The Journal of Systems and Software 85 (2012) 1300–1324

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Automatic test case selection for regression testing of composite service based on extensible BPEL flow graph

Bixin Li[a,*], Dong Qiu[a], Hareton Leung[b], Di Wang[a]

[a] School of Computer Science and Engineering, Southeast University, Nanjing 210096, China
[b] Department of Computing, The Hong Kong Polytechnic University, Hong Kong, China

ARTICLE INFO

Article history:
Received 23 January 2011
Received in revised form 13 January 2012
Accepted 22 January 2012
Available online 8 February 2012

Keywords:
Regression testing
Web service
Test case selection
Extensible BPEL flow graph

ABSTRACT

Services are highly reusable, flexible and loosely coupled components whose changes make the evolution and maintenance of composite services more complex. The changes of composite service mainly cover three types, i.e., the *processes*, *bindings*, and *interfaces*. In this article, an approach is proposed to select test cases for regression testing of different versions of BPEL (business process execution language) composite service where these changes are involved. The approach identifies the changes by performing control flow analysis and comparing the paths in a new version of composite service with those in the old one using a kind of eXtensible BPEL flow graph (XBFG). *Message sequence* is appended to *XBFG path* so that XBFG can fully describe the behavior of composite service. The *binding* and *predicate* constraint information added in different *XBFG elements* can be used for path selection and even for test case generation. Both theoretic analysis and case study show that the proposed approach is effective.

© 2012 Elsevier Inc. All rights reserved.

1. Introduction

There are a lot of Web services that have been developed, registered and deployed in the Internet. We can send *call request* to UDDI (universal description, discovery, and integration) centers to ask for the use of some Web services when we plan to integrate them into our application or compose them into a stronger service. The services are usually classified into two types: (1) *basic service* or *atomic service*, which has been developed by service developer and it is self-contained as it does not require other services; (2) *composite service*, which is composed of some *basic services* and other *composite services* according to some composing mechanism by service developer or service integrator so as to provide stronger function to its users.

In current practice, service-oriented integration is a mainstream application field of service computing, and the emergence of service composition technology makes the integration more convenient and efficient. On the one hand, service is a kind of component that can be highly reusable, flexible and loosely coupled, which makes service computing more significant in the distributed computing discipline. On the other hand, the evolution and maintenance of composite service will take on different looks from some traditional software technologies because of these characteristics. However, service user usually cannot access the source code of a basic service used in his system which adds to the difficulty in controlling the evolution of service.

Regression testing plays a very important role during the evolution and maintenance of composite service (Yoo and Harman, 2010). When any change happened to a service, regression testing must be performed to check whether or not some new faults have been introduced. The inherent characteristics, such as ultra-late binding mechanism and non-observability of web service source codes (Canfora and Penta, 2006, 2009), make the regression testing for web service more challenging. Many works (Hou et al., 2008; Mei et al., 2009, 2011) have applied test case prioritization techniques to select test cases with higher APFD (Elbaum et al., 2002) (average percentage faults detected) to verify whether the functions of the modified service conform to the pre-defined requirements. Since service users cannot obtain the source code, they mainly use interface information that can be covered to ranking the ability of error-detection of test cases. Although prioritization technique can determine the execution order of test cases, it cannot answer the question how many test cases are enough for testing the evolved version of services. So test case selection techniques are introduced in web service regression testing. Some works (Canfora and Penta, 2006; Penta et al., 2007; Keum et al., 2006) proposed their methods, especially aiming at basic services. However, less attention was paid on *composite service*. Existing techniques, such as Ruth et al. (2007) and Ruth and Tu (2007), who have applied *graph walk analysis* technique (Rothermel et al., 1997) in the area of web service, assume that the structure of all participating services are provided by corresponding service developers. They only focus on the functionality of center

* Corresponding author. Tel.: +86 25 83790109; fax: +86 25 52090879.
E-mail addresses: bx.li@seu.edu.cn (B. Li), dongqiu@seu.edu.cn (D. Qiu), cshleung@inet.polyu.edu.hk (H. Leung), di.wang@seu.edu.cn (D. Wang).

0164-1212/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2012.01.036

services and omit the feature of dynamic binding in composite web services.

From the perspective of *service integrators*, they have the testing information of both structure of service process and interfaces of partner services. So the challenge is that, on the one hand, service integrator need to check both the behavior of process itself and the interactive behaviors between services to guarantee the correctness of entire composite service. On the other hand, all participating services, including self-designed services or partner services managed by third parties, may also evolve during their own life cycle. This demands service integrators to proactively detect changes of partner service. We will, therefore, discuss how to model the entire testing procedure in this paper to conquer the above challenge.

The main contribution of this paper with its preliminary version (Wang et al., 2008; Li et al., 2010) is fourfold: (1) we propose the revised *eXtensible BPEL Flow Graph* (XBFG) to model BPEL-based composite service precisely. The core idea of XBFG is to construct *XBFG path* recording the execution trace of web service, with newly introduced concepts *in-process* path and *out-process* path, where the former focuses on depicting the behavior of process itself and the latter focuses on interactive behavior between process and partner services. In addition, both XBFG model construction, including transformation rules of BPEL basic and structure activity, and XBFG path generation are illustrated in detail in this paper. (2) *XBFG message sequence* is newly proposed to record the message exchanges between process and partner services, which is a direct evidence to detect the *interface change* of composite web service. The corresponding message sequence generation and comparison algorithms are provided as well. (3) we provide an updated classification of change types (by removing the "path condition change") from the perspective of service integrator and provide the graphical definitions of different change types, with comparison and relation between them. (4) we explore five versions of carefully designed subject composite service, by which we show how to effectively and

```
     <process name="LoanFlow"...>
       <partnerLinks>
(1)      <partnerLink name="client".../>
(2)      <partnerLink name="creditRatingService".../>
(3)      <partnerLink name="UnitedLoanService".../>
(4)      <partnerLink name="StarLoanService".../>
       </partnerLinks>
       <variables>...</variables>
       <sequence>
         <!--all partnership link assignment activities are ingored here-->
(5)      <receive name="receiveInput" partnerLink="client".../>
         <scope name="GetCreditRating" variableAccessSerializable="no">
           <sequence>
(7)          <assign name="copySSN">
               ...
               <copy>
                 <from><literal><service-ref><EndpointReference>
                   <Address>http://www.ratingService.com/</Address>
                   <ServiceName>invoiceService</ServiceName>
                 </EndpointReference></service-ref></literal></from>
                 <to partnerLink="creditRatingService" />
               </copy>
             </assign>
(8)          <invoke name="invokeCR" partnerLink="creditRatingService".../>
(11)         <assign name="copyRating">...</assign>
           </sequence>
         </scope>
         <scope name="GetLoanOffer" variableAccessSerializable="no">
           <sequence>
(12)         <assign name="prepareApplication">...</assign>
(21)           <flow name="collectOffers">
                 <sequence>
(13)               <invoke name="invokeUnitedLoan"
                          partnerLink="UnitedLoanService".../>
(15)               <receive name="receive_invokeUnitedLoan"
                          partnerLink="UnitedLoanService".../>
                 </sequence>
                 <sequence>
(17)               <invoke name="invokeStarLoan"
                          partnerLink="StarLoanService".../>
(19)               <receive name="receive_invokeStarLoan"
                          partnerLink="StarLoanService".../>
                 </sequence>
(22)           </flow>
           </sequence>
         </scope>
         <scope name="SelectOffer" variableAccessSerializable="no">
(25)       <if name="selectBestOffer">
             <condition>
               "bpws:getVariableData('loanOffer1',
                    'payload','/auto:loanOffer/auto:APR') >
             bpws:getVariableData('loanOffer2',
                    'payload','/auto:loanOffer/auto:APR') "
             </condition>
(23)           <assign name="selectStarLoanOffer">
                 <copy><from variable="loanOffer2" part="payload"/>
                 <to variable="selectedLoanOffer" part="payload"/></copy>
               </assign>
               <else>
(24)           <assign name="selectUnitedLoanOffer">
                 <copy><from variable="loanOffer1" part="payload"/>
                 <to variable="selectedLoanOffer" part="payload"/></copy>
               </assign>
               </else>
(26)       </if>
         </scope>
(27)     <invoke name="replyOutput" partnerLink="client".../>
       </sequence>
     </process>
```

