Automatic Test Case Selection and Generation for Regression Testing of Composite Service Based on Extensible BPEL Flow Graph

Bixin Li, Dong Qiu, Shunhui Ji, Di Wang
School of Computer Science and Engineering, Southeast University
Nanjing, 210096, P.R.China.
Email: bx.li@seu.edu.cn

***Abstract*—Services are highly reusable, flexible and loosely coupled, which makes the evolution and the maintenance of composite services more complex. Evolution of BPEL composite service covers changes of processes, bindings and interfaces. In this paper, an approach is proposed to select and generate test cases during the evolution of BPEL composite service. The approach identifies the changes by using control-flow analysis technique and comparing the paths in new composite service version and the old one using extensible BPEL flow graph (or XBFG). Message flow is appended to the control flow so that XBFG can describe the behavior of composite service integrally. The binding and predicate constraint information added in XBFG elements can be used in path selection and test case generation. Theory analysis and case study both show that the approach is effective, and test cases coverage rate is high for the changes of processes, bindings and interfaces.**

I. Introduction

Service-oriented integration is a main application field of service computing, and the emergence of service composition technology makes the integration more convenient and efficient. Services are highly reusable, flexible and loosely coupled, which makes services computing more significant in distributed computing discipline. Having these characteristics, the evolution, together with the maintenance of composite services take on a new look. Regression testing of Web service is closely associated with its evolution and maintenance. Generally speaking, once any part of the composite service changes, regression testing must be performed.

Among most of the composition languages, BPEL is so popular that it becomes the de-facto standard on service composition[1]. It integrates available services by defining a business process, thus service composed using BPEL is a combination of process and component services. It is invoked the same way as a basic service through the exterior interface exposed to users, though the inner part is more complex and changeful. The evolution of composite service at design and implementation level, including *binding alteration* and *process alteration*, can be embodied in BPEL. BPEL combines *partner link* and the it endpoint reference mechanism from `WS-Addressing` to support service bindings[2]. *Partner link* establishes the relationships among component services interacting with BPEL process and prescribes the interfaces of process interacting with component services. The services which can meet both the interface definition and functional requirements will be regarded as candidate services. Which endpoint the process will bind to depends on the definition of `EndpointReference`. BPEL uses `<assign>` activity to copy endpoint reference to corresponding `<partnerLink>`. Above all, service binding in BPEL can be achieved by assignment.

This work is partially supported by the National Nature Science Foundation of China under No.60973149 and 60773105, and partially by National High Technology Research and Development Program under Grant No. 2008AA01Z113

BPEL composite service is composed of a process, an interface described in WSDL[3] and component services interacted with the process. The component services are either basic service or composite service, while the process specifies behaviors between them. Therefore the evolution of composite service may be caused by changes of process, interface and component services.

Service integrators may change the internal structure of process due to the need of functionality enhancement. The addition or deletion of services, change of activities or execution sequence all belong to *process change*. Once the process has been modified, some interfaces of composite service might be obsolete and some might have to be modified. It is *interface change* of composite service. These two cases are regarded as *active modification*, as the motivation is originated from service integrator. However, as component service is usually published by other vendors so they are out of control by the integrator. When integrators realize that the interface or functionality has been changed, they may make *passive modification* to adapt this change. For example, they may select another candidate service to replace the changed one. This is *binding change*. In conclusion, changes of BPEL composite service can be classified into three kinds, (1) *process change*, which means the BPEL modification, (2) *binding change*, which means the service integrator replaces a component service with another service having the same functionality and interface, (3) *interface change*, which usually means the WSDL modification.

Based on this classification, this paper proposes a solution for regression test case selection and generation of BPEL composite service using an extensible BPEL flow graph, which can express the behavior of composite service integrally. The binding information and predicate constraints are added in graph elements for path selection and test case generation. The comparisons are made on path elements, interfaces and path conditions so that the affected paths could be selected. Some paths can be validated by selecting test cases of baseline version but others may need new test cases. The decision of whether to select or to generate is made based on the results of several kinds of comparisons.

The rest of paper is organized as follows: Section 2 gives an introduction to XBFG; Section 3 discusses the test case selection and generation problem using XBFG in detail; Section 4 gives an evaluation of this approach; Section 5 discusses a case study; Section 6 compares the related work; Section 7 concludes the paper.

II. XBFG Model

A. Extended BPEL Flow Graph

BPEL Flow Graph (BFG) is a control flow model proposed by Y. Yuan et al. to describe BPEL process[4]. However, the interior implementation of BPEL composite service is the combination of process and component services interacting with the process. In order to operate change impact analysis on composite service rather than the process solely, we propose a model called XBFG so that the analysis could be operated on a model representing the whole composite service.

Formally, XBFG is defined as a triple $< N, E, F >$, where $N = IN \cup NN \cup SN \cup EN \cup MN \cup CN$, which denotes the node set; $E = CE \cup ME$, which denotes the edge set; F is the field of XBFG element, *element* is the general designation of node and edge. XBFG nodes are classified into six types:

Interaction Node (IN), which is mapped from those basic activities with which component services interact, including `<invoke>`, `<receive>`, `<reply>` and `<onMessage>` in `<pick>`.

Normal Node (NN), which is mapped from other basic activities of BPEL, such as `<assign>`, `<wait>` and so on. Additionally, `<onAlarm>` activity in `<pick>` is also mapped to NN.

Service Node (SN), which is mapped from the `partnerLinks` defined in BPEL, a SN represents a component service the process interacts with.

Exclusive Node (EN), which is mapped from those structural activities providing conditional behavior, including `<if>`, `<pick>`, `<while>` and `<repeatUntil>`. EN is divided into *Exclusive Decision Node (EDN)* and *Exclusive Merge Node (EMN).*

Multiple Node (MN), which is mapped from the `<link>` with its `<joinCondition>` value equals to ”OR” or null. MN is divided into *Multiple Branch Node (MBN)* and *Multiple Merge Node (MMN).*

Concurrent Node (CN), which is mapped from the `<flow>` activity and `<link>` with its `<joinCondition>` value equals to ”AND”. CN also has two forms, *Concurrent Branch Node (CBN)* and *Concurrent Merge Node (CMN).*

Figure 1 shows the symbols of each kind of XBFG elements.

