; both t_2 and t_3 extract the price 105 of the single room, however, they actually need to book a double room and a family room, respectively. Observe that, t_2 and t_3 can detect the fault in Figure 5(b). Similarly, for the program in Figure 5(c), t_1 and t_2 can extract the right room prices; t_3 to t_5 extract no price value; t_6 extracts a room price –1, although it should not extract any price. Only t_6 can detect the fault in Figure 5(c).

Regression testing uses the coverage data achieved from previous execution round over a preceding version of the application to guide the current round of test case prioritization before executing these test cases on the modified application. Table 1 shows the workflow branch coverage of t_1 to t_6 on the original program *of HotelBooking* (i.e., Figure 5(a)). We use a "•" to represent the item covered by test cases in Figure 8 and Tables 1, 2, and 3.

We observe that the workflow branches covered by t_2, t_3, t_5, and t_6 are same (Table 1). A conventional branch-coverage prioritization technique may simply order them randomly, and thus ignore much useful information that potentially helps prioritize test cases to achieve a higher fault detection rate. Therefore, we introduce how XPath and WSDL can be used to address the challenges. Figure 8 shows the XRG nodes covered by t_1 to t_6 on the XRG (Figure 4) at node A_4 of Figure 5(a).

<Price, Num>	<Price, Num>
Test case 1 (t_1): <200, 1>	Test case 2 (t_2): <150, 2>
Test case 3 (t_3): <125, 3>	Test case 4 (t_4): <100, 2>
Test case 5 (t_5): < 50, 1>	Test case 6 (t_6): < –1, 1>

<hotel> <name>Hilton Hotel</name> <room> <roomno>R106</roomno> <price>105</Price> <persons>1<persons> </room> <room> <roomno>R101</roomno> <price>150</price> <persons>3<persons> </room> </hotel >	<hotel> <name>Hilton Hotel</name> <room> <roomno>R106</roomno> <price>105</Price> <persons>1<persons> </room> <room> <roomno>R101</roomno> <price>150</price> <persons>3<persons> </room> </hotel >	<hotel> <name>Hilton Hotel</name> <room> <roomno>R106</roomno> <price>105</Price> <persons>1<persons> </room> </hotel >
Test Case 1	Test Case 2	Test Case 3
<hotel> <room> <roomno></roomno> <price>100</Price> <persons>2<persons> </room> </hotel >	<hotel> </hotel >	<hotel> <room> <price>-1</Price> <persons>1<persons> </room> <error>InvalidPrice<error> </hotel >
Test Case 4	Test Case 5	Test Case 6

Figure 7. XML messages for *XQ(HotelInformation, //room[@price ≤ 'Price' and @persons = 'Num']/price/).*

Table 1. Workflow branch coverage for t_1 to t_6.

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

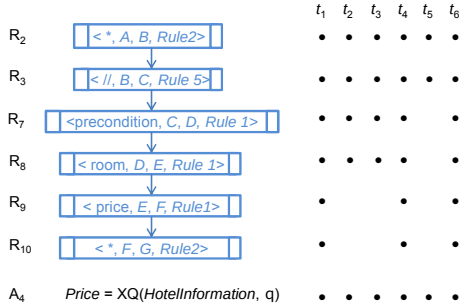


Figure 8. Example of XRG conceptual path.

Different XRG branches may lead to different content selections, and return different values to the workflow (see [12] for how to find out such a path). For example, the XRG branch of t_1 extracts the value 150 from the price tag and assigns the value to the variable *Price*. However, for t_2 and t_5, it will return no value (referred to as the null value for the ease of discussion) to *Price*. We further present Table 2 to show how t_1 to t_6 cover different XRG branches in the above XRG at node A_4 of Figure 5(a).

We observe that the XRG branches covered by t_1 and t_4 are identical. On the other hand, the branches covered by t_2 and t_3 are different from those covered by t_5 and t_6. Let the tuple $\langle top, bottom\rangle$ denote the theoretical highest (*top*) and lowest (*bottom*) priority orders of a test case determined by a potential prioritization technique. If we use the additional coverage data on the XRG branch, the tuples for both t_2 and t_3 are $\langle 2, 3\rangle$. However, using the additional branch coverage data, the tuples for both t_2 and t_3 are $\langle 2, 6\rangle$. This shows that the use of additional XRG branch coverage may increase the chance of achieving a higher fault detection rate.

Table 2. XRG branch coverage for t_1 to t_6.