(a) Verison 1.0

```
     <process name="LoanFlow"...>
       <partnerLinks>
(1)      <partnerLink name="client".../>
(2)      <partnerLink name="creditRatingService".../>
(3)      <partnerLink name="UnitedLoanService".../>
(4)      <partnerLink name="StarLoanService".../>
       </partnerLinks>
       <variables>...</variables>
       <sequence>
         <!--all partnership link assignment activities are ingored here-->
(5)      <receive name="receiveInput" partnerLink="client".../>
         <scope name="GetCreditRating" variableAccessSerializable="no">
           <sequence>
(7)          <assign name="copySSN">
               ...
               <copy>
                 <from><literal><service-ref><EndpointReference>
                   <Address>http://www.ratingService.com/</Address>
                   <ServiceName>invoiceService</ServiceName>
                 </EndpointReference></service-ref></literal></from>
                 <to partnerLink="creditRatingService" />
               </copy>
             </assign>
(8)          <invoke name="invokeCR" partnerLink="creditRatingService".../>
(11)         <assign name="copyRating">...</assign>
           </sequence>
         </scope>
         <scope name="GetLoanOffer" variableAccessSerializable="no">
           <sequence>
(12)         <assign name="prepareApplication">...</assign>
(21)           <flow name="collectOffers">
                 <sequence>
(13)               <invoke name="invokeUnitedLoan"
                          partnerLink="UnitedLoanService".../>
(15)               <receive name="receive_invokeUnitedLoan"
                          partnerLink="UnitedLoanService".../>
                 </sequence>
                 <sequence>
(17)               <invoke name="invokeStarLoan"
                          partnerLink="StarLoanService".../>
(19)               <receive name="receive_invokeStarLoan"
                          partnerLink="StarLoanService".../>
                 </sequence>
(22)           </flow>
           </sequence>
         </scope>
         <scope name="SelectOffer" variableAccessSerializable="no">
(25)       <if name="selectBestOffer">
             <condition>
               "bpws:getVariableData('loanOffer1',
                    'payload','/auto:loanOffer/auto:APR') >
             bpws:getVariableData('loanOffer2',
                    'payload','/auto:loanOffer/auto:APR') "
             </condition>
(23)(46)       <assign name="selectStarLoanOffer">
                 <copy><from variable="loanOffer1" part="payload"/>
                 <to variable="selectedLoanOffer" part="payload"/></copy>
               </assign>
             <else>
(24)           <assign name="selectUnitedLoanOffer">
                 <copy><from variable="loanOffer1" part="payload"/>
                 <to variable="selectedLoanOffer" part="payload"/></copy>
               </assign>
             </else>
(26)       </if>
         </scope>
(27)     <invoke name="replyOutput" partnerLink="client".../>
       </sequence>
     </process>
```

(b) Verison 1.1

Fig. 1. BPELs of *Loan composite service* with two versions.

precisely select test case for evolved version of composite service. Our empirical results indicate that our approach has more expressive capability in recording the entire behavior of composite service and can detect three kinds of change types (e.g. *process change*, *binding change*, and *interface change*). In addition, our approach is effective in selecting test cases for evolved BPEL-based composite services.

The rest of the paper is organized as follows: Section 2 introduces the WSDL and BPEL and gives a motivating example system used to illustrate our idea; Section 3 identifies the classifications of evolution and modification in Web service and gives an overview of our approach; Section 4 illustrates the definition and construction of both *XBFG* and *XBFG path* for modeling composite services; Section 5 discusses how to perform test case selection using XBFG in detail; Section 6 performs some experiment and evaluation of our approach by using the motivating example and its four modified versions; Section 7 compares the related works; Section 8 concludes the paper.

2. Background

In this section, the prerequisite knowledge of WSDL and BPEL are introduced first. Then BPEL-based service composition is summarized and a motivating example is provided for convenience in illustration.

2.1. WSDL summary

WSDL (Web Services Description Language) is an XML-based language for describing Web services and how to access them (Christensen et al., 2001). It specifies the location of the service and the operations (or methods) the service exposes. It stipulates the interactive rules to use or also integrate the services.

WSDL defines a service's abstract description in terms of messages exchanged in a service interaction (Curbera et al., 2002). A standard WSDL document usually contains two pieces of information. One is abstract-level description that mainly includes `portType`, `operation`, `message` and `type`; the other is access information that mainly includes `port` and `binding`.

Abstract-level description provides the functional interface of the service. A `portType` is consisted of a set of operations. An `operation` defines the message exchange pattern which stipulates the interaction between services. A `message` is an aggregation of *part*s, each of which is described by `type`. The `type` can be a kind of XSD (XML schema definition) built-in type, such as *string* and *boolean*, or a complex type that the user predefines.

Access information guides the service user to access service at concrete service end points. A `binding` defines how services communicate over the specified protocol. A `port` describes a single end point as a combination of a binding and a network address.

2.2. BPEL summary

Service composition is a way of reconstruction using existing services to provide value-added application. BPEL, as the de-facto standard on service composition among all composition languages (Alves et al., 2007), is popular in not only academic but also industrial community. It is an OASIS standard and XML-based executable language for specifying interactions with Web services. BPEL extends the Web services interaction model and enables it to support business processes. *Processes* written in BPEL can orchestrate interactions between Web services using standard XML documents.

Composite service generated using BPEL is a combination of *process* and *partner services*. *Process* is a plan composed of many baseline steps, where each step is called an *activity*. *Partner service*, like a basic service, is invoked through its external interface exposed to users, though its inner can be complex and changeful.

In BPEL specification, *activity* is classified into *basic activity* and *structural activity*. *Basic activity* can exist independently or in a *structural activity* and is used to describe the unit behaviors of *process*. The nine main kinds of *basic activities* defined in BPEL 2.0 specification are `invoke`, `receive`, `reply`, `assign`, `throw`, `wait`, `empty`, `extensionActivity`, `exit`, and `rethrow`. *Structural activity* prescribes the execution order of activities with control flow logic, and is generally regarded as a *container* of other activities. The main *structural activities* in BPEL 2.0 specification include `sequence`, `if`, `while`, `repeatUntil`, `pick`, `flow`, `forEach`, `scope`, etc. More details about these activities can be found in BPEL 2.0 specification (Alves et al., 2007).

In addition, both `partnerLink` defined in BPEL and the *endpointReference* mechanism from `WS-Addressing` are used to support service bindings (Gudgin et al., 2006). `PartnerLink` prescribes the interaction rules between BPEL process and partner services and only those satisfied with interface definition and functional requirement can be considered as candidate partner services. *endpointReference* is used to decide the service endpoint that the process will bind. In BPEL, we can use `assign` activity to copy the content of `EndpointReference` to corresponding `partnerLink`.

In this study, we focus on the problem of test case selection for regression testing of BPEL-based composite service.

2.3. A motivating example

We use the *loan composite service*[1] (LCS for short) extracted from the project of Oracle BPEL Process Manager as an running example. Here BPEL specifications of both the original version and modified version are shown in Fig. 1. For the sake of simplicity, nonessential statements such as space declaration, variable definition and assignment activities are ignored for saving space. The version 1.0 of *LCS* (*v1.0* for short) is composed of one service process *LoanFlow* and three partner services including *CreditRatingService*, *UnitedLoanService* and *StarLoanService*. *CreditRatingService* is a synchronous service which provides users with functions such as inquiring *loan grade*, accepting user's inquiry request, returning inquiry result, etc.; both *UnitedLoanService* and *StarLoanService*, which share the same WSDL file, are asynchronous services providing the function of loan. The process *LoanFlow* has four `partnerLink`s, where *client* is used to call this composite service; *CreditRatingService*, *UnitedLoanService* and *StarLoanService* all denote partner services that the process invokes. *LoanFlow* first receives a loan request from a client, then calls partner service *CreditRating Service* for confirming the client's loan grade using SSN (social security number) filled in client's application form. The process will activate two concurrent tasks as soon as the inquiring result has been received and confirmed: *UnitedLoanService* and *StarLoanService* both receive request from the process and return loan application result. Then, *LoanFlow* compares all results and chooses the partner service with the minimal APR (annual percentage rate) value as the loan application goal, and returns the chosen result to client. In version 1.1 of *LCS* ($v1.1$ for short), the service integrator modified the content of `assign` in line 23.

3. Testing perspectives and composite service evolution

In this section, we identify five key perspectives of regression testing web services. Then we propose a new classification

[1] Detail is available at http://www.oracle.com/technology/products/ias/bpel/index.html.

Table 1
Testing perspective for different stakeholders.