The mapping from BPEL to BFG is limited to the process; XBFG needs to make appropriate extension so that not only the process, but also the component services as well as the message exchanges between them could be integrated in this model. Extensions are embodied in following aspects:

The first extension is that SN is added into BFG. In this way, component services are involved in the testing.

The second extension is about the control flow. BFG model uses edges to express control flows in BPEL. However, it can’t express how process interacts with component services through message exchanges. XBFG keeps the Control Edge (CE) linking the BPEL activities, and add Message Edge (ME) linking IN and SN to denote message calling relationship between process and its component services.

Field F is another extension to BFG with the purpose of facilitating the comparison of elements. Each the XBFG element has a *name*, *id*, *hashcode*, *source*, *target* and *category* field. The *name* field is copied from the *name* attribute of the correspondent activity. The *hashcode* is a multi-byte value generated for each BPEL activity when mapping BPEL to XBFG. The change of BPEL element could be detected by comparing the *hashcode* of elements in two XBFGs. Therefore, only those activities whose names, attributes and subelements all keep the same can be regarded as unchanged. The value of id field is a natural number generated according to the *hashcode* of this node. It is unique in XBFG so it can be regarded as the identity of the node. The reason for distinguishing nodes by id rather than *hashcode* is that same activities may exist in the same BPEL document and their *hashcodes* are identical. Besides, as for the edges which are not mapped from `<link>`, they have identical *id* only if they have the same *source* node and *target* node. The *source* and *target* field store set of precedent elements and subsequent elements of this node. The *category* field denotes the category of elements, such as IN, SN, CE, and so on. In addition, some categories of elements have their own fields. IN has EPR field, value of which is name of the `partnerLink` interacting with this IN. SN has *endpoint* field, value of which is physical address of this component service. EN, MN and CN use *condition* field to hold the transition condition (predicate constraint) of this node. CE also has *condition* field. If the edge is mapped from `<link>`, the value of condition field is the value of `<transitionCondition>` in `<link>`.

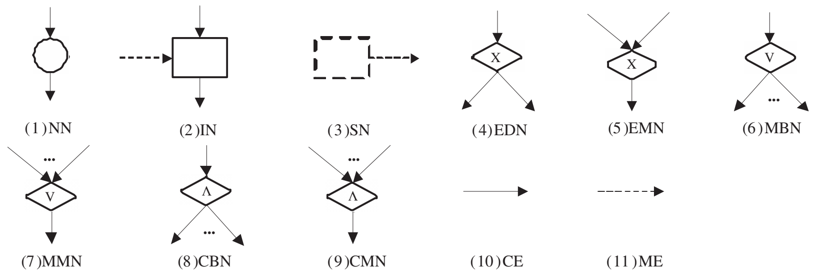


Figure 1. Symbols of XBFG elements

B. XBFG Path

Based on XBFG, the concept of path has to be customized and redefined accordingly. The BFG path definition is similar to the traditional CFG path, which is a sequential list of nodes. However, node sequence is not enough for change impact analysis of BPEL composite service. There are several reasons for this: firstly, condition of CE might be modified so that path should contain edges also; secondly, component services and messages exchanged between process should be considered as not only the binding of partners but the message order has the possibility to be changed; thirdly, it may encounter some difficulties when expressing paths using node sequence when BPEL has concurrent and synchronization structures.

As a matter of fact, in the view of flow graph, path can be interpreted as all the nodes and edges visited, together with their orders in one execution of program. Simple paths of the whole process could be obtained by analyzing the information carried by XBFG elements. This is because the decision and merge nodes are divided into several categories: AND, OR, NOR. Secondly, id of source nodes and target nodes are involved in the fields of edges. As a result, set of XBFG elements can be used to interpret simple path p. The execution order is hidden in the source and target fields of elements.

Path of XBFG is defined as an element set containing nodes and edges. The logical order of elements will be determined according to the source and target field of each element. Each path can be started from the initial node of XBFG. The initial node of path has two cases: the first case is path begins with an SN when the process starts from start activity `<receive>`, which means the process will be invoked by a component service through a message; the second case is path begins with an EDN when the process starts from the other start activity `<pick>`. The XBFG paths are defined as follows:

Control flow path is a set $p_f = \{e_i, ..., e_{i+k}, ..., e_{i+n}\}$ $(0 \leq k \leq n, e_{i+k} \in IN \cup NN \cup EN \cup MN \cup CN \cup CE)$, for $\forall e_{i+k}(e_{i+k} \in p_f)$, $\exists e_{i+p}(0 \leq p \leq n), e_{i+p} \in e_{i+k}.target.$

Message path is a set $p_m = \{e_j, ..., e_{j+k}, ..., e_{j+m}\}(0 \leq k \leq m, e_{j+k} \in SN \cup ME)$, p_m starts from a message edge e_m and ends at another message edge e'_m. For e_{j+k} ($e_{j+k} \in p_m \wedge e_{j+k}$ final message edge), $e_{j+q}(0 \leq q \leq m)$, $e_{j+q} \in e_{j+k}$.target.

A *path* of XBFG is composed of a control flow path p_f and several message path p_m, that is, $p = p_{f1} \cup p_{m1} ... \cup p_{mi} \cup ... p_{mm}$ $(1 \leq i \leq n$). For $p_{mi}(1 \leq i \leq n)$, $\exists in_m, in_m \in p_f \wedge in_m \in IN \wedge e_m \in p_{mi}(e_m \in in_m$.target $e_m \in in_m$.source).

The steps of generating XBFG and XBFG paths have been illuminated in our previous work[5]. Here we focus on the test case selection and generation problem.

III. Test Case Selection and Generation

The influence of the three types of evolution of BPEL composite service on testing can be analyzed by comparing different XBFG models. For example, there will be differences between elements of two XBFGs if the process changes. Again, if a binding change occurs on composite service, the value of endpoint field in the two service node must be different. In order to analyze the influence on the testing paths, we classify changes embodied on XBFG as four types:

Process change, which includes the modification of BPEL activities and order of activities.

Binding change, which means the modification of service endpoints.

Path condition change, which means change of path conditions caused by modification of process or predicate constraints.

Interface change, which includes modification of messages or variables defined in WSDL of composite service or component services.

Let $S_1, ..., S_i, ..., S_n$ denote versions of composite service, $\triangle_i$ denote the modification from S_i to S_{i+1}, that is, the increment of S_{i+1} relative to S_i, then $S_{i+1} = S_i + \triangle_i(1 \leq i \leq n-1)$.