XRG branch	t_1	t_2	t_3	t_4	t_5	t_6
$\langle R_2, R_3\rangle$	•	•	•	•	•	•
$\langle R_2, A_4\rangle$						
$\langle R_3, R_7\rangle$	•	•	•	•		•
$\langle R_3, A_4\rangle$					•	
$\langle R_7, R_8\rangle$	•	•	•	•		•
$\langle R_7, A_4\rangle$						
$\langle R_8, R_9\rangle$	•			•		•
$\langle R_8, A_4\rangle$		•	•			
$\langle R_9, R_{10}\rangle$	•			•		•
$\langle R_9, A_4\rangle$						
Total	5	4	4	5	2	5

Table 3. Statistics of WSDL elements for t_1 to t_6.

XML schema	t_1	t_2	t_3	t_4	t_5	t_6
hotel	•	•	•	•	•	•
name	•	•	•			
room	•	•	•	•		•
roomno	•	•	•	•		
price	•	•	•	•		•
persons	•	•	•	•		•
error						•
val(*name*)	•	•	•			
val(*roomno*)	•	•	•			
val(*price*)	•	•	•	•		•
val(*persons*)	•	•	•	•		•
val(*error*)						•
Total	10	10	10	7	1	8

We have shown that merely using the workflow branch coverage data may not reveal the internal conceptual branches and message types caused by the XPath and WSDL, and thus the performance of test case prioritization has not been fully maximized. This observation motivates us to present new techniques that take the XRG and WSDL coverage data into consideration.

4. OUR TEST CASE PRIORITIZATION

Given a test suite T for a service-oriented business application, our target is to reorder T according to the coverage data of the test cases in T when P is executed, with a view to effective regression testing of modified versions of P. In this section, we present a family of new test case prioritization techniques for such regression testing.

In view of the presence of heterogeneous artifacts, we propose a new coverage model to facilitate the development of our test case prioritization techniques. A *coverage model* for a service-oriented business application P is a 4-tuple $\langle T, \Gamma_\alpha, \Gamma_\beta, \Gamma_\gamma\rangle$, where (a) T is a regression test suite for P; (b) Γ_α, Γ_β, and Γ_γ represent, respectively, sets of workflow branches, sets of XRG branches, and sets of WSDL elements collected from various executions of P; and (c) $\Gamma_\alpha(t)$, $\Gamma_\beta(t)$, and $\Gamma_\gamma(t)$ represent, respectively, the set of workflow branches, the set of XRG branches, and the set of WSDL elements covered by the execution of P with respect to a test case $t \in T$.

We propose to utilize the coverage data of the test cases by levels. Level 1 covers only workflow, which is the basis of a business process. Next, since workflow may use XPath expressions to manipulate XML messages, level 2 covers both workflow and XPath. Finally, since XML messages must conform to the WSDL specification, level 3 covers workflow, XPath, and WSDL. For ease of presentation, we refer to the three levels of coverage data as **CM-1**, **CM-2**,

and **CM-3,** respectively, where CM stands for *Coverage Model*. Through the level-by-level use of coverage data, we propose a new family of test case prioritization techniques.

4.1 Our Prioritization Techniques

This section presents our test case prioritization techniques. If we considered a workflow program as a conventional program, the first two techniques (M1 and M2) would resemble to the branch coverage techniques of conventional programs [5][19]. Examples of each technique are shown in Table 4.

4.1.1 CM-1 (Level 1): Using Γ_α only.

M1 (Total-CM1) [19]**:** This technique sorts the test cases in *T* in descending order of the total number of workflow branches executed by each test case. If multiple test cases cover the same number of workflow branches, M1 orders them randomly.

M2 (Addtl-CM1) [19]**:** This technique iteratively selects a test case *t* that yields the greatest cumulative workflow branch coverage, and then removes the covered workflow branches, $\Gamma_\alpha(t)$, from all remaining test cases to indicate that those removed workflow branches have been covered by the selected test cases. Additional iterations will be conducted until all workflow branches have been covered by at least one selected test case. If multiple test cases cover the same number of workflow branches in the current round of selection, the technique selects one of them randomly. If no remaining test cases can further improve the cumulative workflow branch coverage, the technique resets the workflow branch covered of each remaining test case to its original value. It applies the above procedure until all test cases in *T* have been selected.

Let *m* and *n* be the test suite size and the maximum number of workflow branches covered by a test case *t*, respectively. Collecting the branch coverage of test cases will take $O(mn)$ time. Sorting test cases will take $O(m \log m)$ time. Therefore, M1 can be finished in $O(mn + m \log m)$ time and M2 can be finished in $O(m^2 n + m^2 \log m)$ time.

4.1.2 CM-2 (Level 2): Using Γ_α and Γ_β.

M3 (Total-CM2-Sum): This technique is the same as Total-CM1, except that it uses the total number of workflow and XRG branches covered by each test case, rather than only the total number of workflow branches as in Total-CM1. It treats workflow branches and XRG branches in the same way.

M4 (Addtl-CM2-Sum): This technique is the same as Addtl-CM1, except that it uses the set of workflow and XRG branches covered by each test case, rather than only the set of workflow branches as in Addtl-CM1. It also treats workflow branches and XRG branches in the same manner.