Testing perspective	Ownership	Testing strategy
Service developer	Source code of BS[a] WSDL of BS	Black-Box White-Box
Service provider Service publisher	WSDL of S[c] WSDL of S	Black-Box Black-Box
Service integrator	BPEL of CS[b] WSDL of CS and BS	White-Box Black-Box
Service user	WSDL of S	Black-Box

[a] BS is the abbreviation of basic service.
[b] CS is the abbreviation of composite service.
[c] S is the abbreviation of service which is composed of BS and CS.

of evolution types of composite service. Finally, an overview of our approach for testing composite service is provided.

3.1. Testing perspectives

Due to the discriminative accessibility of service resources, the testing emphasis of different stakeholders may be different. It is necessary to clearly define the testing duties and strategies of every stakeholder as the service evolves. Table 1 shows the ownership and test strategy of five key stakeholders: *service developer*, *service provider*, *service publisher*, *service integrator*, and *service user*.

In this article, we will focus on the perspective of *service integrator* when testing change to BPEL process and the interaction with partner services.

3.2. Change types of composite service

In general, BPEL-based composite service is composed of a *process*, an *interface* described in WSDL specification (Christensen et al., 2001) and *partner services* interacting with the process. Therefore, the evolution of BPEL composite service usually involves three types of changes, i.e., the change of *process*, the change of *interface* and the change of *binding*. Fig. 2 shows the evolution of composite service caused by different types of changes, where $S1$, $S2$, $S3$, $S4(a)$, and $S4(b)$ denote composite services, $A1, A2, \cdots, A7$ denote activities, $P1$ and $P2$ denote partner services, and $W1$ and $W2$ denote WSDL specifications of $S1$ and $P1$, respectively.

- *Process change* includes the change of BPEL activities and the change of activities order. Service integrators may change the internal structure of process due to new functional requirements, where the addition or deletion of services, change of activities, and the changes of execution sequence are all regarded as *process change*. In Fig. 2, composite service $S1$ evolves to $S2$ by adding a new activity $A7$ to $S1$.
- *Binding change* is the change of endpoint addresses of *partner services*. For example, the service integrator selects another candidate service to replace the original one which now is unavailable. In Fig. 2, $S1$ evolves to $S3$ because the partner service that interacts with $A2$ has changed from $P1$ to $P2$.
- *Interface change* includes *composite service interface change* and *partner service interface change*. In WSDL Specification(Christensen et al., 2001), the interface of service is composed of the definitions of the variables, messages, operations and ports. So the interface change of service usually means the change of these variables, messages, operations and ports, as defined in a WSDL document. In most cases, the service integrator modifies the interface of composite service to improve the readability and programmability of WSDLs. In Fig. 2, $S1$ evolves to $S4(a)$ because interface document $W1$ has changed to $W1'$. In addition, if the provider of a partner service modifies the interface of the partner service, this will force the service integrator to make corresponding change to the interface or process of composite service in order to use the same partner service. In Fig. 2, $S1$ evolves to $S4(b)$ because the interface $W2$ of partner service $P1$ has changed to $W2'$ which causes $A2$ to change in order to match the modified interface of $P1$.

Table 2
Comparison of change types from the perspective of service integrator.

Change type	Location	Manageability	Propagation
Process change	BPEL	Controllable	Binding change Interface change (CS)
Binding change	BPEL	Controllable	
Interface change (CS[a])	WSDL	Controllable	Process change
Interface change (PS[b])	WSDL	Uncontrollable	Process Change

[a] CS is the abbreviation of Composite Service.
[b] PS is the abbreviation of Partner Service.

It is important to understand the characteristics of these different types of changes and their relationship since we can gain some insight into regression testing. Table 2 provides a brief comparison from three aspects:

- *Location*. *Location* refers to the position where changes take place. *Process change* and *binding change* occur in BPEL documents while *interface change* occurs in WSDL documents.
- *Manageability*. Manageability refers to the control the service stakeholders have over the changes. From the perspective of *service integrator*s, they own both process implementation and process interface, which means that *process change*, *binding change* and *composite service interface change* are controllable. When these changes occur during the service evolution, *service integrators* can perform testing based on the result of change impact analysis. However, *partner service*s used in the process are often developed and managed by other *service developer*s or *service provider*s, which means the *partner service interface* and its implementation are out of control of *service integrator*s. If no change notification is received, *service integrator*s will not know when and where the changes occur.
- *Propagation*. Propagation refers to the influence that the occurrence of one change type may have on the occurrence of other change type(s). On the one hand, *process change* may cause *composite service interface change* and *binding change* because service integrator can modify the interface of composite service or use other service to replace the current service. On the other hand, since *service integrator*s do not have the control over *partner services*, they may have to make passive *process change* to adapt to the change in the interface of the partner service. That is, the occurrence of *process change* is forced by the occurrence of *partner service interface change*.

3.3. Outline of our approach

We propose a new approach to solve the regression test case selection problem of BPEL-based composite service. We will use the binding change in Fig. 2 (from P1 to P2) as the example to explain our approach. Fig. 3 provides a diagrammatic presentation of our solution. There are four key steps:

- **XBFG construction**. For any composite service, XBFG is created to express the complete behavior of composite service, where binding information and predicate constraints are added as XBFG elements for XBFG path computation and comparison. In Fig. 3, the visual XBFG models of both old version S1 and new version S3 are constructed as *XM1* and *XM3*, respectively. Take *XM1* as an

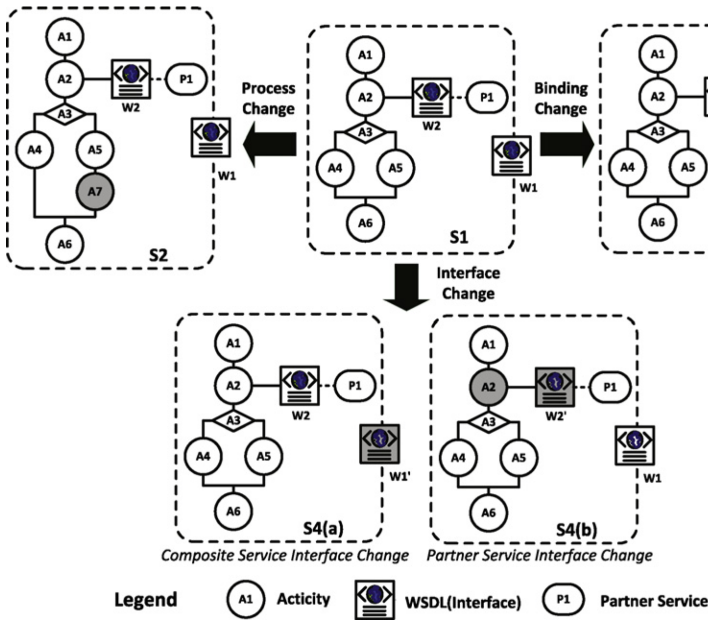


Fig. 2. Evolution of composite service.

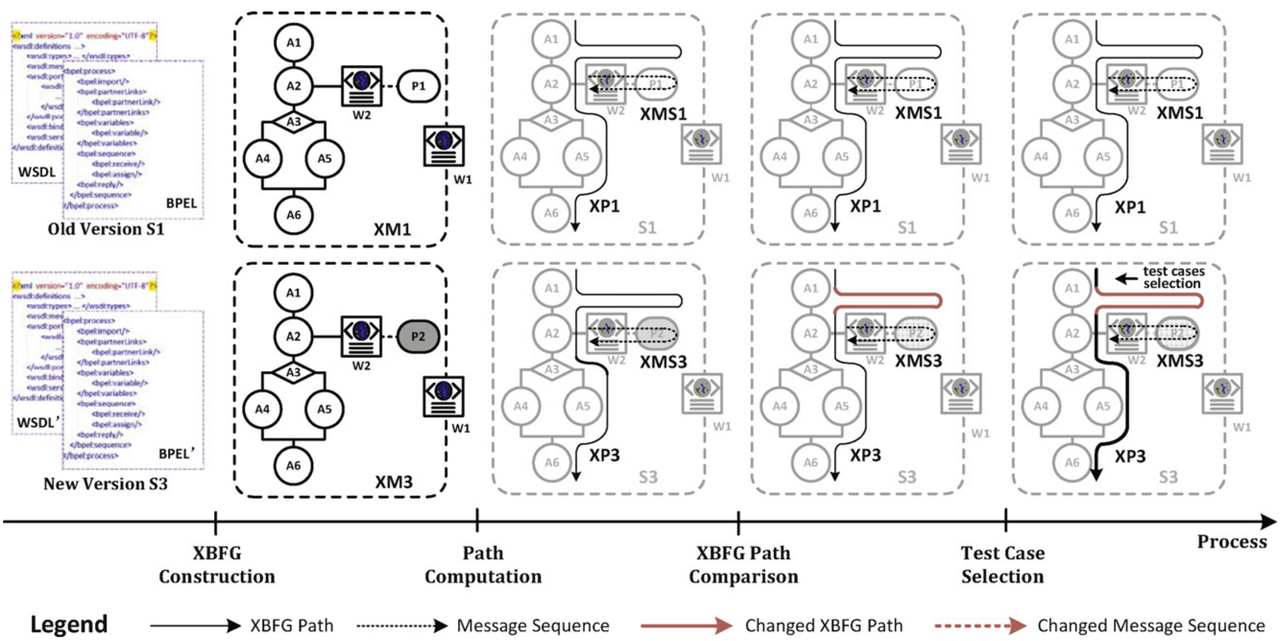


Fig. 3. Outline of our approach.

example, it consists of activities in process (such as *A1*), partner services (such as *P1*) and control flow relation (such as solid line between *A1* and *A2*).

