Test Case Prioritization for Web Service Regression Testing

Lin Chen  Ziyuan Wang  Lei Xu  Hongmin Lu  Baowen Xu
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
Department of Computer Science and Technology, Nanjing University, Nanjing, China
Shanghai Key Laboratory of Computer Software Testing and Evaluating, China
{lchen, wangziyuan, bwxu}@nju.edu.cn

Abstract—Regression testing is necessary to assure the quality of service-oriented business applications in their evolutions. However, because of the constraint of testing resource, entire test suite may not run as a result. Therefore, test case prioritization technique is required to increase the efficiency of Web service application regression testing. In this paper, we propose a dependence analysis based test case prioritization technique. First, we analyze the dependence relationship using control and data flow information in an orchestration language: WS-BPEL. Then we construct a weighted graph and do impact analysis to identify modification-affected elements. After that, we prioritize test cases according to covering more modification-affected elements with the highest weight. Finally we conduct a case study to illustrate the applicability of our method.

Keywords-Web service; regression testing; test case prioritization; impact analysis; dependence analysis

I. INTRODUCTION

Service-oriented business applications have become popular in recent years. Service application evolution is a frequent activity. In a typical service-oriented application evolution, there are three participants: service provider, system integrator and service user. An evolution usually begins with a service requirement change. Then the integrator identifies the change and may update the service composition respectively. There are three typical evolution activities that may happen in a web service system: a service is updated to a new version; a service is replaced by a new one; or service composition is changed. In any of these evolution scenarios, an integrator or a user may need to retest it to build confidence. Web service regression testing plays a key role in these evolution scenarios. In many situations, entire retesting needs lots of resource and it is impossible to retest all, so test case prioritization technique is required to increase the efficiency of Web service regression testing.

The main issue we focus in Web service regression testing is how to prioritize the existing test cases to improve testing efficiency and quality. The traditional regression testing techniques take code coverage as the main criterion for test case prioritization, and select test cases by comparing the code structures [10-18]. These methods focus on structural programs. However, they do not consider that interactions between services and changed requirements may have different contribution to test case prioritization. We argue that when a service evolves, service interactions and changed impacts are two key points in service-oriented application regression testing. So we propose a test case prioritization model on which we are able to construct a weighted dependence propagation graph to model service interaction and to compute requirement change impacts. Based on this model, we discuss two test case prioritization strategies.

The rest of this paper is organized as the follows. Firstly, we introduce the background of test case prioritization techniques. Secondly, we analyze the dependence in Web Service interactions and discuss how to eliminate fake dependences. Then we do an impact analysis to identify modification-affected elements with different weight. After that, we prioritize test cases according to covering more modification-affected elements with the highest weight. Finally we give a case study to illustrate the applicability of our method.

II. BACKGROUND

Test prioritization technique aims to schedule test cases in an order that increase their effectiveness at meeting some given performance goal. S. Elbaum et al firstly defined the test prioritization problem formally as follows [14]:

Definition 1. For an initial test suite \( T_{init} \), \( PT \) is the set of permutations of \( T_{init} \). \( f \) is a function from \( PT \) to real numbers. Test prioritization aims to find a \( T \in PT \) such that:

\[
(\forall T') (T \in PT) (T \neq T') ([f(T)] \geq [f(T')])
\]

Here, \( PT \) is the set of all possible orderings of \( T_{init} \). And \( f \) is an objective function that, applied to any such ordering, yields an award value for that ordering.

For different application scenarios, numerous test prioritization techniques and empirical studies have been reported in recent years. W. E. Wong et al. firstly proposed a technique which combines modification, minimization and prioritization based selection using source code changes and test history [13]. G. Rothermel et al. proposed several test prioritization techniques based on source code and test history [11]. S. Elbaum et al. proposed a technique by incorporating varying test costs and fault severities [14]. A. Srivastava et al. proposed a technique based on basic block coverage for large systems [16]. J. M. Kim et al. proposed a technique based on test history [17]. Bo Qu et al. proposed a technique for black box testing scenario [12]. Xiaofang Zhang et al. proposed a technique for varying test
requirement priorities and test costs [18]. In order to prioritize test cases for such above scenarios, several algorithms have been proposed. Most of them could be divided into two classes: greedy methods and meta-heuristic search methods.