Another way to prioritize test cases is to reorder test cases using the number of workflow branches, and when encountering a tie, in which multiple test cases have the same number of workflow branches, a technique may use the number of XRG branches to break the tie.

M5 (Total-CM2-Refine): This technique is the same as Total-CM1 except that, if multiple test cases cover the same number of workflow branches, to break the tie, M5 orders them in descending order of the total number of XRG branches covered by each test case involved in the tie. If there is still a tie, M5 randomly orders the test cases involved.

M6 (Addtl-CM2-Refine): This technique is the same as Addtl-CM1 except three things. First, in each iteration, M6 removes the covered workflow branches and the XRG branches of the selected test cases from the remaining test cases to indicate that those removed workflow branches and XRG branches have been covered by the selected test cases (despite that M6 still selects test cases based on the workflow branch coverage as in Addtl-CM1). Second, if multiple test cases cover the same number of workflow branches in the current round of selection, rather than selecting one of them randomly, the technique selects the test case that has the maximum number of uncovered XRG branches. If there is still a tie, it randomly selects one of the test cases involved. Third, when resetting is needed, the technique resets each remaining test case to its original workflow branch coverage and XRG branch coverage.

Let *m*, *n*, and *x* be the test suite size, the maximum number of workflow branches, and XRG branches covered by a test case *t*, respectively. Collecting the branch coverage and XRG branch coverage of test cases will take $O(mn + mx)$ time. Sorting test cases will take $O(m \log m)$ time. Therefore, M3 can be finished in $O(mn + mx + m \log m)$ time and M4 can be finished in $O(m^2 n + m^2 x + m^2 \log m)$ time. The time complexity of M5 and M6 are the same with those of M3 and M4, respectively.

4.1.3 CM-3 (Level 3): Using Γ_α, Γ_β, and Γ_γ.

M7 (Total-CM3-Sum): This technique is the same as Total-CM2, except that it uses the total number of workflow branches, XRG branches, and WSDL elements covered by each test case, rather than only the total number of workflow and XRG branches as in Total-CM2. It treats workflow branches, XRG branches, and WSDL elements in the same way.

M8 (Addtl-CM3-Sum): This technique is the same as Addtl-CM2-Sum, except that it uses the set of workflow branches, XRG branches, WSDL elements covered by each test case, rather than just the set of workflow and XRG branches as in Addtl-CM2-Sum. It also treats workflow branches, XRG branches, and WSDL elements in the same fashion.

M9 (Total-CM3-Refine): This technique is the same as Total-CM2-Refine, except that in the case of a tie, M9 arranges the test cases in descending order of the total number of WSDL elements covered by each test case involved. If it still cannot resolve a tie, the technique randomly orders the test cases involved.

M10 (Addtl-CM3-Refine): This technique is the same as Addtl-CM1, except three things. First, in the each iteration, M10 removes the covered workflow branches, the covered XRG branches and the covered WSDL elements of the selected test cases from the remaining test cases to indicate that those removed workflow branches, XRG branches and WSDL elements have been covered by the selected test cases (despite that M10 still selects test cases based on the workflow branch coverage as in Addtl-CM1). Second, if multiple test cases cover the same number of workflow branches in the current round of selection, the technique selects the test case that has the maximum number of uncovered XRG branches. If there is a tie, it selects the test case that has the maximum number of uncovered WSDL elements. If there is still a tie, it randomly selects one of the test cases involved. Third, if resetting is needed, the technique resets each remaining test case to its original workflow branch coverage, XRG branch coverage, and WSDL element coverage.

Let *m* be the test suite size; *n*, *x*, and *w* be the maximum numbers of workflow branches, XRG branches, and WSDL elements covered by a test case *t*, respectively. Collecting the coverage data of workflow branches, XRG branches, and WSDL elements of *m* test cases takes $O(mn + mx + mw)$ time. Sorting *m* test cases takes $O(m \log m)$ time. Hence, M7 can be completed in $O(mn + mx + mw + m \log m)$ time while M8 can be completed in $O(m^2 n + m^2 x + m^2 w + m^2 \log m)$ time. The time complexities of M9 and M10 are the same as those of M7 and M8, respectively.

4.2 Benchmark Techniques

In Section 5, we will follow [5][19] and compare our test case prioritization techniques with two control techniques, namely *random* and *optimal*. For the sake of completeness, we revisit them in this section.

C1: Random prioritization [19]. This technique randomly orders the test cases in a test suite *T*.

C2: Optimal prioritization [19]. Given a program *P* and a set of known faults in *P*, if we know the specific test cases in a test suite *T* that expose specific faults in *P*, then we can determine an optimal ordering of the test cases to maximize the fault detection rate of *T*. Such a prioritization is an approximation to the optimal case [19].