- **XBFG path computation**. Based on generated XBFG, all XBFG paths are defined and constructed for the selection of test cases, where message sequences are calculated and attached to the corresponding XBFG paths. In Fig. 3, both XBFG paths of *XM1* and *XM3* are calculated and only one XBFG path in each version (*XP1* for *XM1* and *XP3* for *XM3*) is shown as a representative in this figure. The dashed line represents the message sequence of the corresponding XBFG path (*XMS1* for *XP1* and *XMS3* for *XP3*).
- **XBFG path comparison**. XBFG path comparison are performed to find *process change* and *binding change* and attached message sequence comparison to find *interface change* so as to determine which paths can be checked again by using selected test cases of the baseline version, and which paths must be checked using newly generated test cases. In Fig. 3, we perform pair-wise comparison on XBFG paths in two versions and find out that binding change has occurred in new version *S3* since *XP3* does not equal to *XP1*.
- **Test case selection**. After XBFG path comparison, test sets that can be reused on the subsequent version are identified according to the comparison result and mapping relation between XBFG

path and test suite. In Fig. 3, we select test cases attached with *XP1* to test *XP3*.

4. XBFG model

In this section, we discuss how to define XBFG, how to construct XBFG, how to define and generate XBFG path and how to define the message sequence.

BPEL flow graph (BFG) is a control flow model of BPEL process (Yuan et al., 2006). It supports concurrent control flow compared with traditional control flow graph (CFG) since it can describe BPEL process completely and can be used to identify the *process change*. However, the implementation of BPEL composite service involves the combination of *process* and *partner services* interacting with the process. The inability to express the interaction between BPEL process and partner services makes BFG not suitable for change impact analysis involving *binding change* and *interface change*.

In order to do change impact analysis on composite service rather than just the process, we propose the XBFG model. Compared with BFG, XBFG has following advantages: (1) BFG models BPEL process, while XBFG models not only BPEL process but also partner services that are used by the process; (2) based on the control flow relation for BPEL, XBFG defines message sequence to depict the interactive message flow between process and partner services; (3) *field* is introduced in XBFG to record information about interfaces and path conditions for the purpose of regression testing, while no such information is available in BFG.

4.1. XBFG definition

We first give the original definition of BFG, and then the formal definition of XBFG.

Definition 1 *(BFG).* The structural definition of BFG is as follows: BFG= $\langle N, E, s, F\rangle$, where *N* is a set of nodes, *E* is a set of edges, *s* is the start node, and *F* is a set of final nodes. $N=\{n_i\}$, $1\le i\le p$, *p* is the number of BFG nodes, where $n_1=s$, $n_i\in\{NN, DN, MN, FN, JN\}$, where *NN*, *DN*, *MN*, *FN*, *JN* denote *Normal Node*, *Decision Node*, *Merge Node*, *Fork Node*, and *Join Node*, respectively; $E=\{e_j\}$, $1\le j\le q$, *q* is the number of BFG edges, where $e_j=\langle a, b\rangle$, $a, b\in N$, $e_j\in\{TE, FE\}$ where *TE* and *FE* denote *True Edge* and *False Edge/Dead Path*, respectively (Yuan et al., 2006).

Definition 2 *(BFCG).* XBFG is defined as a quadruple $\langle$XE, s, F, $\Xi\rangle$, where *XE* is a set of XBFG elements which consist of XBFG nodes and XBFG edges. *s* is the start element, and *F* is a set of final elements; Ξ is the field of XBFG element. $XE=N\cup E$ where *N* and *E* denote the set of all XBFG nodes and edges, respectively. $N=IN\cup NN\cup SN\cup EN\cup MN\cup CN$ where *IN*, *NN*, *SN*, *EN*, *MN*, *CN* denote *Interaction Node*, *Normal Node*, *Service Node*, *Exclusive Node*, *Multiple Node* and *Concurrent Node*, respectively; $E=CE\cup ME$ where *CE* and *ME* denote *Control Edge* and *Message Edge*, respectively.

In BPEL, not only basic but also structural activities can be transformed into XBFG nodes which are classified into following six types:

- *Interaction Node (IN)*. It is created for those basic activities which interact with the *partner services*. These basic activities include `invoke`, `receive`, `reply` and `onMessage` in `pick`.
- *Normal Node (NN)*. It is created for other basic activities which do not belong to *IN*, such as `assign`, `wait` and so on. Additionally, it is also created for `onAlarm` in `pick`.
- *Service Node (SN)*. It is created for every partner service that is defined by `partnerLink` in BPEL document. *SN* always appears accompanied by *IN*. Additionally, it is also created for `receive` when it is used as a start activity defined in BPEL document.
- *Exclusive Node (EN)*. It is also called "XOR" node which has two sub types: *Exclusive Decision Node (EDN)* when its in-degree equals to 1 while its out-degree is greater than 1; *Exclusive Merge Node (EMN)* when its in-degree is greater than 1 while its out-degree equals to 1. *EN* is created for each of those activities that contain conditional behavior, including `if`, `pick`, `while`, and `repeatUntil`.
- *Multiple Node (MN)*. It is also called "OR" node, which is created for `link` when its value of `joinCondition` is "OR" or null. *MN* is divided into *Multiple Branch Node (MBN)* and *Multiple Merge Node (MMN)*.
- *Concurrent Node (CN)*. It is also called "AND" node, which is created for `flow` activity and `link` when its value of `joinCondition` is "AND". *CN* also has two forms, i.e., *Concurrent Branch Node (CBN)* and *Concurrent Merge Node (CMN)*.

In addition, the connection between activities defined in BPEL can be characterized by XBFG edges which are classified into following two types:

- *Control Edge (CE)*. It is created for control flow of *XBFG nodes*.
- *Message Edge (ME)*. It is created for message exchange between *IN* and *SN*.

Field is a determinative part in XBFG definition. It records related information of each XBFG element to support further analysis. In our approach, the following fields, namely, *ID*, *Source*, *Target* and *Category* are required for all XBFG elements.

- *ID* field is used to identify the XBFG element and discover whether its content has been changed. It is defined as a two-tuples $\langle$*id*, *hashcode*$\rangle$. The sub field *hashcode* is a string array generated by a hash function to check the changes of XML document (Maruyama et al., 2012). BPEL specification is in fact a XML document. It is very important to produce *hashcode* for each XBFG element during the transforming process from BPEL to XBFG, because the change of BPEL activities could be detected easily by comparing the *hashcode* of elements in two XBFGs after transforming BPEL to XBFG. Only those activities whose *names*, *attributes* and *sub-elements* all are the same can be regarded as unchanged. The sub field *id* is needed since *hashcode* cannot distinguish XBFG elements when the same activity exists in the same BPEL document many times. The value of *id* is a natural number generated according to the *hashcode* and it is unique to serve as the identity of XBFG element. We use *ID.id* to represents an XBFG element for short.
- *Source (Target)* field records the set of precedent (subsequent) elements of a XBFG element where precedent (subsequent) elements are consisted of XBFG edges for an XBFG node or XBFG nodes for a XBFG edge.
- *Category* field denotes the category of each XBFG elements. Its value can be *IN*, *SN*, *NN*, *EN*, *MN*, *CN*, *CE* and *ME*.