Let $\triangle_{ip}$, $\triangle_{ib}$, $\triangle_{ic}$ and $\triangle_{ii}$ denote *process changes*, *binding changes*, *path condition changes* and *interface changes* of S_{i+1} relative to S_i, then $\triangle_i = \triangle_{ip} + \triangle_{ib} + \triangle_{ic} + \triangle_{ii}$. Let G_i and G_{i+1} denote the XBFGs mapped from S_i and S_{i+1}, P_i and P_{i+1} denote the path set of G_i and G_{i+1}, B_i and B_{i+1} denote the partner link set of G_i and G_{i+1}, C_i

and C_{i+1} denote the path condition set of G_i and G_{i+1}, I_i and I_{i+1} denote the interface set of G_i and G_{i+1}, then $\triangle_{ip} = P_{i+1} - P_i$, $\triangle_{ib} = B_{i+1} - B_i$, $\triangle_{ip} = C_{i+1} - C_i$, $\triangle_{ii} = I_{i+1} - I_i$.

Suppose P^s_{i+1} is the paths to be retested of S_{i+1}, where S_{i+1} is the composite service under test. Obviously, $P^s_{i+1} \subseteq P_{i+1}$. Let P^{sp}_{i+1}, P^{sb}_{i+1}, P^{sc}_{i+1} and P^{si}_{i+1} denote the XBFG paths of S_{i+1} influenced by process changes, binding changes, path condition changes and interface changes respectively, then $P^s_{i+1} = P^{sp}_{i+1} \cup P^{sb}_{i+1} \cup P^{sc}_{i+1} \cup P^{si}_{i+1}$

As all the four types of changes can be embodied in XBFG paths ultimately, so there exists a mapping φ, which makes $P^{sp}_{i+1} = \varphi(\triangle_{ip})$, $P^{sb}_{i+1} = \varphi(\triangle_{ib})$, $P^{sc}_{i+1} = \varphi(\triangle_{ic})$, $P^{si}_{i+1} = \varphi(\triangle_{ii})$.

Suppose T_i is the test case suite of S_i, for each test case t in T_i, there exists a mapping ψ, which makes $p = \psi(t)(t \in T_i, p \in P_i)$. That is to say, there is always a test case suite T_{ij} for each path p_{ij} in P_i, namely, $P_{ij} = \psi(T_{ij})(T_{ij} \subseteq T_i, p_{ij} \in P_i)$, and therefore, $T_{ij} = \psi^{-1}(T_{ij})(T_{ij} \subseteq T_i, p_{ij} \in P_i)$.

Again, there is always a test case suite $T^s_{(i+1)j}$ for each path $p^s_{(i+1)j}$ in P^s_{i+1}, namely, $T^s_{(i+1)j} = \psi^{-1}(p^s_{(i+1)j})(T^s_{(i+1)j} \subseteq T^s_{i+1}, p^s_{(i+1)j} \in P^s_{i+1})$. So test case suite selected is $T^s_{i+1} = \bigcup_{j=1}^{|P_{i+1}|} T^s_{(i+1)j}$

Where $|P_{i+1}|$ is the number of elements in P_{i+1}, i.e., the number of paths of G_{i+1}. Paths in P^s_{i+1} are partly from P_i of G_i, denoted as P^{so}_{i+1}, and partly from new paths of G_{i+1}, denoted as P^{sn}_{i+1}. So testing some of the paths in P^s_{i+1} can use test cases in T_i, and others need new test cases.

The steps for performing test case selection and generation on S_{i+1} against S_i are as follows.

(1) Path comparison (process and binding comparison): compare the paths in P_i and P_{i+1} one by one to get P^{sp}_{i+1} and P^{sb}_{i+1}. Some of the paths in P^{sp}_{i+1} are same as the paths in P_i, denoted as P^{spo}_{i+1}, others are new paths, denoted as P^{spn}_{i+1}. All the paths in P^{sb}_{i+1} are the same as paths in P_i. So move elements in either P^{spo}_{i+1} or P^{sb}_{i+1} into P^{so}_{i+1}, move elements of P^{spn}_{i+1} into P^{sn}_{i+1}.

(2) Interface comparison: compare the interfaces of each path in P^{so}_{i+1} with the corresponding path of P_i one by one. If some of the interfaces are found different, it indicates that new test cases are needed for these paths. So they need to be moved to P^{sn}_{i+1}. In succession, do comparison on the interfaces of each path in $\{P_{i+1} - P^{so}_{i+1} - P^{sn}_{i+1}\}$ and P_i to find out paths whose interfaces changed and move them to P^{sn}_{i+1}.

(3) Path condition comparison: generate path conditions for P_i and P^{sn}_{i+1} generated in step (1). Once it is found that certain path in P^{sn}_{i+1} has the same path condition as a path in P_i, move it to P^{so}_{i+1} because the two paths could use the same test cases.

(4) Match the test cases in T_i and the paths in P^{so}_{i+1} and add these test cases into T^s_{i+1}.

(5) Generate certain quantity of test cases for each path in P^{sn}_{i+1} according to interfaces and path conditions obtained by step (2) and step (3).

Suppose we are going to operate regression testing on S' against a baseline version S, G and G' are the XBFGs of S and S', P and P' are path set of G and G', T is the test case suite of S, P_s is the set of path selected for regression testing, which is comprised of two parts, one is the set of paths which can use the test cases of baseline version, named P_{s1}, and the other is the set of paths which need new test case, named P_{s2}. The following sections will explain how to do test case selection and generation on S and S'.

A. XBFG Path Comparison

The purpose of path comparison is to find out the path influenced by process and binding changes. If a path has binding changes and has no process changes, it should be moved to P_{s1}, and otherwise, it should be moved to P_{s2}.

As different XBFG elements have different ids, set of changed elements, including SN with different endpoints, can be obtained by comparing id of elements in set N (element set of G) and N'(element set of G'). The situations of changing can be divided into two categories: (1) if element $n \notin N \wedge n \in N'$, take n as a new element; (2) if element $n \in N \wedge n \notin N'$, take n as a deleted element. Create a set of new elements, named N_{add}, and a set of deleted elements, named N_{del}. For each element in N_{add}, search all the paths which contain this element and move them into P_{s2}. N_{del} will be dealt with as follows: for each p_i in P, let N_{di} denotes all the deleted elements in p_i, $p'_i = p_i - N_{di}$, if p'_i is in P', add p'_i into P_{s2}. Algorithm 1 describes the process of computing P_{s2}.