4.3 Analysis of Prioritization Techniques

In total, we have reported 10 techniques. The acronyms of these techniques are listed in Table 4. We use the motivating example prioritization results on t_1–t_6 to help illustrate each technique.

Table 4. Categories of prioritization techniques and examples.

Category	Name	Index	t_1	t_2	t_3	t_4	t_5	t_6
CM-1 (workflow)	Total-CM1	M1	1	6	4	2	5	3
	Addtl-CM1	M2	1	5	6	3	2	4
CM-2 (summation)	Total-CM2-Sum	M3	2	4	5	1	6	3
	Addtl-CM2-Sum	M4	1	5	2	4	3	6
CM-2 (refinement)	Total-CM2-Refine	M5	2	4	5	1	6	3
	Addtl-CM2-Refine	M6	1	6	4	3	2	5
CM-3 (summation)	Total-CM3-Sum	M7	2	1	3	4	6	5
	Addtl-CM3-Sum	M8	1	5	3	6	4	2
CM-3 (refinement)	Total-CM3-Refine	M9	1	4	5	2	6	3
	Addtl-CM3-Refine	M10	1	4	2	3	6	5

Inspired by subsumption relations among the coverage criteria in unit testing, we propose a notion of subsumption for test case prioritization techniques.

Subsumption: Given two test case prioritization techniques *X* and *Y*, we say that *X subsumes Y* if and only if any permutation of any test suite produced by *Y* can also be produced by *X*.

Obviously, subsumption is reflexive, transitive, and anti-symmetric. It is, therefore, an equivalence relation. We have analyzed the subsumption relations among our techniques, and the result is summarized in Figure 9. For instance, we have proved that (M1) Total-CM1 subsumes (M5) Total-CM2-Refine, and we use an arrow from M1 to M5 to represent this relation in the figure. Other arrows can be interpreted similarly.

The proof of the subsumption relations is straightforward and hence we omit the details because of space limit. The basic idea is that, if random selection in resolving ties in one technique is replaced by a more deterministic procedure in another technique, then the former technique subsumes the latter. For instance, unlike M1 (which always use the random selection approach to resolve tie cases), M5 references the XRG branch coverage of test cases to resolve tie cases before using a random selection as the last resort. Because any test case that M5 can pick to resolve a tie may also be selected by chance in M1, any test case permutation produced by M5 must be a permutation that can be produced by M1. Other subsumption relations shown in Figure 9 can also be reasoned similarly.

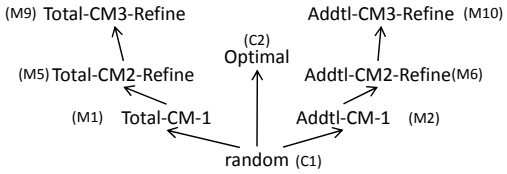


Figure 9. Hierarchy of test case prioritization techniques.

5. THE EXPERIMENT

5.1 Experimental Design

We choose WS-BPEL [22], a representative type of service-oriented business application [1][16][24], to evaluate our approach. The Software Engineering community also uses these applications to evaluate approaches related to service-oriented business applications (e.g., see [12]). We adopt the set of applications evaluated in [12] as our subject. Table 5 shows the descriptive statistics of the subject applications. For example, the size of each application is described using the number of XML elements ("Element") and the lines of code ("LOC").

Table 5. Subject programs and their descriptive statistics.

Ref.	Applications	Modified Versions	Element	LOC	XPath	XRG Branch	WSDL Element	Used Versions
A	atm [1]	8	94	180	3	12	12	5
B	buybook [16]	7	153	532	3	16	14	5
C	dslservice [24]	8	50	123	3	16	20	5
D	gymlocker [1]	7	23	52	2	8	8	5
E	loanapproval[1]	8	41	102	2	8	12	7
F	marketplace [1]	6	31	68	2	10	10	4
G	purchase [1]	7	41	125	2	8	10	4
H	triphandling [1]	9	94	170	6	36	20	8
	Total	60	527	1352	23	114	106	43

We use the known faults and associated test suites to measure the effectiveness of different prioritization techniques. The faults in the modified versions have been reported by [12] (in which faults are created following the methodology presented in [7]). We then follow [4][5][7] and discard any fault version if more than 20 percent of test