Some XBFG elements have special fields:

- *Name* field represents the name of XBFG element. Its value is the *name* attribute in the corresponding BPEL activity.
- *PartnerLink* field denotes the partner service that the element interacts with. It only exists in *IN* and its value is `name` attribute of `partnerLink` in the corresponding BPEL activity.
- *Condition* field denotes transition conditions (or predicate constraints) of the XBFG element. It exists in *EN*, *MN*, *CN* and *CE*, and its value is `condition` attribute of corresponding BPEL structural activity; but for the edge produced by `link`, its value is `transitionCondition` attribute of `link`.

- *Endpoint* field represents the binding address of XBFG element. It only exists in *SN* and its value is the endpoint address of the service.
- *PortType* field and *Operation* field stipulate the interactive interface between *IN* and *SN*. They only exist in *IN* and their values are `portType` and `operation` attributes in the corresponding BPEL activity.
- *InMsg* field and *OutMsg* field define the type of interactive messages between BPEL process and partner services where the former represents the message received by the process and the latter represents the sent out message. They only exist in *IN* and their value can only be acquired by analyzing the corresponding WSDL document of partner service since information stored in BPEL document is limited. More details are illustrated in part D of this section.

In order to analyze control flow to capture transition information of paths, it is necessary to add *transition condition* to those edges starting from *EDN*. For example, let *edn* denotes an *EDN*, if *edn* . *condition* = *c*, the values of *condition* of two outgoing edges from *edn* are *c* and ! *c*. In addition, if the value of attribute `condition` in `if` is *isKnown* ! = *true*, XBFG will be added with following three elements: (1) an *EDN* with *condition* value of *isKnown* ! = *true*; (2) one corresponding control edge with its *condition* value of *isKnown* ! = *true*; (3) the other control edge with its *condition* value of ! (*isKnown* ! = *true*).

4.2. XBFG construction

The process of XBFG construction consists of four steps:

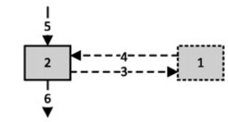
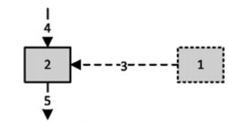
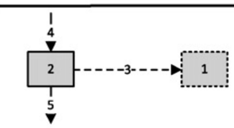
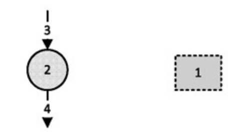
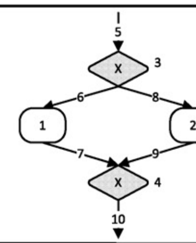
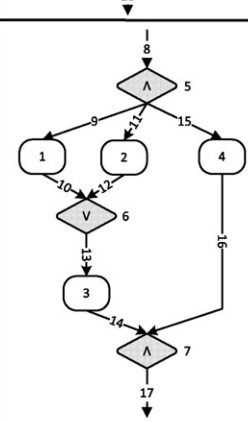
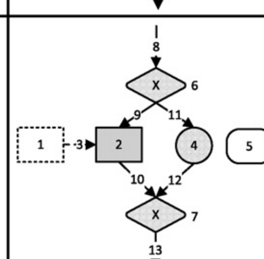
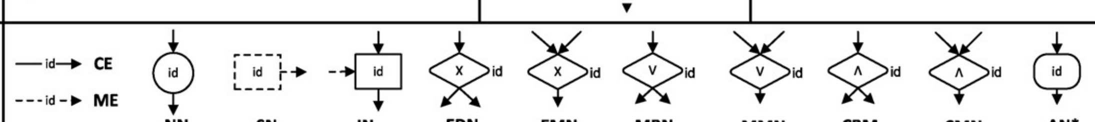
(1) Create *SN*s for all *partnerLinks*.
(2) Create other kinds of XBFG nodes for all BPEL activities.
(3) Create *ME* according to interactive relation between *IN* and *SN*.
(4) Create *CE* according to the relation of execution order among XBFG nodes.

All steps are based on an analysis of BPEL document by transforming all BPEL activities (including `partnerLink`) into XBFG nodes. The construction of XBFG edges is accompanied with the construction of XBFG nodes. The transformation methods are highlighted below and Fig. 4 gives some typical transformations of BPEL activity snippets for better understanding. We first give six transformation methods for basic activities (including `partnerLink`).

- `partnerLink`. A *SN sn* is created where the value of *sn* . *name* is attribute `name` of `partnerLink`. The value of *sn* . *endpoint* may be the value of sub-element `Address` in `partnerLink`. It can also be complemented by `assign` activity based on *EndpointReference* mechanism of WS-Addressing which is detailed in the transformation of `assign` activity.
- `invoke` activity. An *IN in* is created where the values of *in* . *name* and *in* . *partnerLink* are attributes `name` and `partnerLink` of `invoke`, respectively. Suppose the corresponding *SN sn* where *sn* . *name* = *in* . *partnerLink* exists, if the value of attribute `inputVaribale` in `invoke` is not empty, a *ME me* is created from *in* to *sn* where *me* . *source* = *in* and *me* . *target* = *sn*; If the value of attribute `outputVariable` in `invoke` is not empty, an *ME me*′ is created from *sn* to *in* where *me* . *source* = *sn* and *me* . *target* = *in*.
- `receive` activity. An *IN in* is created where the values of *in* . *name* and *in* . *partnerLink* are attributes `name` and `partnerLink` of `receive`, respectively. Suppose the corresponding *SN sn* where *sn* . *name* = *in* . *partnerLink* exists, an *ME me*′ is created from *sn* to *in* where *me* . *source* = *sn* and *me* . *target* = *in*.
- `reply` activity. An *IN in* is created where the values of *in* . *name* and *in* . *partnerLink* are attributes `name` and `partnerLink` of `reply`, respectively. Suppose the corresponding *SN sn* where *sn* . *name* = *in* . *partnerLink* exists, an *ME me* is created from *in* to *sn* where *me* . *source* = *in* and *me* . *target* = *sn*.
- `assign` activity. A *NN nn* is created where the value of *nn* . *name* is attribute `name` of sub-element `copy` in `assign`. If usage of this activity is partnerLink assignment, we can find the corresponding *SN sn* where *sn* . *name* is equal to attribute `partnerLink` of `copy` and update the value of *sn* . *endpoint* according to sub-element `address` in `copy`.
- `wait` activity. A *NN nn* is created where the value of *nn* . *name* is attribute `name` of `wait`.
- `empty` activity. No XBFG node is created.
- `rethrow` activity. No XBFG node is created.
- `extensionActivity` activity. No XBFG node is created.

For structural activities, we mainly focus on the transformation of the structure itself. So the transformations of basic activities that are embodied in structural activities are not given here. We describe eight transformation methods for structural activities below.

- `sequence` activity. Though no XBFG node is created here, we traverse its sub-elements sequentially to form the sequence relations of sub-elements. *CE* is used to connect them if there is a sequential relation between nodes corresponding to the two sub-elements.
- `scope` activity. It is processed the same way as the `sequence` activity.
- `if` activity. A pair of *EN edn* and *emn* is created where the value of *edn* . *name* and *edn* . *condition* are attributes `name` and `condition` of `If`, respectively. The first activity in `If` is set as the left child of *edn*. If sub-element `elseif` exists in `If`, a new pair of *EN edn*′ and *emn*′ is created for each `elseif` where the value of *edn*′ . *condition* is attribute `condition` of `elseif` and the whole pair is set as the right child of *edn*. In Fig. 5, the `elseif` whose condition is $p = v2$ becomes the right child of `If` whose condition is $p = v1$. When more `elseif`s are included, each is processed the same way as the first `elseif` and set as the right child of the prior `elseif`. In addition, the activity in `else` is considered as the right child of last `elseif` and all *CE ce*s should be created according to the branch relations.
- `while` activity. Only an *EDN edn* is created where the values of *edn* . *name* and *edn* . *condition* are attributes `name` and `condition` of `while`, respectively. In addition, a *CE ce* is created from the last XBFG node in `while` to *edn*.
- `forEach` activity. Only an *EDN edn* is created where the value of *edn* . *name* is attribute `name` of `forEach`. Mark attribute `counterName` of `forEach` as *cN*, `startCounterValue` as v_s, and `finalCounterValue` as v_f, then the value of *edn* . *condition* is $v_s \le cN \le v_f$. In addition, a *CE ce* is created from the last XBFG node in `forEach` to *edn*.
- `repeatUntil` activity. Only an *EDN edn* is created where the values of *edn* . *name* and *edn* . *condition* are attributes `name` and `condition` of `repeatUntil`, respectively. In addition, a *CE ce* is created from *edn* to the first XBFG node in `repeatUntil`.
- `pick` activity. A pair of *EN edn* and *emn* is created where the value of both *edn* . *name* and *emn* . *name* are attribute `name` of `pick`; An *IN in* is created for each sub-element `onMessage`, where the value of *in* . *partnerLink* is attribute `partnerLink` of `onMessage`. Suppose the corresponding *SN sn* where *sn* . *name* = *in* . *partnerLink* exists, a *ME me* is created from *sn* to *in*. In addition, a *NN nn* is created for each sub-element `onAlarm` in `pick`.
- `flow` activity. A pair of *CN cbn* and *cmn* is created where the value of both *cbn* . *name* and *cmn* . *name* are attribute `name` of `flow`; for all BPEL basic activities synchronized by sub-element `link` in `flow`, which can be usually transformed into *NN* or *IN*, check all their sub-elements `source` and `target`. On the one hand, if $s > 1$