III. OVERVIEW OF OUR METHOD

We can see from the test case prioritization problem definition above that the key point in prioritization process is to find a good criterion to compute the mapping function \( f \). Most of the existing prioritization methods take code coverage as the criterion to compute the mapping function. However, since information inside a service is invisible to service users and only interfaces can be seen when invoking the service. So code coverage cannot be taken directly as a criterion for test case prioritization.

To model service evolutions, we use Web Service for Business Process Execution Language (WS-BPEL, BPEL for short) [2] to describe service composition. BPEL is an OASIS standard executable language for specifying interactions with Web Services. When a service evolves, for example, business requirements change, process engineers may modify the business process by BPEL. Since the service integrator cannot control the evolution of services, any changes to the service may have an impact on all the service users. These change impacts, and interactions between services, can be identified by analyzing dependence of the elements in BPEL process.

Our test case prioritization process can be done by the following steps:

1. Construct a BPEL flow graph (BPFG);
2. Analyze the dependence of BPEG;
3. Do change impact analysis to identify the modification-affected elements and calculate the weight of each element;
4. Prioritize test case according to covering more change impacts with the highest weight.

IV. WS-BPEL DEPENDENCE AND CHANGE IMPACT ANALYSIS

In this section, we construct a BPEL flow graph first, and then analyze the dependence information to do impact analysis.

A. WS-BPEL dependence analysis

A BPEL process defines a number of activities, including basic and structured activities. Basic activities are corresponding to atomic actions such as: invoke, receive, reply, assign, throw, exit, and empty. Structured activities include: sequence, flow, pick, while, scope. Structured activities can be nested and combined to build complex BPEL process. BPEL uses sequence, flow, switch, pick, while and control links to expressing control flow dependencies.

We adopt some existing works [3] to construct a BPEL flow graph. In a service-oriented application system, there may be several participants, including BPEL processes and partnerLinks. They can be combined together to build a complete system flow graph.

**Definition 2.** A BPEL flow graph (BPFG) is a direct graph \( G = < N, E, b, e > \) with node set \( N \) and edge set \( E \), \( b \) is the begin node and \( e \) is the end node. The activities, processes and partnerLinks are represented by nodes \( s_j \in N \) and edge \( e = \{(s_i, s_j) \mid s_i, s_j \in N \} \). \( s_j \) may be immediately executed after \( s_i \).

BPFG construction is similar to BFG construction [3]. We will not pay more attention on the detail of the BPFG construction but only discuss the differences between BPFG and BFG:

1. BPEL basic activities and partnerLinks are mapped to BPFG nodes. BPFG describes not only BPEL activities, but also interactions between BPEL processes and partnerLinks.

2. BPEL structural activities are mapped to BPFG nodes and edges. A pair of begin-end nodes according to each structural activity are added, and activities within the structural activity will be inclusion within the begin-end nodes. For structural activities, BFG does not need to transform multiple-choice to exclusive-choice. BFG is used to generate test cases to cover all possible paths, so the transformation makes it possible to model all possible execution scenarios.

BPFG is a CFG like graph to represent BPEL activity interactions. The computation of the dependence can be adopted from the existing works [4].Let \( s \) denotes an input event, output event or assignment, so it is a node of BPFG. We use the following notations to facilitate discussing dependence computation in BPFG:

- \( \text{Def}(s) \): returns the assigned variable;
- \( \text{Ref}(s) \): returns the variables on the right hand side of the assignment.

The formal semantic of the dependence in BPEL process is not discussed here. We only give some informal description. If \( \text{Def}(s_j) \) are used in the execution of \( s_i \), then \( s_i \) is data dependent on \( s_j \), denoted as \( \text{DEP}(s_i, s_j) \). If \( s_i \) is an immediate predecessor of \( s_j \), then \( s_i \) is control dependent on \( s_j \), denoted as \( \text{DEP}(s_i, s_j) \). In the rest of this paper, we do not distinguish the data dependence and control dependence.
Although the control flow constructs of BPEL have been designed carefully to avoid problems, composition of structured activities with control links can lead to fake dependence. Figure 1 shows an example.