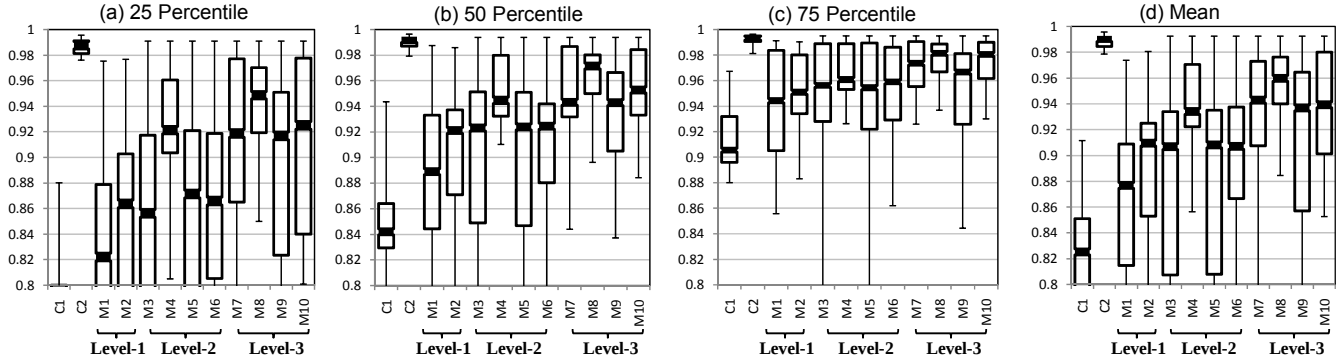


Figure 10. Overall comparisons using APFD measurement.

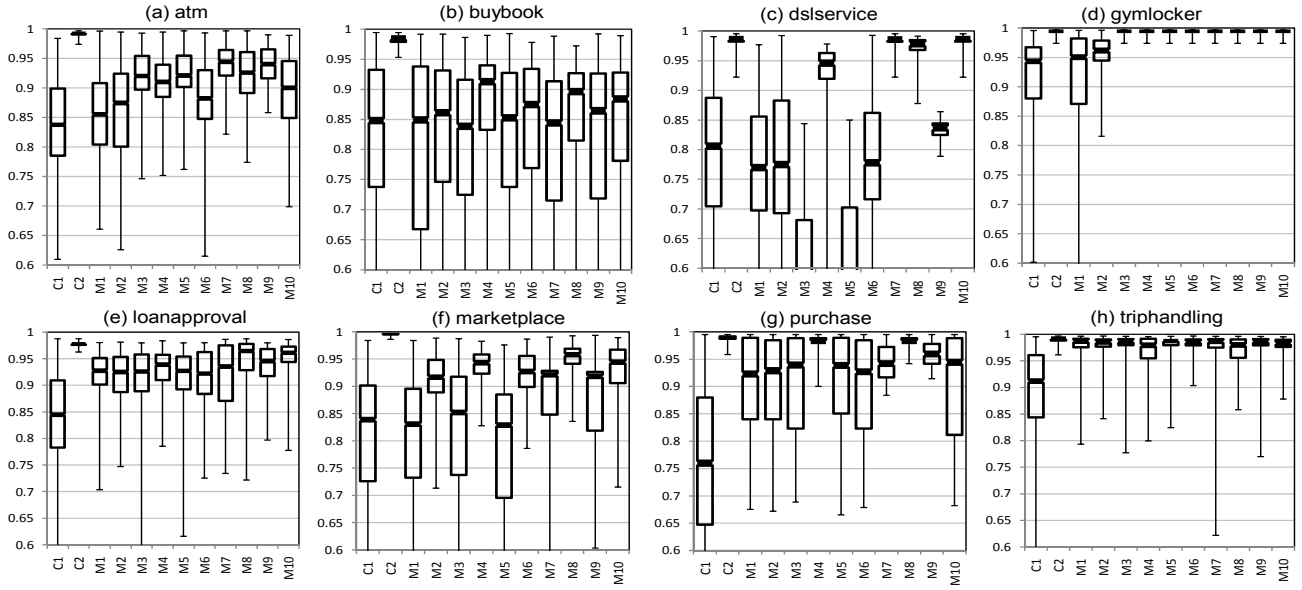


Figure 11. Comparisons on each application using APFD measurement (CM-3 techniques always outperform random).

cases can detect failures due to its fault. The statistics of the selected modified versions from [12] are shown in the rightmost column of Table 5.

We implement a tool to automatically generate test cases for each application. Based on WSDL specifications, XPath queries, and workflow logics of the application (not using modified versions), we generate test cases to ensure that the generated test cases can cover all workflow branches, XRG branches, and WSDL elements (dubbed CM-3 elements) at least once. In total, for each application, 1000 test cases are generated and formed up a test pool. This construction process is also adopted in [5][19].

Table 6. 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	140.6
Average	95	43	56	80	155	103	82	80	86.8
Minimum	29	12	16	19	50	30	19	27	25.3

We select test cases one by one randomly from a test pool and put them into a test suite (which is initially empty). Such selection is iteratively done until all the CM-3 elements have been covered at least once (and each fault has been detected by at least one selected test case). The process is similar to the test suite construction in [5][17]. We apply each test suite to applicable modified versions of corresponding application. In total, we successfully generate 100 test suites for each application. Table 6 shows the statistics of the test suites.