B. Service Interface Comparison

By comparing path elements, it could be decided which paths need new test cases, but we can't conclude the other paths can be tested using test cases of baseline version, as in some cases, input variables may be different. Actually, not only the external interface of BPEL composite service, but also the internal interfaces between process and component services need to be compared. The previous is usually specified in WSDL of composite service and the latter is specified in WSDL of component services, both of which are visible to the service integrator.

BPEL makes message exchanges with component services and client applications using `<invoke>`, `<receive>`, `<reply>` and `<onMessage>` activities. In these activities, the interface information, such as `partnerLink`, `operation` and `portType`, is recorded in the attributes nested in activities. As a result, variable definition in WSDL can be located by analyzing attribute information carried in XBFG elements.

In WSDL specification, *message* is defined as an entity encapsulating variables to be exchanged between Web services. Message is a collection of parts, each of which is

```
input  : P, P': Two path sets to be compared
         N, N': Element sets of P and P'
output: $P_{s1}$: Path of old version to be retested
         $P_{s2}$: Path of new version to be retested
1  PathComparison(P, P', N, N'): $P_{s1}, P_{s2}$;
2  $N_{all} = N \cup N'$;
3  for each element of $N_{all}$ : $n$ do
4      if $n \notin N \wedge n \in N'$ then
5          $N_{add} = N_{add} \cup \{n\}$
6      end
7      else if $n \in N \wedge n \notin N'$ then
8          $N_{del} = N_{del} \cup \{n\}$
9      end
10     for each element of $N_{all}$ : $n_{add}$ do
11         for each element of $P'$ : $p'$ do
12             if $n_{add} \in p'$ then
13                 $P_{s2} = P_{s2} \cup \{p'\}$
14             end
15         end
16     end
17     for each element of $P_{s2}$ : $p_{s2}$ do
18         for each element of $P_{s2}$ : $n$ do
19             if $n \in N_{add} \&\& (n.category == SN || n.category == ME)$ then
20                 $P_{s1} = P_{s2} \cup \{p_{s2}\}$
21             end
22         end
23     end
24     for each element of $P$ : $p_i$ do
25         for each element of $N_{del}$ : $n_{del}$ do
26             if $n_{del} \in p_i$ then
27                 $N_{di} = N_{di} \cup \{n_{del}\}$
28             end
29         end
30         if $p_i' = (p_i - N_{del}) \in P'$ then
31             $P_{s2} = P_{s2} \cup \{p_i'\}$
32         end
33     end
34     return $P_{s1}, P_{s2}$;
35 end
```

Algorithm 1: path comparison algorithm

comprised of a *name* and a *data type*. The data type of part adopts the XSD (XML Schema Definition) as the standard, which contains embedded standard data type, such as *string*, *Boolean*, *decimal* and so on, and complex data type will be defined by users. So interface comparison is actually a process of comparison on variables and messages. As INs of XBFG contain `operation` and `portType` field, the interface comparison could start from these INs and draw to the end when affected paths are found out.

Suppose p is a path of G, p' is a path of G'. Consider path pair p and p', both of them have the same elements. Let S_{IN} denotes sequential list of INs in p and $S_{IN'}$ denote sequential list of INs in p', obviously, $S_{IN} = S_{IN'}$. Firstly, a list of WSDL messages are found out by analyzing each element in S_{IN} for p and p' respectively. Suppose p has a sequential list of messages $M = M_1 M_2 ... M_i ... M_n$, p' has another message list $M' = M_1' M_2' ... M_i' ... M_n'$. The next step is to compare message definitions one by one, that is, to check if definition of M_i is identical to that of M_i'. According to WSDL schema, a message is usually composed by one or more parts, each of which is an element of `basicType` or `ComplexType`. So the items to be compared conclude not only the message definitions, but also the definitions of part and element. This checking is restrict because even if one type of element is different from the other, it will draw to the conclusion that M is not equal to M', testers need to move this path to P_{s2} and generate new test cases or modify the old ones when $M! = M'$. Algorithm 2 describes the details of interface comparison, where the method `findMsg(n)` is the function of finding message name according to the given $n \in IN$.

```
input  : p, p': Two paths to be compared based on interfaces
output: $result$: Comparison result of $p$ and $p'$
1  InterfaceComparison(p, p');
2  for each element of $p$ : $n$ do
3      if $p.category == SN$ then
4          $S_{IN} = S_{IN} \cup \{n\}$;
5          $m = findMsg(n)$;
6          $M = M \cup \{m\}$
7      end
8  end
9  for each element of $p'$ : $n'$ do
10     if $p'.category == SN$ then
11         $S_{IN}' = S_{IN}' \cup \{n'\}$;
12         $m' = findMsg(n')$;
13         $M' = M' \cup \{m'\}$
14     end
15 end
16 for each message pair$(m, m')$ do
17     if $m == m'$ then
18         compare each part of $m$ with that of $m'$
19     end
20 end
```

Algorithm 2: interface comparison algorithm

C. Path Condition Comparison

After path and interface comparison, paths to be retested have been divided into two parts: one is the paths with no need of generating new test cases, and the other is those paths which need new cases. In order to make use of test cases of baseline versions in a maximum and avoid redundant test case generation, we adopt the principle of predicate logic and compare path conditions of two versions. If they are proved to be identical, test cases corresponding to the path in baseline version will be used in regression testing.

BPEL program is in fact a structured program. However, compared to general control flow, BPEL flow is more complex because some new mechanism, such as Control Dependencies and Dead Path Elimination, are introduced to BPEL. The predicate constraint occurs not only in branch nodes, but also in merge nodes.

The *condition* fields record predicate constraint (or *prc*) of XBFG elements. Predicate constraint is composed of expression and operand. In BPEL, expression may be variable or function. Let E denote the expression, *op* is the operand set and *op* $\in \{=, >, >=, <, <=, ! =, !\}$. Generally speaking, there are three kinds of predicate constraint in XML Schema:

(1) *Boolean prc*. Its general format is *op* E_1, where E_1 is an expression, the value of which is a Boolean datum, *op* $\in \{!\}$.