In figure 1, the dependences denoted by dotted lines are fake dependences. In the example, a flow is defined, so there are two control flows executed in parallel. C is an unreachable activity by an and-join condition because A and B are defined in a pick block, and they are exclusive. So C cannot be dependent on A. Moreover, B is dependent on E, and E is dependent on A, so it seems that there is a dependent chain: B -> E -> A, however, B cannot be dependent on A because they are defined in a pick structure. So, similar to concurrent program dependence [9], dependence relationship in BPEL process is not transitive.

These fake dependences discussed above can be identified by two rules:

1. If the executions of the nodes are exclusive, then the nodes cannot have dependence relationship;
2. If a node \( s_i \) cannot be executed before node \( s_j \), then \( s_j \) cannot be dependent on \( s_i \).

These two rules can explain the fake dependences in figure 1.

Some formal methods, such as Petri-net or Automata, can be used to detect fake dependences. However, these techniques are time-consuming. In this paper, we pay our attention to solve these fake dependences with an efficient method.

**B. Change impact analysis**

When a service evolves, regression testing should retest the modified-affected parts in the system preferentially. Further more, we believe that the modified-affected elements may have different contributions to the regression testing. So we consider assigning a weight to each element to reflect the differences. Then the goal of our test case prioritization method is to cover elements with the highest weight.

A change impact analysis should be done to identify the modified-affected elements before test case prioritization. The details are described as Algorithm 1.

In Algorithm 1, we use two auxiliary functions:

\[
\text{Mutex}(s_j) = \{ s_j \mid \text{Mutex}(s_i, s_j) \} ; \\
\text{NotPre}(s_j) = \{ s_j \mid s_j \text{ cannot be executed before } s_i \cap s_i \text{ and } s_j \text{ are in the same sequence flow} \} .
\]

These auxiliary functions are derived from the rules discussed in the previous section. They can help to identify fake dependences.

The algorithm uses a broad-prior search strategy to compute the dependence set of the modified element. In the mean time, weight reflecting the impact degree of each element is propagated with the dependence. First, the modified element is assigned a highest weight which value is 1. For test case prioritization, it means that the modified element must be covered by test cases. Then the weight of a node is decreased with its distance to the modified element increasing. We use a simple auxiliary function \( (\alpha)^{\text{dist}} \) to calculate the weight: \( \text{dist} \) denotes distance between the node being calculated and the modified node; \( \alpha \) is a parameter which value is between 0 to 1.

All nodes being dependent on the modified node can be identified by algorithm 1. In the next section, two test case prioritization strategies are proposed, which criterion is to cover elements with the highest weight.

**Algorithm 1: Impact analysis of BPFG**

**Input:** A modified Node \( s \)  
**Output:** \( S \), the dependence set of \( s \)  

1. \( \text{begin} \)  
2. \( \text{// initialization} \)  
3. \( S = \{ s \}; Q = \{ s \}; W = \alpha; \)  
4. \( \text{weight}(s) = 1; \) //the weight of \( s \) is assigned 1  
5. \( \text{dist} = 0; \)  
6. \( \text{repeat} \)  
7. \( \text{repeat} \)  
8. \( \text{take an element } u \text{ from } Q; \)  
9. \( \text{weight}(u) = (\alpha)^{\text{dist}}; \)  
10. \( \text{if } (DEP(u), S, DEP(u), \text{Mutex}(u), DEP(u), \text{NotPre}(u)) \text{ then} \)  
11. \( S = S \cup DEP(u); \)  
12. \( W = W \cup DEP(u); \)  
13. \( \text{end if} \)  
14. \( \text{until } S \text{ is not changed;} \)  
15. \( \text{for element } u \text{ \{ \} do} \)  
16. \( \text{weight}(u) = (\alpha)^{\text{dist}}; \)  
17. \( \text{end for}; \)  
18. \( \text{end if}; \)  
19. \( \text{end repeat} \)  
20. \( \text{end repeat} \)  
21. \( \text{end} \)

**V. Test Case Prioritization**

Assume that in a service evolution scenario, some node in the BPEL, denoted as \( s \), was changed. Regression testing requires to cover the modified node \( u \) as soon as possible, to detect the faults that may be caused by such modification. Meanwhile, the nodes that dependent on the modified ones are also need to assign a higher priority to cover in regression testing. In this section, we will propose two different algorithms, which prioritize test cases in a given initial test suite \( T_{\text{init}} \) to get a prioritized test suite \( T \). And such operation will be according to the control flow coverage information.