For each subject program and for each constructed test suite, our tool applies C1, C2, and M1–M10 to prioritize the test suite. For every prioritized test suite, the tool executes each modified version of the corresponding subject program over the test cases according to their order in the prioritized test suite. Since all the test case execution results on these applications are determined, we can figure out whether a fault has been revealed by a test case through comparing the test result on the modified version to that on the original program. Our tool automates the comparisons.

5.2 Data Analysis

5.2.1 Prioritization effectiveness

For each application, we apply C1, C2, and M1–M10 on a test suite, run the modified applications over the test suite, and then calculate *APFD* values. We repeat this procedure 100 times using the generated test suites. In total, 69,440 test cases have been executed, and we collect 833,280 *APFD* values. The results are represented using box-plots in Figures 10 and 11. A box-plot shows the 25th percentile, median, and 75th percentile of a technique in a graph. We summarize the overall results using the 25th percentile, median, 75th percentile, and mean *APFD* in Figure 10, respectively. The results for individual applications are given in Figure 11.

In Figure 10, we find that M6 and M7–M10 (i.e., one technique at level 2 and all techniques at level 3) are generally better than all the other techniques except the optimal technique (C2). When we focus on the techniques M1–M6, M2 and M4 are the best two techniques among the techniques in the same level. Both M3 and M5 are better than M1. M1 reports the worst performance among M1–M10 in this experiment.

The overall result may not represent the result of each benchmark application, and hence we further compare C2 and M1–M10 with the random technique (C1). If the *APFD* achieved at the 25th percentile of a technique is larger than, equal to or smaller than the random technique (C1), then we add 1 at the category "> Random", "= Random", and "< Random" of the technique, respectively. Similarly, we compare C2 and M1–M10 with C1 using the median and 75th percentile *APFD* values. Table 7 shows the comparison results.

Table 7. Comparisons with random technique.

Category / Technique		C2	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10
25%	**> Random**	8	5	7	6	8	5	8	7	8	7	8
	< Random	0	3	1	2	0	3	0	1	0	1	0
Median	**> Random**	8	6	7	6	8	6	7	8	8	8	8
	< Random	0	2	1	2	0	2	1	0	0	0	0
75%	**> Random**	8	6	7	7	8	6	7	8	8	7	8
	< Random	0	2	1	1	0	2	1	0	0	1	0

From Table 7, we note that C2, M4, M8, and M10 outperform the random technique (C1) in all categories. It is not surprising that C2 is better than C1, since C2 is an optimal approximation technique. C1 shows the worst performance generally. Among our techniques (M1–M10), M1, M3, and M5 show the worst performance when comparing to C1. This observation also holds when using the mean *APFD* values, as shown in Figure 10(d).

5.2.2 Hypothesis analysis

We further apply hypothesis analysis on the results to identify the differences among different techniques. We follow [9] to explore where the differences lie by using a multiple-comparison procedure. The *Least Significant Difference* (*LSD*) method was employed in multiple-comparison to compare test case prioritization techniques [9]. If the significance level is less than 0.05, the difference among the metrics is statistically significant.

We compare each pair of techniques for each application, and categorize the results into two groups (> 0.05 and < 0.05) using the significance of the mean difference. We do not show the cases when $x - y = 0$. The results are shown in Table 8.

We group M1–M10 into three groups according to the coverage model: M1–M2 (CM-1), M3–M6 (CM-2), and M7–M10 (CM-3). The between-group comparisons measure the differences between M1–M2 and M3–M6, between M1–M2 and M7–M10, and between {M4, M6} and M7–M10. The within-group comparisons measure the differences within M1–M2, M3–M6, and M7–M10. Due to page limit, we only report the results on comparing M4 and M6 in the group M3–M6 (CM-2) to compare with M7–M10. We choose M4 and M6 as the representative techniques for M3–M6 because these two techniques show a better performance in Figure 10. We mark the rows which indicate CM-2 techniques and CM-3 techniques are significantly better than CM-1 techniques into gray in background.

Table 8. Multiple comparisons (*least significance differences*).