	BPEL Activity Snippet	XBFG element	Field
Basic Activities	<partnerLink name="P" ... /> ... <**invoke** name="iName" partnerLink="P" portType="a:Pport" operation="O" inputVariable="X" outputVariable="Y"/>		The field of *invoke*: **ID**=<2, hashcode>, **Category**=IN, **Source**={4,5}, **Target**={3, 6}, **Name**=iName, **PartnerLink**=P **portType**=a:pPort, **operation**=O The field of *partnerLink*: **ID**=<1, hashcode>, **Category**=SN, **Source**={3}, **Target**={4}, **Name**=P
	<partnerLink name="P" ... /> ... <**receive** name="rName" partnerLink="P" portType="a:Pport" operation="O" Variable="X" />		The field of *receive*: **ID**=<2, hashcode>, **Category**=IN, **Source**={3,4}, **Target**={5}, **Name**=rName, **PartnerLink**=P **portType**=a:pPort, **operation**=O The field of *partnerLink*: **ID**=<1, hashcode>, **Category**=SN, **Source**={}, **Target**={3}, **Name**=P
	<partnerLink name="P" ... /> ... <**reply** name="rName" partnerLink="P" portType="a:Pport" operation="O" Variable="X" />		The field of *receive*: **ID**=<2, hashcode>, **Category**=IN, **Source**={4}, **Target**={3, 5}, **Name**=rName, **PartnerLink**=P **portType**=a:pPort, **operation**=O The field of *partnerLink*: **ID**=<1, hashcode>, **Category**=SN, **Source**={3}, **Target**={}, **Name**=P
	<partnerLink name="P" ... /> ... <**assign** name="cName"> <copy> <from><literal><service-ref><**EndpointReference**> <Address>http://www.istv.com/</Address> <ServiceName>servName</ServiceName> </EndpointReference></service-ref></literal></from> <to partnerLink="P"/> </copy> </assign>		The field of *assign(binding)*: **ID**=<2, hashcode>, **Category**=NN, **Source**={3}, **Target**={4}, **Name**=cName The field of *partnerLink*: **ID**=<1, hashcode>, **Category**=SN, **Source**={}, **Target**={}, **Name**=P **Endpoint**=http://www.istv.com/
Structural Activities	<**if** name="ifName"> <condition>P=v1</condition> <actType name="actNameA">...</actType> <else> <actType name="actNameB">...</actType> </else> </if> ***Note:The same is applied to activities <switch>, <while>, <repeatUntil> and <foreach>***		The field of *If is consisted of two parts*: (1) **ID**=<3, hashcode>, **Category**=EDN, **Source**={5}, **Target**={6, 8}, **Name**=ifName, **Condition**="P=v1" (2) **ID**=<4, hashcode>, **Category**=EMN, **Source**={7, 9}, **Target**={10} The field of edges starting from EDN: (1) **ID**=<6, hashcode>, **Category**=CE, **Source**={3}, **Target**={1}, **Condition**= "P=v1" (2) **ID**=<8, hashcode>, **Category**=CE, **Source**={3}, **Target**={2}, **Condition**= "P!=v1"
	<**flow** name="fName"> <links> <link name="AtoC"/> <link name="BtoC"/> </links> <actType name="A"> <source linkName="AtoC" transactionCondition="p=v1"/> </actType> <actType name="B"> <source linkName="BtoC" transactionCondition="p=v2"/> </actType> <actType name="C" joinCondition="getLinkStatus("AtoC") **OR** getLinkStatus("BtoC") "> <target linkName="AtoC"/> <target linkName="BtoC"/> </actType> <actType name="D">...</actType> </flow>		The field of *flow is consisted of three parts*: (1) **ID**=<5, hashcode>, **Category**=CBN, **Source**={5}, **Target**={9, 11, 15}, **Name**=fName (2) **ID**=<7, hashcode>, **Category**=CMN, **Source**={14, 16}, **Target**={17} (3) **ID**=<6, hashcode>, **Category**=MMN **Source**={10, 12}, **Target**={13} **Condition**=getLinkStatus("AtoC") **OR** getLinkStatus("BtoC") The field of edges ending at MMN: (1) **ID**=<10, hashcode>, **Category**=CE, **Source**={1}, **Target**={6}, **Name**=AtoC, **Condition**= "P=v1" (2) **ID**=<12, hashcode>, **Category**=CE, **Source**={2}, **Target**={6}, **Name**=BtoC, **Condition**= "P=v2"
	<partnerLink name="P" ... /> ... <**pick** name="pName"> <onMessage partnerLink="P" portType="a:Pport" oprtation="O" variable="X"> <actType name="A"></actType> </onMessage> <onAlarm for="T"> <actType name="B"></actType> </onAlarm> </pick>		The field of *pick is consisted of four parts*: (1) **ID**=<6, hashcode>, **Category**=EDN, **Source**={8}, **Target**={9, 11}, **Name**=pName (2) **ID**=<7, hashcode>, **Category**=EMN **Source**={9, 11}, **Target**={12} (3) **ID**=<2, hashcode>, **Category**=IN **Source**={9, 3}, **Target**={10} **PartnerLink**=P, **portType**=a:pPort, **operation**=O (4) **ID**=<4, hashcode>, **Category**=NN, **Source**={11}, **Target**={12}
Legend			

* Note: **AN** can be any type of basic or structural activity

Fig. 4. XBFG element construction for BPEL activities.

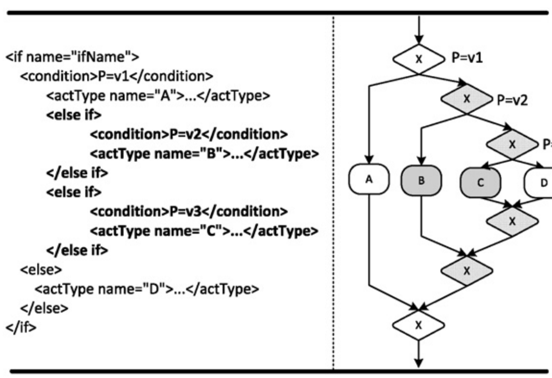


Fig. 5. Transformation for multi-`elseif` in `If` Activity.

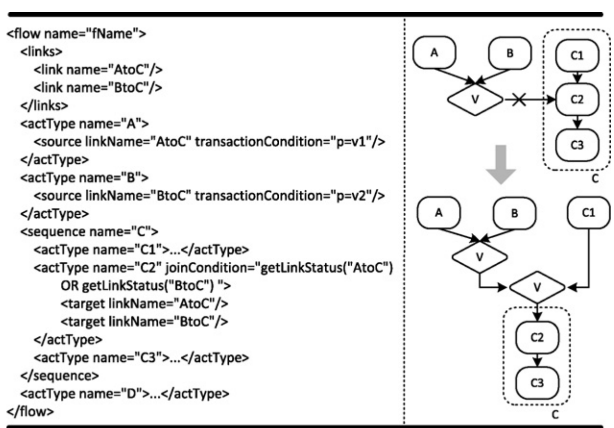


Fig. 6. Transformation for `Flow` Activity with `link`.

(*s* denotes the number of sub-element `target` of current activity *act*), namely, the node is the target node of many `links`, an *MN mn* is created at the merge point of these `links`. The category of *mn* is determined by the attribute `joinCondition` of the current activity. If the bool expression in `joinCondition` contains *AND* operator, *mn.category=CMN*. If the bool expression in `joinCondition` contains *OR* operator, *mn.category=MMN*. If there is no `joinCondition`, *mn.category=MMN*. It is noted that if *act* is a part of structural activity (`sequence` as an example) and *s>1*, a *CMN cmn* need to be created between *mn* and the subsequent node of *act*, as shown in Fig. 6. On the other hand, if $t > 1$ (*t* denotes the number of sub-elements `source` of current activity *act*), a *CBN cbn* is created between *act* and the subsequent node of *act*.