(2) *Numeric prc*. Its general format is E_1 *op* E_2, where E_1 and E_2 are two expressions, the values of which are

numeric data, $op \in \{=, >, >=, <, <=, !=\}$.

(3) *String prc*. Its general format is E_1 *op* E_2, where E_1 and E_2 are two expressions, the values of which are string data, $op \in \{=, !=\}$.

Due to the classification of XBFG elements, EDN, EMN, MBN, MMN, CBN, CMN and CE may have *condition* field. But predicate constraints only exist in CEs whose sources are branch nodes and in those whose targets are merge nodes. So they are divided into branch predicate and merge predicate.

We define *branch predicate* as a condition expression attached with control edges whose source is branch node. As we mentioned in section 2, the branch predicate are fetched from `<condition>` sub element nested in `<if>`, `<while>` and `<repeatUntil>`. Usually, the predicate is Boolean, Numeric or String type.

Merge predicate is defined as a condition expression attached with XBFG merge nodes. BPEL designed `<joinCondition>` for synchronization of several activities by evaluating link status. The link has three statuses: true, false and unset. This kind of predicate is of the Boolean type. The expression of this predicate is name of link, which could be regarded as a variable.

Predicate constraint (*prc*) is defined as a triple *prc=<EP, TP, F>*, where EP is the constraint expression, TP is predicate type and *TP={Boolean, Numeric, String}*, F denotes how will prc combined in path condition, $F = \{AND, OR\}$.

Suppose prc and prc' are two predicate constraints, $prc = prc'$ iff $prc.EP == prc'.EP$ && $prc.TP == prc'.TP$ && $prc.F == prc'.F$. That is to say, two predicate constraints are identical only if their constraint expressions, types and conjunction are all same as each other.

Path condition (*pac*) is a vector containing predicate constraints of this path. The way of getting path condition is to gathering all the predicate constraints in the path before combining them together. As predicate constraints reside in CEs, path condition can be obtained by traversing CEs in the path. Let pac denote the path condition of p, e denote CEs in p, then

$$p.pac = \cup\{e.prc | e \in p\}$$

Two path conditions are identical if and only if for each prc' in pac', we can find one prc in pac, which makes $prc = prc'$. Algorithm 3 describes comparison of path conditions.

D. New Test Case Generation

New test cases should be generated for each path in P_{s2} after comparison of path, interface and path condition. Test case is composed of *input variables*, *value of input variables* and *path elements*. Based on the analysis of interface and path condition in section 3.2 and 3.3, it is clear that the input variables exist in the interface documents; also, the path condition can be worked out by Algorithm 3. In general,

```
input : $p, p'$: Paths to be compared based on path condition
output: $result$: Comparison result of $p$ and $p'$
PCComparison(p, p'): $result$
if p.pac.size !=p'.pac.size then
    return $false$
end
for each prc in $p.pac.prc_i$ do
    $prc_i.isMatch = false$
end
for each prc in $p'.pac.prc'_i$ do
    $prc'_i.isMatch = false$
end
for each prc in $p.pac.prc_i$ do
    for each prc in $p'.pac.prc'_i$ do
        if $prc_i.isMatch == false$ && $prc_i.EP == prc'_i.EP$ &&
        $prc_i.TP == prc'_i.TP$ && $prc_i.F == prc'_i.F$ then
            $prc'_i.isMatch = prc_i.isMatch = true$;
        end
    end
end
for each prc in $p'.pac.prc'_i$ do
    if $prc'_i.isMatch == false$ then
        result=false
    end
    result=true;
    return result;
end
```

Algorithm 3: path condition comparison algorithm

new test cases could be generated using XBFG. Let S_{IN} denote the sequence of INs in path p. Definition of input messages and related variables can be found in WSDL or xsd file according to the operation field and `portType` field of INs in S_{IN}. Variables defined in input messages are just the input variable of test cases. Test cases are divided by operations. If the variable definitions and value ranges of an operation in new version are the same as those of another operation in the baseline version, the test cases of the operation in the baseline version could be used in regression testing. Otherwise, new test cases are needed.

Having fetched the input variables during service interface comparison, we have formed a skeleton of test case actually. The test input depends on the strategy adopted. For example, functional testing needs only the messages from client application, but integration testing probably needs messages from all partner services, as well as client. Value of these input variables is another issue to be considered. As the path condition has been figured out, the range of value of these variables could be determined. Indeed, some variables of the path condition are internal variables of BPEL, but they can be deduced from the interface variables using the way of symbolic execution. That is, by replacing the internal variables with input variables, the predicate constraint is an expression only about input variables. In this way, range of value can be worked out using constraint solving.

IV. Experimental Evaluation

A. Evaluation Mechanism

A test cases is modification-traversing for P and P' if and only if it execute new or modified code in P or formerly executed code that has been deleted from P'[6]. Our method could cover all of the new and modified activities and part of the deleted activities in BPEL, as well as part of the new, modified and deleted elements in WSDL. Instead of traversing the WSDL document line by line, our method only retrieves interfaces interacting with BPEL process. And those modifications that are irrespective with the composite service in WSDL have not been covered. Although it will decrease the inclusiveness but it can promote the precision of this test selection technique because those modifications can't reveal the possible faults of composite service.

The modification of process, bindings and path conditions occurs in BPEL and change of interfaces occurs in WSDL. BPEL is composed by activities, WSDL is composed by elements and both of them are XML format. Suppose the actual numbers of changes of BPEL activities, bindings, path conditions and interfaces are denoted as $\mid \triangle_{ip} \mid$, $\mid \triangle_{ib} \mid$, $\mid \triangle_{ic} \mid$,and $\mid \triangle_{ii} \mid$ respectively, the covered numbers of these four items are m_{ip}, m_{ib}, m_{ic} and m_{ii}, then the coverage of process changes is $\rho_{ip} = \frac{m_{ip}}{\triangle_{ip}} \times 100\%$, the coverage of bindings changes is $\rho_{ib} = \frac{m_{ib}}{\triangle_{ib}} \times 100\%$, the coverage of path condition changes is $\rho_{ic} = \frac{m_{ic}}{\triangle_{ic}} \times 100\%$, and coverage of interface changes is $\rho_{ii} = \frac{m_{ii}}{\triangle_{ii}} \times 100\%$.