**A. Total technique to prioritize test cases**

Firstly, we propose an algorithm to prioritize test cases according to the total contribution of each test case. It is clear that the test cases that cover more important nodes should
have a higher priority than others. To evaluate such priority, we define the total contribution for each test case:

\[
Total(test) = \sum_{u \in Cover(test)} \text{weight}(u)
\]

(1)

In this formula, \(Cover(test)\) denotes all the elements covered by the test case, and \(\text{weight}(u)\) denotes the weight of each element which was discussed in the previous. So, the formula (1) computes the total weight of all the elements that covered by test case \(test\).

After calculate it for all \(m\) test cases in initial test suite \(T_{init}\), we sort test cases according to their total contributions in descending order, and get a prioritized test suite \(T\). And for the test cases that have the same priority, we select randomly. Algorithm 2, which adopts the selection sorting algorithm as example, describes the details of such process.

Next, we analyze the time performance of proposed algorithm. Assuming that the weighted dependence graph has been already constructed, then the time performance of computing total contribution for one test case \(O(S)\), where \(S\) denotes the number of elements covered by test case. Besides, the average time complexity of selection sort algorithm is \(O(m^2)\). And if utilize quick sort algorithm, the average time complexity can be improved to \(O(m \times \log(m))\). Therefore, the time complexity of algorithm 2, including average time complexity and worst time complexity are \(O(m \times O(S) + m^2)\) and \(O(m \times O(S) + m^2)\) respectively.

**Algorithm 2. Prioritize Test Cases According to the Total Contribution of each test case**

**Input:** \(T_{init}\)

**Output:** Prioritized Test Suite \(T\)

1. Initialize \([T[1]...m]\)
2. Initialize \([Count[1]...m]\)
3. for \(i:=1\) to \(m\)
4. \([T[i]]:=T_{init}[i]; Count[i]:=Total(T_{init}[i])\)
5. end for
6. for \(i:=1\) to \(m\) //Selection sort according to total contribution
7. for \(k:=i\) to \(m\)
8. \(Max:=0; Best:=0\)
9. if \((Count[k]>Max)\) then
10. \(\{Max:=Count[k]; Best:=k\}\)
11. else
12. if \((Count[k]==Max)\ AND \text{random}[0,1]<0.5\) then
13. \(Best:=k\) //Select test case randomly
14. end if
15. end if
16. end for
17. if \((i\neq Best)\) then
18. Swap\((T[i], T[Best])\);
19. Swap\((Count[i], Count[Best])\)
20. end if
21. end for

**B. Additional technique to prioritize test cases**

There may be a limitation in first algorithm: the nodes that have been covered by previous test cases should be ignored when compute the priority of unselected test cases. So in each step of test selection, we define an additional contribution for each test case that has not been selected:

\[
Addtl(test) = \sum_{u \in \text{Cover}(test) \setminus \text{Cover}(pretest)} \text{weight}(u)
\]

This formula calculates additional contribution for each test case. When a test case was executed, the additional contribution of it is the total weight of elements which have not been covered by the previous test cases.

At each time, we calculate the additional contribution for all test cases that have not been selected into the prioritized test suite \(T\), and select one with the highest additional contribution. And for the test cases that have the same priority, we select randomly. This operation will repeat until all test cases in \(T_{init}\) have been selected into \(T\). Algorithm 3 describes the details of such process.

Next, we analyze the time performance of second proposed algorithm. When selecting the \(i\)-th test case for \(T\), we must calculate the additional contribution for \(m-i+1\) different test cases. It means that the total times of calculating is \(\sum_{i=1}^{m} (m-i+1)\). \(O(m^2)\) and \(O(m^2+m^2)\). And the time complexity of calculating additional contribution is \(O(S)\), where \(S\) is the number of elements covered by one additional test case. Therefore, we can conclude that the time complexity is \(O(m^2 \times O(S))\).