Category		Techniques (*x*, *y*)	Sig. < 0.05		Sig. > 0.05	
			x–*y*>0	*x*–*y*<0	*x*–*y*>0	*x*–*y*<0
Between Group	M1–M2 vs. M3–M6	M1, M3	1	2	4	1
		M1, M4	0	6	1	1
		M1, M5	1	2	4	1
		M1, M6	0	3	2	3
		M2, M3	3	2	3	0
		M2, M4	0	6	1	1
		M2, M5	2	2	4	0
		M2, M6	0	3	1	4
	M1–M2 vs. M7–M10	M1, M7	0	5	3	0
		M1, M8	0	6	1	1
		M1, M9	1	5	0	2
		M1, M10	0	6	0	2
		M2, M7	2	4	1	1
		M2, M8	0	7	1	0
		M2, M9	2	5	1	0
		M2, M10	0	5	2	1
	M4, M6 vs. M7–M10	M4, M7	3	2	1	0
		M4, M8	0	2	0	5
		M4, M9	2	1	2	2
		M4, M10	1	2	3	1
		M6, M7	2	3	1	1
		M6, M8	0	6	1	0
		M6, M9	2	4	1	0
		M6, M10	0	2	1	4
Within Group	M1–M2	M1, M2	0	2	0	6
	M3–M6	M3, M4	0	4	2	1
		M3, M5	0	0	4	3
		M3, M6	1	3	1	2
		M4, M5	4	0	1	2
		M4, M6	4	0	2	1
		M5, M6	1	2	1	3
	M7–M10	M7, M8	1	4	2	0
		M7, M9	1	0	3	3
		M7, M10	2	3	0	2
		M8, M9	3	1	2	1
		M8, M10	3	0	1	3
		M9, M10	2	3	0	2

In the between-group category, M4 and M6–M10 all show significantly better results than both M1 and M2 (using the workflow coverage data). The difference between {M3, M5} and {M1, M2} are not significant. In the within-group category, we note that both M4 and M6 are significantly better than M3 and M5, which confirms our observation in Figure 10. The techniques within M7–M10 are similar in performance, and we only find the significant differences between M8 and {M7, M9, M10}.

5.2.3 Adequacy of coverage data

This section analyzes the impact of different levels of coverage data on the effectiveness of the technique. We use the overall mean *APFD* result of each technique in Figure 10(d). C1 and C2 report the worst and best mean result using the mean *APFD* in the box-plot of Figure 10(d).

Let us focus on the mean effectiveness of M1–M10. Using the mean *APFD* in Figure 10(d), the techniques using the *additional* coverage data are better those using the *total* coverage data. The pairs of techniques (M1, M2), (M3, M4), (M5, M6), (M7, M8), and (M9, M10) all demonstrate this conclusion.

We also check the subsumption relations (Figure 9) and overall effectiveness (Figure 10) for two groups of techniques: (M1, M5, M9) and (M2, M6, M10). The comparison result indicates a technique being subsumed may achieve a higher fault detection rate. For example, M5 is better than M1, and M9 is better than M5.

We observe that the mean effectiveness increases when more types of artifact have been considered in test case prioritization technique (i.e., as we include Γ_α, to Γ_α and Γ_β, and finally Γ_α, Γ_β, and Γ_γ). For example, when we categorize the techniques at Level 2 and Level 3 into pairs (M3, M7), (M4, M8), (M5, M9), and (M6, M10), the differences between two techniques in each pair support our observation. Similar observation can also be found in the between-group comparisons in Table 8.

5.3 Threats to Validity

The construct validity of our experiment relates to the metrics used to evaluate the effectiveness of test case prioritization. We use the metrics *APFD* in the experiment. Although normally knowing the faults exposed by a test case in advance is impractical, and hence an *APFD* value cannot be estimated before testing has been done. However, *APFD* can be used as a measure to show the feedback of prioritization techniques when testing has finished.

The external validity is whether the experiment can be generalized. We use WS-BPEL applications as subjects. They are a representative kind of service-oriented business application. Our experiments can be conducted using other service-oriented applications that use XPath queries and WSDL specifications. We will find more such applications to evaluate our techniques.

6. DISCUSSIONS

First, we use XRGs to model XPath queries in the presence of WSDL specifications. Other models to represent the XPath queries can also be used after defining coverage properly. However, the effectiveness of different XPath models may be different. In addition, our coverage model arranges the three artifacts in a particular order: ⟨workflow, XPath, WSDL⟩. It would be interesting to study the effectiveness of other potential orders (such as ⟨workflow, WSDL, XPath⟩), and compare them with our proposed techniques. Other such orders may result in different test case prioritization techniques. We plan to collaborate with the industry to apply our techniques in real-world projects and study the effectiveness of our presented techniques. We also plan to apply other statistical analyses of the results to gain more insights in the future.

Second, test case prioritization techniques can be categorized generally into two types [19]: *general test case prioritization* and *version-specific test case prioritization*. General test case prioritization reorders a test suite *T* for a program *P* to be useful in subsequent revised versions of *P*. Version-specific test case prioritization reorders test cases in a test suite *T* to be useful in a specific version *P'*

of *P*. Our work is under the category of general test case prioritization. It would be interesting to extend our techniques to version-specific test case prioritization.