To save space, we will not give the detailed algorithms for the above 14 transformations. Fig. 7(a) shows the XBFG of *LCS* $v1.0$. The number attached near each XBFG element is its id in Field *ID*. The id corresponds to the BPEL code number (shown in bracket) in Fig. 1(a). We also list detailed *field* information of some typical XBFG elements in gray boxes. These information are extracted from BPEL code according to our transformation algorithms. The XBFG of *LCS* $v1.1$ is also shown in Fig. 7(b). The modification of `assign` activity in $v1.0$ is reflected in the newly generated XBFG node (with id 46) and edges (with ids 47 and 48).

4.3. XBFG path definition

Based on definition of XBFG model, the concept of XBFG path has to be customized and redefined accordingly. The BFG path definition (Yuan et al., 2006) is similar to the traditional CFG path, which is a sequence of nodes. However, this node sequence cannot support a comprehensive change impact analysis of BPEL-based composite service for the following reasons:

- BFG path mainly describes the sequential execution relations of BPEL activities in the composite service to be tested while no special path are included to describe the message exchange between BPEL process and partner services, which is a special feature of composite service.
- Relying solely on node sequences, BFG path cannot clearly depict many kinds of structures in BPEL process, such as *sequence*, *selection*, *loop*, *concurrency*, *synchronization*, etc.
- BFG path comparison, which is often node sequence comparison, cannot detect certain changes in the evolution in composite service such as change of constraint condition in control flow and change of message exchange between BPEL process and partner services.

Thus, it is necessary to take XBFG edges into account in the new path definition. Based on this consideration, XBFG path is constructed from the nodes and edges visited according to their orders in an execution of program. The execution order is implicitly encoded in *source* and *target* field of XBFG elements. As a result, all XBFG paths in BPEL process can be obtained by analyzing the information carried by XBFG elements.

4.3.1. XBFG path

To provide a clearer description of the XBFG path, we firstly define its two subtypes, namely, *in-process path* and *out-process path*, where the former depicts the internal behavior of the process, and the latter considers the interactive behaviors between process and partner services.

Definition 3 *(In-process path).* Let *ip* be a set that is composed of *XBFG Nodes* (except *SN*) and *CE*s. It is called a XBFG *in-process path* if the following conditions are satisfied:

- $\forall xe \in ip$, $xe = n \in N$ or $xe = ce \in CE$ where *N* is set of non-*SN Node*s and *CE* is the set of *CE*s.
- $\forall ce \in ip, \exists n \in ip \mapsto n \in ce\,.\,target$[2]

Definition 4 *(Out-process path).* Let *op* be a set that is composed of *SN*s and *ME*s. It is called a XBFG *out-process path* if the following conditions are satisfied:

- $\forall xe \in op$, $xe = sn \in SN$ or $xe = me \in ME$ where *SN* and *ME* are set of *SN*s and *ME*s, respectively.
- $\forall me \in op$, if *me* is not the last *ME* of *op*, $\exists sn \in SN \mapsto sn \in me\,.\,target$.

XBFG path combines both *in-process path* and *out-process path*, which can reflect the whole behavior of composite service. *XBFG path* is defined as follows:

Definition 5 *(XBFG path).* Let *xp* be a set that is composed of a *in-process path ip* and a set of *out-process paths* op_i $(1 \le i \le n)$. It is called a XBFG *path* if the following conditions are satisfied:

- $\forall xe \in xp$, $xe \in ip$ or $xe \in op_k (1 \le k \le n)$.
- $\forall op \in xp$, $\exists in$ and *me*, $in \in ip \wedge in \in IN$ and $me \in op \mapsto me \in in\,.\,target \vee me \in in\,.\,source$.

Each XBFG path can be started from the initial node of XBFG. There are two cases: (1) path begins with an *IN in* when the process

[2] We use notation $ce \in ip$ to denote that *ce* is a control edge of XBFG in-process path *ip* and $n \in ip$ denotes that *n* is a node of in-process path *ip*. Similar notation is used in Definitions 4 and 5.

begins with a start activity `receive` and will be activated by receiving a call message sent by partner service; (2) path begins with an *EN edn* when the process begins with start activity `pick`.

4.3.2. XBFG path generation

Now we discuss how to generate XBFG path using the information of *source* and *target* fields recorded in XBFG elements. It is feasible to find an XBFG path by traversing XBFG and composing in-process paths and corresponding out-process paths. The whole generation process is depicted in Algorithm 1.

Algorithm 1. XBFG path generation.

```
Input p[count]: current XBFG path to be processed;
Input e: current XBFG element to be processed;
Output p[count]: all XBFG paths to be generated;
Variable op[mcount]: all XBFG out-process paths generated;
ProcessPath(p[count], e)
//termination condition of recursion
if e == null then
  return
end if
if e.category == IN then
  p[count] = p[count] ∪ {e};
  for each e_i ∈ e.source or e.target do
    if e_i.category == ME then
      mcount + +;
      create a new XBFG out-process path op[mcount];
    end if
    //create out-process path from ME
    CreateOP(op[mcount], e_i);
    p[count] = p[count] ∪ op[mcount];
  end for
  for each e_i ∈ e.target do
    if e_i.category! = ME then
      ProcessPath(p[count], e_i);
    end if
  end for
else if e.category == EDN then
  p[count] = p[count] ∪ {e}
  for each e_i ∈ e.target do
    if e_i ∉ p[count]
      pTemp = p[count];
      if i > 1 then
        count + +;
        create a new XBFG path p[count];
        P[count] = pTemp;
      end if
    end if
    p[count] = p[count] ∪ {e_i};
    ProcessPath(p[count], e_i)
  end for
else if e.category == MBN then
  p[count] = p[count] ∪ {e};
  compute all combinations of e.target as set CS
  for each cs_i ∈ CS do
    pTemp = p[count];
    if i > 1 then
      count + +;
      create a new XBFG path p[count];
      p[count] = pTemp;
    end if
  end for
  for each element e_i ∈ cs_i do
    p[count] = p[count] ∪ {e_i}
    ProcessPath(p[count], e_i);
  end for
else if e.category == CBN then
  p[count] = p[count] ∪ {e}
  for each successor e_i ∈ e do
    p[count] = p[count] ∪ {e_i}
    ProcessPath(p[count], e_i);
  end for
else if e.category == default then
  p[count] = p[count] ∪ {e}
  ProcessPath(p[count], e.target);
end if
```

Before executing the XBFG path generation algorithm, all *CE ces* that are originated from the start element *s* must be found. If the number of *ces* is *m*, *m* paths $p[i]$ $(1 \le i \le m)$ are created where *s* is included in all paths and ce_i in corresponding path $p[i]$.

Suppose $p[count]$ is the current XBFG path to be generated, where *count* is a global variable for distinguishing the generated paths. The format of $p[count]$ is a sequence of XBFG elements, $p[count] = s \cdot ce_{count} \cdot n \cdot e \cdot \ldots$ (n and e are XBFG node and edge, respectively).

Algorithm 1 works on different types of XBFG elements as follows:

- If the current element *e* is *IN*, it is added into $p[count]$ at first. Then out-process path *op* will be generated as follows: (1) Check *source* (*target*) field of *e* to find all *ME mes* which are connected to *e*; (2) Traverse backward (forward) along each *me* based on its *source* (*target*) field till an *IN in* has been visited (here *in* may be the same as *e*); (3) Add all traversed *ME mes* and *SN sns* into *op*. Finally, all generated *ops* are added into $p[count]$.
- If *e* is *EDN* with *k* branches, copy $p[count]k-1$ times for generating another $k-1$ paths where each path represents a branch of the execution path. Then add *e* and all *CE* starting from *e* into $p[count]$ and the other copied paths.
- If *e* is *MBN*, suppose that the out-degree of *e* is *k*, which means the edges that are executed at the same time are at most *k*, there are at most $CS = C_k^1 + C_k^1 + \cdots + C_k^k = 2^k - 1$ kinds of execution order after *e* has executed. So copy $p[count]$ $2^k - 2$ times for generating another $2^k - 2$ paths and add *e* into all $2^k - 1$ paths. Then each path represents one of the $2^k - 1$ kinds of execution order. For example, if a *MBN mbn* has three out-going edges, i.e., e_1, e_2 and e_3 $(k = 3)$, seven paths $(2^3 - 1)$ will be composed of $p[i]$: $\{mbn, e_1\}$, $\{mbn, e_2\}$, $\{mbn, e_3\}$, $\{mbn, e_1, e_2\}$, $\{mbn, e_1, e_3\}$, $\{mbn, e_2, e_3\}$, $\{mbn, e_1, e_2, e_3\}$.
- If *e* is *CBN*, we can use reachability testing (Lei and Carver, 2006) to generate synchronization sequences within concurrent block and further generate many execution paths, but these paths have the same execution conditions and expected output for a given input data. Therefore we can regard them as one path. Based on this consideration, we add only *e* and all *CE* starting from *e* into $p[count]$.
- If *e* is *NN* or other kinds of elements such as *CE* (marked as *default* in Algorithm 1, *ME* and *SN* are not included here for they are considered in the first *if-then-else* block), *e* is added into $p[count]$ directly.