**Algorithm 3. Prioritize Test Cases According to the Additional Contribution of each test case**

**Input:** \(T_{init}\), Weights of Combinations, Costs of Test Cases

**Output:** Prioritized Test Suite \(T\)

1. Initialize \([T[1]...m]\)
2. for \(i:=1\) to \(m\)
3. \(Max:=0; Best:=0\)
4. for \(k:=1\) to \(m\)
5. if \((T_{init}[k]\notin T)\) then
6. \(Count:=Addtl(T_{init}[k])\)
7. if \((Count>Max)\) then
8. \(\{Max:=Count; Best:=k\}\)
9. else
10. if \((Count==Max)\ AND \text{tc}<tc_{Best}\) then
11. \(Best:=k\) //Select test case randomly
12. end if
13. end if
14. end if
15. end for
16. \([T[i]]:=T_{init}[Best]\)
17. end for

**VI. A CASE STUDY**

We adopt an ATM example from [12] to study the applicability of our method. To facilitate discussing, we
directly give the simplified BPFG and the dependence of it in figure 2, thus we can pay attention to the test case prioritization process. In this example, we consider only control dependence. Data dependence can be treated similarly.

When algorithm 3 is used to prioritize the test cases, we may get the test case sequence: \{t3, t4, t1, t2, t5\}. The prioritization process is shown in table 2.

![Simplified BPFG for ATM example of BPEL.](image)

Assume that there are 5 test cases, and their control flow coverage information is shown in table 1. Now considering the scenario that the withdraw activity is changed, and the corresponding dependence relationships are represented as dotted lines. So we can calculate the weight of each node in the graph. If the parameter \( \alpha \) used in algorithm 1 is assigned 0.1, then we have the weight of each node: \{0.01, 0.01, 0.01, 0.01, 1, 0.1, 0.1, 0.01\}.

When algorithm 2 is used to prioritize the test cases, we may have the test case sequence: \{t3, t4, t1, t2, t5\}. The test cases are ranked by the weight from high to low.

### Table 1. Control flow coverage and total weight of each test case.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.03</td>
</tr>
<tr>
<td>t2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.03</td>
</tr>
<tr>
<td>t3</td>
<td>*</td>
<td>*</td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.12</td>
</tr>
<tr>
<td>t4</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td>1.12</td>
</tr>
<tr>
<td>t5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.02</td>
</tr>
</tbody>
</table>

When algorithm 3 is used to prioritize the test cases, we may get the test case sequence: \{t3, t4, t1, t2, t5\}. The prioritization process is shown in table 2.

### Table 2. Additional technique for the example.

<table>
<thead>
<tr>
<th>Prioritization process</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>Highest additional weight</th>
<th>Selected test case</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.03</td>
<td>0.03</td>
<td>1.12</td>
<td>1.12</td>
<td>0.02</td>
<td>1.12</td>
<td>t3</td>
</tr>
<tr>
<td>2</td>
<td>0.01</td>
<td>0.01</td>
<td>-</td>
<td>0.1</td>
<td>0.01</td>
<td>0.1</td>
<td>t4</td>
</tr>
<tr>
<td>3</td>
<td>0.01</td>
<td>0.01</td>
<td>-</td>
<td>-</td>
<td>0.01</td>
<td>0.01</td>
<td>t1</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
<td>0.01</td>
<td>-</td>
<td>-</td>
<td>0.01</td>
<td>t2</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.01</td>
<td>t5</td>
</tr>
</tbody>
</table>

### VII. Conclusion

In this paper, we propose a dependence analysis based test case prioritization technique for Web Service regression testing. A weighted dependence propagation model is proposed to facilitate prioritization process. We discuss why fake dependence may happen in BPEL process and present how to eliminate them. Future works include formalize the dependence analysis and study more efficient algorithms. Also, more cases should be conducted to verify our work.

**ACKNOWLEDGMENT**

This work is supported by the National Natural Science Foundation of China (90818027, 60633010), the National High Technology Development Program of China (863 Program) (2009AA01Z147), and the Key Laboratory Funding of Science and Technology Commission of Shanghai Municipality (09DZ2272600).

**REFERENCES**