7. RELATED WORK

This section reviews the related literature. In the context of test suite construction, Martin et al. [10] generated test cases based on WSDL specifications and treated them as requests for web services. Their technique perturbed the web requests, in the spirit of mutation testing, to test whether web services may robustly handle the perturbation. Their work discussed briefly the potential usage of the technique in regression testing of web services. Our previous work [2] applied metamorphic relations to construct test cases for stateless web services. Our previous work [12] proposed XPath Rewriting Graphs (XRGs) to represent conceptual paths (see Section 2.2). The XRGs help reveal the connection between WSDL and Workflow. It also proposed several unit testing criteria to exploit such connections to guide the construction of test suites. In this paper, we do not generate test suites but study the techniques to reorder existing test suites for regression testing.

Next, we review the research on test case selection for service-oriented applications. Ruth and Tu [20] proposed to conduct impact analysis to identify revised fragments of code in a service by comparing the control flow graph (CFG) of the new version with that of the preceding version. Their technique selected test cases associated with modified edges of the CFG. We study the test case prioritization problem in regression testing, rather than the test case selection problem. According to [15], these are distinct (but important) problems in regression testing research.

Hou et al. [6] proposed to add quota to constraint the number of requests to specific web services. They further developed techniques to prioritize test cases to maximize test requirement coverage under such quota constraints. Our work studies the internal organization of service-oriented business applications (representing the internal organization using a multilevel coverage model) and prioritizes test cases according to such properties.

8. CONCLUSION

Regression testing is the de facto means to assure the quality of a program against unintended effects of maintenance. Test case prioritization has been an effective means to order test cases in regression test suites so that faults can be detected earlier. When maintaining a service-oriented business application such as one written in WS-BPEL, process engineers may unintentionally introduce faults into various artifacts including the workflow programs, XPath queries, and WSDL specifications. Traditional test case prioritization techniques, which do not take all the artifacts into consideration, may no longer be effective for such an application.

In this paper, we have examined the important impact of considering these heterogeneous artifacts on test case prioritization in the regression testing of service-oriented business applications, and demonstrated the shortcomings of traditional test case prioritization techniques in this aspect. We have proposed a family of test case prioritization techniques that take into account the coverage data of test cases at three levels (workflow, XPath, and WSDL). We have further presented a hierarchy of subsumption relations among the test case prioritization techniques. To the best of our knowledge, this is the first hierarchy to relate test case prioritization techniques in the literature. The experiment results show that our techniques significantly outperform conventional test case prioritization techniques in terms of the fault detection rate (the most widely used metric for evaluating test case prioritization techniques in the software engineering community). Our experiment results also confirm that considering the artifacts level by level is an effective strategy in regression testing for assuring the quality of service-oriented business applications.

In the future, we will continue to study how to make use of non-imperative artifacts to develop effective techniques to address other challenges in the regression testing of service-oriented business applications. It would also be interesting to adapt our techniques to other service-oriented applications.

9. REFERENCES

[1] BPWS4J: a Platform for Creating and Executing BPEL4WS Processes, Version 2.1. IBM, 2002. Available at http://www.alphaworks.ibm.com/tech/bpws4j.

[2] 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.

[3] H. Y. Chen, T. H. Tse, and T. Y. Chen. TACCLE: a methodology for object-oriented software testing at the class and cluster levels. *ACM Transactions on Software Engineering and Methodology*, 10 (1): 56–109, 2001.

[4] 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. 2004.

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

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

[7] 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. 1994.

[8] 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. 1989.

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

[10] 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. 2007.

[11] 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. 2008.

[12] 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. 2008.

[13] G. Miklau and D. Suciu. Containment and equivalence for a fragment of XPath. *Journal of the ACM*, 51 (1): 2–45, 2004.

[14] O. Moser, F. Rosenberg, and S. Dustdar. Non-intrusive monitoring and service adaptation for WS-BPEL. In

Proceedings of the 17th International Conference on World Wide Web (*WWW 2008*), pages 815–824. 2008.

[15] 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.

[16] *Oracle BPEL Process Manager*. Oracle Technology Network. Available at http://www.oracle.com/technology/products/ias/bpel/. Last access February 8, 2009.

[17] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE TSE*, SE-11 (4): 367–375, 1985.

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

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

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

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

[22] *Web Services Business Process Execution Language Version 2.0*. OSAIS. Available at http://www.oasis-open.org/committees/wsbpel/. Last access February 8, 2009.

[23] *Web Services Description Language* (*WSDL*) *1.1*. W3C, 2001. Available at http://www.w3.org/TR/wsdl.

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

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

[26] C. Ye, S. C. Cheung, and W. K. Chan. Publishing and composition of atomicity-equivalent services for B2B collaboration. In *Proceedings of the 28th International Conference on Software Engineering* (*ICSE 2006*), pages 351–360. 2006.