Changes of BPEL activity, binding and path condition behaves as changes of elements in XBFG model, so ρ_{ip}, ρ_{ib} and ρ_{ic} can be represented as the ratio of covered changes of XBFG elements and actual ones. The path set worked out by Algorithm 1 covers the new elements, modified elements, new bindings, modified bindings and part of deleted elements. Let $|n_n|$, $|n_m|$ and $|n_d|$ denote the number of new elements, modified elements and deleted elements respectively, $|n'_d|$ denote the number of covered deleted elements, then

$$\rho_{ip} = \frac{m_{ip}}{\triangle_{ip}} \times 100\% = \frac{|n_n| + |n_m| + |n'_d|}{|n_n| + |n_m| + |n_d|} \times 100\%$$

It can be inferred that ρ_{ip} depends on the proportion between $|n'_d|$ and the actual changes of XBFG elements.

Let $|b_n|$, $|b_m|$ and $|b_d|$ denote the number of new bindings, modified bindings and deleted bindings, then

$$\rho_{ib} = \frac{m_{ib}}{\triangle_{ib}} \times 100\% = \frac{|b_n| + |b_m|}{|b_n| + |b_m| + |b_d|} \times 100\%$$

Let $|c_n|$, $|c_m|$ and $|c_d|$ denote the number new path conditions, modified ones and deleted ones, then

$$\rho_{ic} = \frac{m_{ic}}{\triangle_{ic}} \times 100\% = \frac{|c_n| + |c_m|}{|c_n| + |c_m| + |c_d|} \times 100\%$$

WSDL document is composed of the definitions of variables, messages, portTypes, bindings, ports and services, while our method may only cover the messages and variables used by the composite service. Let $|V_{Def}|$, $|M_{Def}|$, $|PT_{Def}|$, $|B_{Def}|$, $|P_{Def}|$ and $|S_{Def}|$ denote the *variables*, *messages*, *portTypes*, *bindings*, *ports* and *services* defined in WSDL of composite service and components services, $|V_{Use}|$ and $|M_{Use}|$ denote the number of *covered variables* and *messages*, then

$$\rho_{ii} = \frac{m_{ii}}{\triangle_{ii}} \times 100\%$$

$$= \frac{|V_{Use}| + |M_{Use}|}{|V_{Def}| + |M_{Def}| + |PT_{Def}| + |B_{Def}| + |P_{Def}| + |S_{Def}|} \times 100\%$$

It can be inferred that ρ_{ii} depends on the proportion between $(|V_{Use}| + |M_{Use}|)$ and number of actual changed elements in WSDL documents.

B. Experimental Results

Loan Flow is a composite service deployed in Oracle BPEL PM Server. It has passed through a continuous evolution from v1.0, v1.1 to v2.0. v1.0 is the initial version and the other two are modified versions based on v1.0. As a result, v1.1 and v2.0 have to be retested to assure their availability. V1.0 is composed of a BPEL process `LoanFlow.bpel` and three component services, which are `CreditRatingService`, `UnitedLoanService` and `StarLoanService`. The internal business flow of v1.0 is as follows. The process receives loan application from the client and picks up SSN from application. Then it invokes `CreditRatingService` and waits for its result of customer loan rank. After receiving this query result, the process initiates two tasks concurrently. In the first task the process sends messages to `UnitedLoanService` and receives its loan application result. In the second task, the process sends messages to `StarLoanService` and receives its result. Having done the two tasks, the process compares the two loan application result and select the service whose APR is lower as the loan service. In the end, the process sends the selection result to the customer.

In v1.1, the designer decided to use another candidate service of `CreditRatingService` to replace the one in v1.0 without any functionality change and interface change. That is, a binding change has occurred in v1.1.

However, more improvements have been made in v2.0. Two additional component services are introduced to this version, one is `customerService`, which provides the function of SSN querying, and the other is `TaskService`, which is a user task providing manual checking by users. So two partner links are added to the process, and some new interfaces are introduced in. Besides, process of this version is more complex than the previous version.

In order to generate test suite for v1.1 and v2.0, comparisons should be made among the three versions based on XBFG model. The models constructed for the three versions are shown in Figure 2. Figure 2(a), (b) and (c) is mapped from v1.0, v1.1 and v2.0 respectively. The number near node or edge is the id. Note that the comparison is between two versions, so the regression testing analysis of v1.1 and v2.0 is independent from each other. In order to identify the two modeling, id with a bracket in Figure 2(b) represents the id when operating regression testing analysis between v1.1 and v2.0.

Test case selection and generation result of v1.1 is shown in Figure 3. In this figure, "Previous" denotes the baseline version, i.e. v1.0; "New" denotes the modified version, i.e. v1.1. "$N(E)$" is the number of XBFG elements and "$C(E)$" is number of changed XNFG elements. Both of the versions have 45 elements and the changed number is 6. "$N(M)$" is the number of messages related to the paths of XBFG and "$C(M)$" is the number of changed ones. "$N(V)$" is the number of variables related to the composite service and "$C(V)$" is the number of changed ones. It can be found out that both the messages and variables have no changes from v1.0 to v1.1. "$N(P)$" is the number of paths and "$C(P)$" is the number of changed ones. As the 6 changed elements are the common elements of the 2 paths, so the 2 paths are both changed paths. "$N(T)$" is the number of test cases, "$S(T)$" is the number of selected ones and "$G(T)$" is the newly generated ones. There are 6 test cases in v1.0, 3 are of p1 and the other 3 is of p_2. As p_1' in v 1.1 can use test cases of p_1, p_2' can use test cases of p_2, v 1.1 need no new test cases. So the number of selected ones is 6 and the generated number is 0. In this case, 1 binding has changed and test cases covered this changed binding. As a result, $\rho_{ib} = 100\%$. This case shows that regression testing caused by binding change doesn't need to generate new cases. This result is consistent with the expectation.

Figure 3 and figure 4 show test case selection and generation result of v1.1 and v2.0, respectively.

As all of the 4 paths in v 2.0 have been influenced by process, binding and interface changes, they should be retested and all of them need new test cases. The number of test cases of v1.1 is 6 and the number of v2.0 is 48, 12 of which are selected and 36 of which are newly generated. The coverage of test cases is as follows:

- Process: Actually 13 activities were added and the experiment covered these 13 activities, so ρ_{ip}=100
- Binding: Actually 2 bindings were added and the experiment covered these 2 activities, so ρ_{ib}=100
- Path condition: Actually 2 predicate constraint were added and the experiment covered these 2, ρ_{ic} =100
- Interfaces: Actually 132 elements were added to *CreditRatingService.wsdl* and *TaskServiceWSIF.wsdl*, the experiment covered 10 of them, so ρ_{ii} = 10/132*100% = 7.58%. The 10 covered interface elements are all used by the composite service.

In this experiment, we adjust the number of interface elements irrelative to the composite service and the result shows keeps increscent when the number of irrelative elements grows smaller.

V. Related work

The emergence of Web service imposes a great challenge on the concept and technology of regression testing. Researchers are trying to find ways of operating regression testing on composite service according to its characteristics. In [7], comparisons were made on the cost and restriction of service regression test operating by different stakeholders. Xiao et al. made change impact analysis on business process level and code level, and construct Impact Propagation Graph on the basis of analyzing Call Graph in [8]. In [9], Web service was abstracted as a two-level model, which can be expressed by input-complete TLTS. Based on the safe and efficient regression test selection technique proposed by Gregg Rothermel et al.[10], M. Ruth et al. exploited regression testing selection algorithm by integrating CFGs[12], [11], [13]. However, it is difficult for service integrators to obtain whether TLTS, CFG or internal process of basic services. Cost is another problem that should not be ignored during regression testing. The cost can be reduced by building service stubs to simulate behaviors of messages exchanges between services against data collected by monitoring[14]. Algebraic model and control flow are two models used in Regression testing of BPEL composite service until recently. J. A. Ginige et al expresses BPEL control flow as algebraic expression using Kleen Algebra, in this way changes of process are identified by comparing algebraic expressions[15]. Compared with CFG model, algebraic expression model may encounter difficulties when expressing complex structures. Control flow model is used in change impact analysis and regression testing path selection on BPEL process in [16]. Li et al.[17] exploited a test-selection minimization algorithm based on [16]. This method only considers process, which is just a part of composite service. Component service and interfaces are ignored in [16], but is discussed in this paper.

VI. Conclusion and Future Work

The new characteristics of Web service impose a great challenge on the concept and technology of testing and maintenance. This paper attempts to eliminate the influence caused by binding, process and interface changes by selecting and generating a proper quantity of test cases. Taking process and interface specification as the entry of analyzing, this approach models the composite service using XBFG, which can express not only the process, but also the message exchanges between process and component services. Furthermore, the binding and predicate constraint added to XBFG element is used for test paths selection and test data generation. Using set of XBFG elements, including nodes and edges, to express paths, can help finding out paths to be tested according to the new elements and deleted elements gained by path comparison. The analysis of path conditions cuts down the quantity of test cases generated. The approach makes use of XBFG model to analyze the influence on regression testing path caused by binding and interface changes together with process modification, which cover the main aspects of functional regression testing

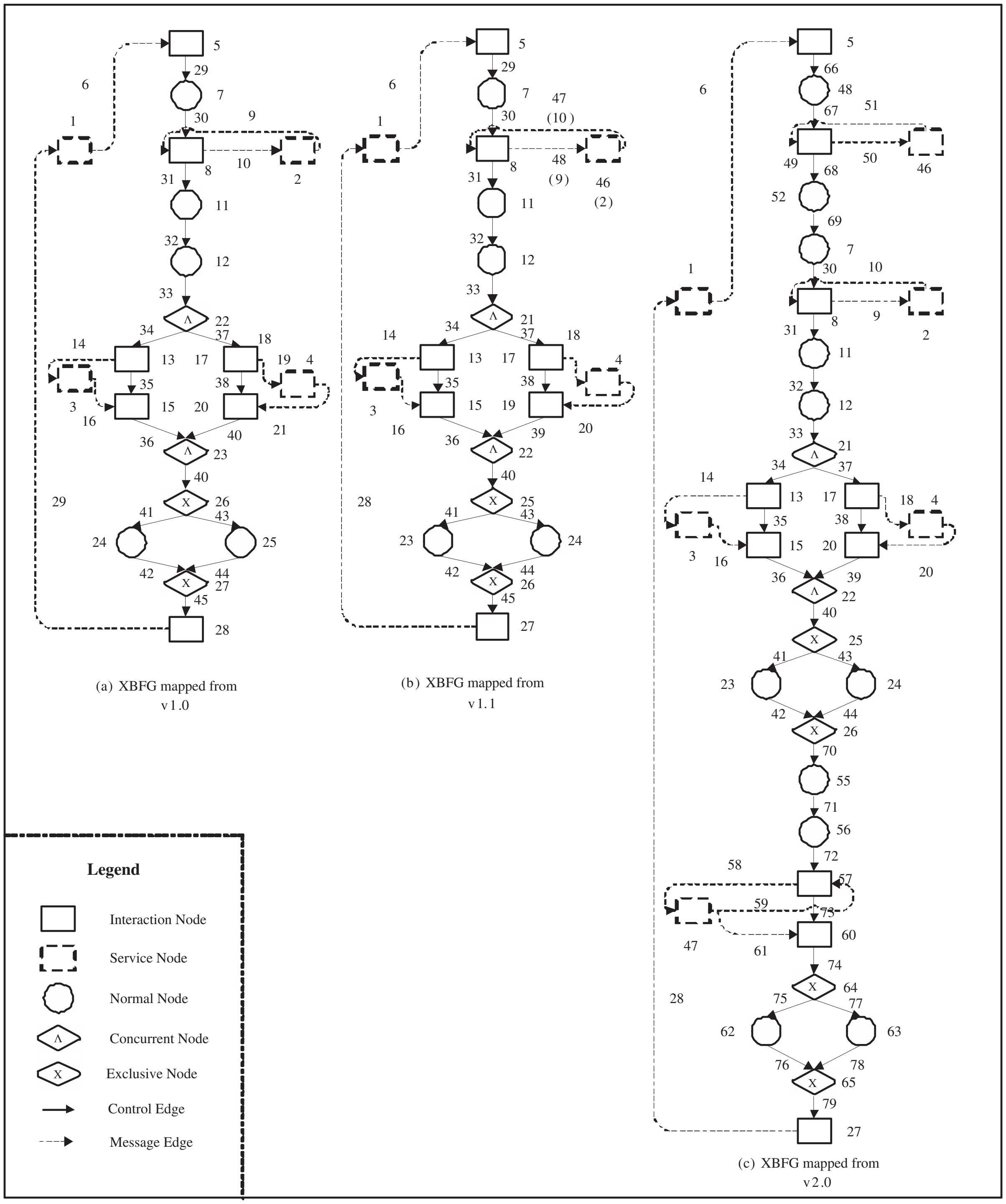
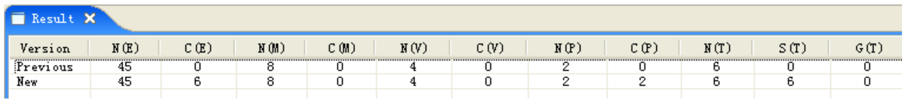
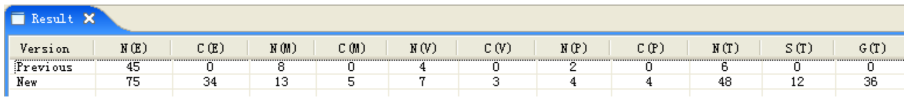


Figure 2. XBFG mapped from LoanFlow composite service



Version	N(E)	C(E)	N(M)	C(M)	N(V)	C(V)	N(P)	C(P)	N(T)	S(T)	G(T)
Previous	45	0	8	0	4	0	2	0	6	0	0
New	45	6	8	0	4	0	2	2	6	6	0

Figure 3. Test case selection and generation experimental result of v1.1


Result

Version	N(E)	C(E)	N(M)	C(M)	N(V)	C(V)	N(P)	C(P)	N(T)	S(T)	G(T)
Previous	45	0	8	0	4	0	2	0	6	0	0
New	75	34	13	5	7	3	4	4	48	12	36

Figure 4. Test case selection and generation experimental result of v2.0

of service composition. Generally speaking, this approach extends the study on regression testing to the composite service but not only the process. The result of test cases selection and generation get a high coverage of modifications when changes occurred in process, bindings and interfaces. Composite services usually come from different vendors and service invoking will increase the cost of regression testing. So a problem to be solved is how to reduce the cost of regression testing. Besides, BPEL is only one of the service composition languages. It would be more useful to develop a more genetic regression testing framework.

Acknowledgements

The authors thank Rajiv Gupta, Dennis Jeffrey, and Chen Tian at University of California Riverside for their valuable suggestions on the first draft of this paper.

References

[1] A. Alves, A. Arkin , S. Askary , et al, *OASIS Standard, Web Services Business Process Execution Language Version 2.0*, http://docs.oasis-open.org/wsbpel/2.0/, 2007.

[2] M. Gudgin, M. Hadley, T. Rogers, et al, *Web Services Addressing 1.0 - WSDL Binding*, http:// www. w3. org/ TR/ ws-addr-wsdl, 2006.

[3] Christensen E, Curbera F, Meredith G, et al, *W3C Note, "Web Services Description Language (WSDL) 1.1* http://www.w3.org/TR/wsdl, 2001-03

[4] Y. Yuan, Z. Li, and W. Sun, *A Graph-search Based Approach to BPEL4WS Test Generation*, In: Proceedings of the International Conference on Software Engineering Advances (ICSEA'06) , Tahiti , Oct. 2006, pp. 14-14.

[5] D. Wang, B. Li, and J. Cai, *Regression Testing of Composite Service: An XBFG-Based Approach*, In: Proceedings of 2008 IEEE Congress on Services Part II (SERVICES-2).Beijing, 2008, pp. 112-119.

[6] G. Rothermel and M. J. Harrold, *Analyzing Regression Test Selection Techniques*. IEEE Transactions on Software Engineering, 1996, 22 (8), pp. 529-551.

[7] G. Canfora, and M. D. Penta, *Testing Services and Services-Centric Systems: Challenges and Opportunities*, IT Pro March ? April 2006, pp. 10-17.

[8] H. Xiao, J. Guo, and Y. Zou, *Supporting Change Impact Analysis for Service Oriented Business Applications*. In: Proceedings of the 29th International Conference on Software Engineering Workshops: International Workshop on Systems Development in SOA Environment (SDSOA), Minneapolis, 2007, pp. 116-121.

[9] A. Tarhini, H. Fouchal, and N. Mansour, *Regression Testing Web Services-based Applications*, In: Proceedings of the IEEE International Conference on Computer Systems and Applications (AICCSA), 2006, pp. 163-170.

[10] G. Rothermel and M. J. Harrold, *A Safe, Efficient Regression Test Selection Technique*, ACM Transactions on Software Engineering and Methodology, 1997, 6 (2) , pp. 173-210.

[11] M. Ruth, S. Oh, and A. Loup *Towards Automatic Regression Test Selection for Web Services*, In: Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC 2007), USA, 2007, pp. 729-736.

[12] M. Ruth and S. Tu, *A Safe Regression Test Selection Technique for Web Services*, In: Proceedings of the Second International Conference on Internet and Web Applications and Services (ICIW'07), pp. 47-47.

[13] F. Lin, M. Ruth, and S. Tu, *Applying Safe Regression Test Selection Techniques to Java Web Services*, In: Proceedings of the International Conference on Next Generation Web Services Practices (NWeSP), Seoul, 2006, pp. 133-140.

[14] G. Canfora and M. D. Penta, *SOA Testing and Self-Checking*, In: Proceedings. of International Workshop on Web Services - Modeling and Testing (WS-MaTe), Palermo, pp. 3-12.

[15] J. A. Ginige, U. Sirinivasan, and A. Ginige, *A Mechanism for Efficient Management of Changes in BPEL based Business Processes: An Algebraic Methodology*, In: Proceedings of the IEEE International Conference on e-Business Engineering (ICEBE'06), USA, 2006, pp.171-178.

[16] H. Liu, Z. Li, J. Zhu, and H. Tan *Business Process Regression Testing*, In: Proceedings of the 5th International Conference on Service Oriented Computing (ICSOC 2007), LNCS 4749, 2007, pp. 157-168.

[17] Z. J. Li, H. F. Tan, H. H. Liu, J. Zhu, and N. M. Mitsumori *Business-process-driven gray-box SOA testing*, IBM System Journal, 2008, 47 (3), pp.457-472