For the recursive Algorithm 1, we assume that the number of XBFG elements to be processed is *n*. Suppose the time performance of this algorithm for *n* XBFG elements is $O(n)$, we have $O(n) = (n-1) * O(n-1)$ (We choose the most time-consuming block to compute the recursive function). So the time complexity of Algorithm 1 is $O(n) = n!$.

Fig. 7 lists all *XBFG paths* calculated for both *v*1.0 and *v*1.1 of LCS. Field *ID*.*id* is used as the representation of each XBFG element. Take *v*1.1 as an example, the path computation begins with the start node (XBFG element 5); it traverses the XBFG according to the *source* and *target* information of each XBFG element. The arrow lines with number show the traverse sequence of the whole XBFG. When we meet the *IN* (XBFG element 5), an *out-process* is created containing XBFG element 6, 1 and 28. The same handling is also applied for XBFG element 8, 13 and 17. When we meet the *CN* (XBFG element 21), a specific order for concurrent branches is chosen at random. Here the left branch is selected first. When we meet the *EN* (XBFG element 25), a replicated XBFG path is created for handling the branch statement. According to Algorithm 1, two XBFG paths are

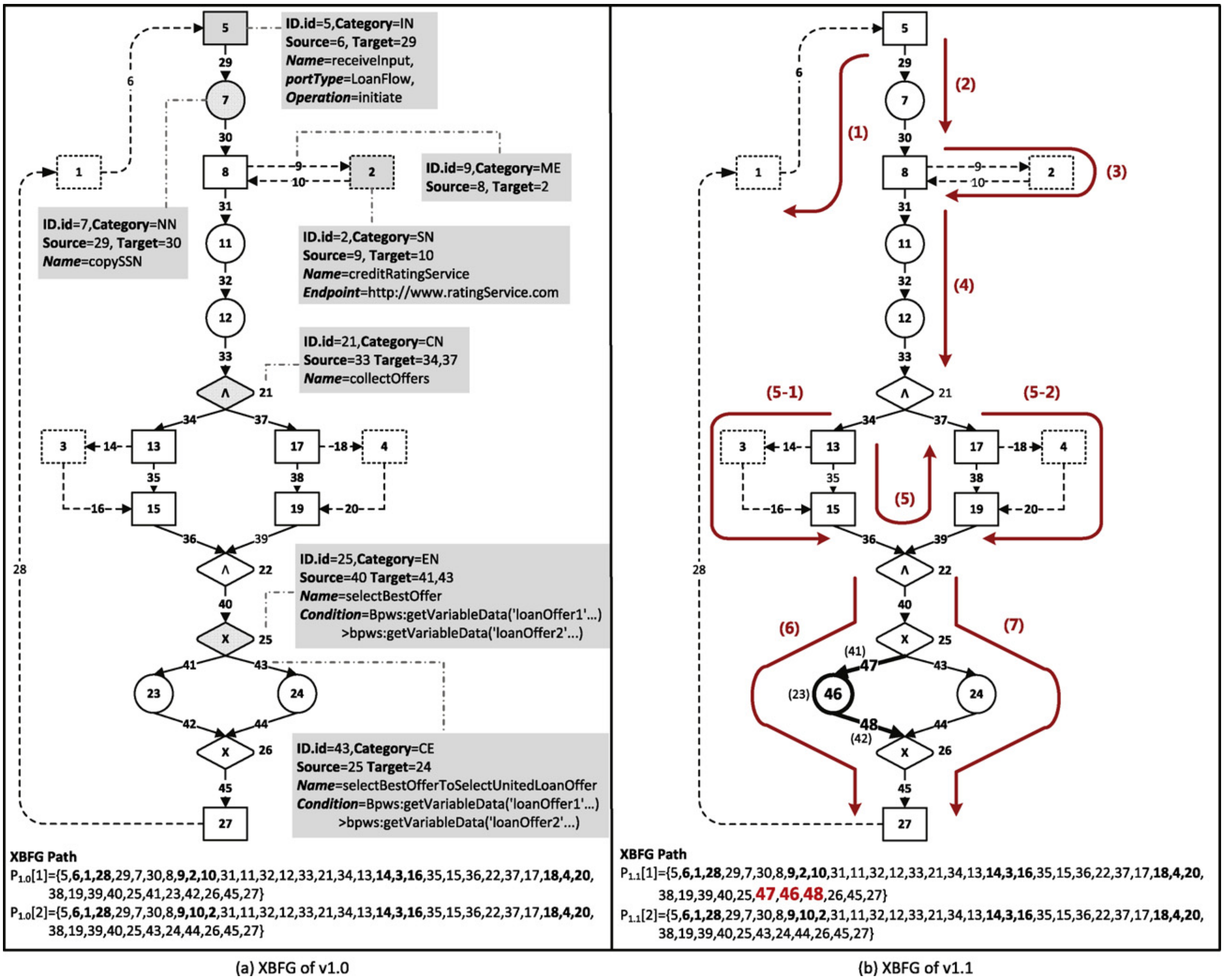


Fig. 7. The graphical XBFG of loan service of Version 1.0 and Version 1.1.

generated finally. Each XBFG path contains four *out-process* paths, which are marked in bold in the figure.

4.4. XBFG message sequence

An XBFG path actually represents an execution trace of the composite service. It records not only the behavior of inner-process but also the communication between the process and the invoked services. However, the detailed information of such communication depicted by BPEL document is limited. When the process invokes a partner service, for example, only the information about the interface of partner service and message exchange pattern (MEP) can be acquired from the BPEL document. We cannot get the concrete type of input (output) variables that are delivered between services.

In Fig. 8, the $v1.0$ of *LCS* is represented by XBFG with all *IN*s shown in gray. The BPEL code snippets that are used for depicting the service interaction are attached with each related *IN*. The definition of `partnerLink` in BPEL is also provided. Based on the analysis of BPEL document, we can only find out that the process defined in *LoanFlow.bpel* interacts with the partner services whose exposed interfaces are stored in *LoanService.wsdl* and *CreditRatingService.wsdl* by In-Out (or Request-Response) MEP. So the order of input and output messages exchanged between services can be used to reflect the interactive behavior, corresponding to `inputVariable` and `outputVariable` in BPEL. By parsing the BPEL document, a variable sequence can be obtained as shown in the middle section of Fig. 8. However, the concrete type of each variable is unknown as this information is not available from BPEL. So it is necessary to import corresponding WSDL documents that contain complete definition of `message` type to extract the missing information. `Message` defined in WSDL document contains two attributes, `name` and `part` where the former represents the name of message type and the latter represents the concrete type of message. We define *message* as follows:

Definition 6 *(Message).* Message is defined as a triple $M = \langle N, T, IN\rangle$ where N is a string that represents the name of message, T is the concrete type of message. *IN* is a bool variable that represents the message transfer direction where *true* represents M is an *In-Message* and *false* represents M is *Out-Message*.

Here, T can be either an XSD (XML Schema Definition) built-in type or user-defined complex type. *IN-Message* is used to represent the message received by the process while *Out-Message* is used to represent the message delivered from the process to partner service.

Fig. 8 depicts the procedure to obtain the full definition of `message` by locating and parsing the corresponding WSDLs of partner services or composite services. We take the `inputVariable` cnInput that is contained in the first `invoke` activity (corresponding to the XBFG element with ID.id 8) as an example to illustrate how to find the true type of the messages. Through the name of `partnerLink`, `portType` and `operation` in the same `invoke` activity, we can locate the complete definition of `message` CreditRatingServiceRequest in *CreditRatingService.wsdl* which contains the type

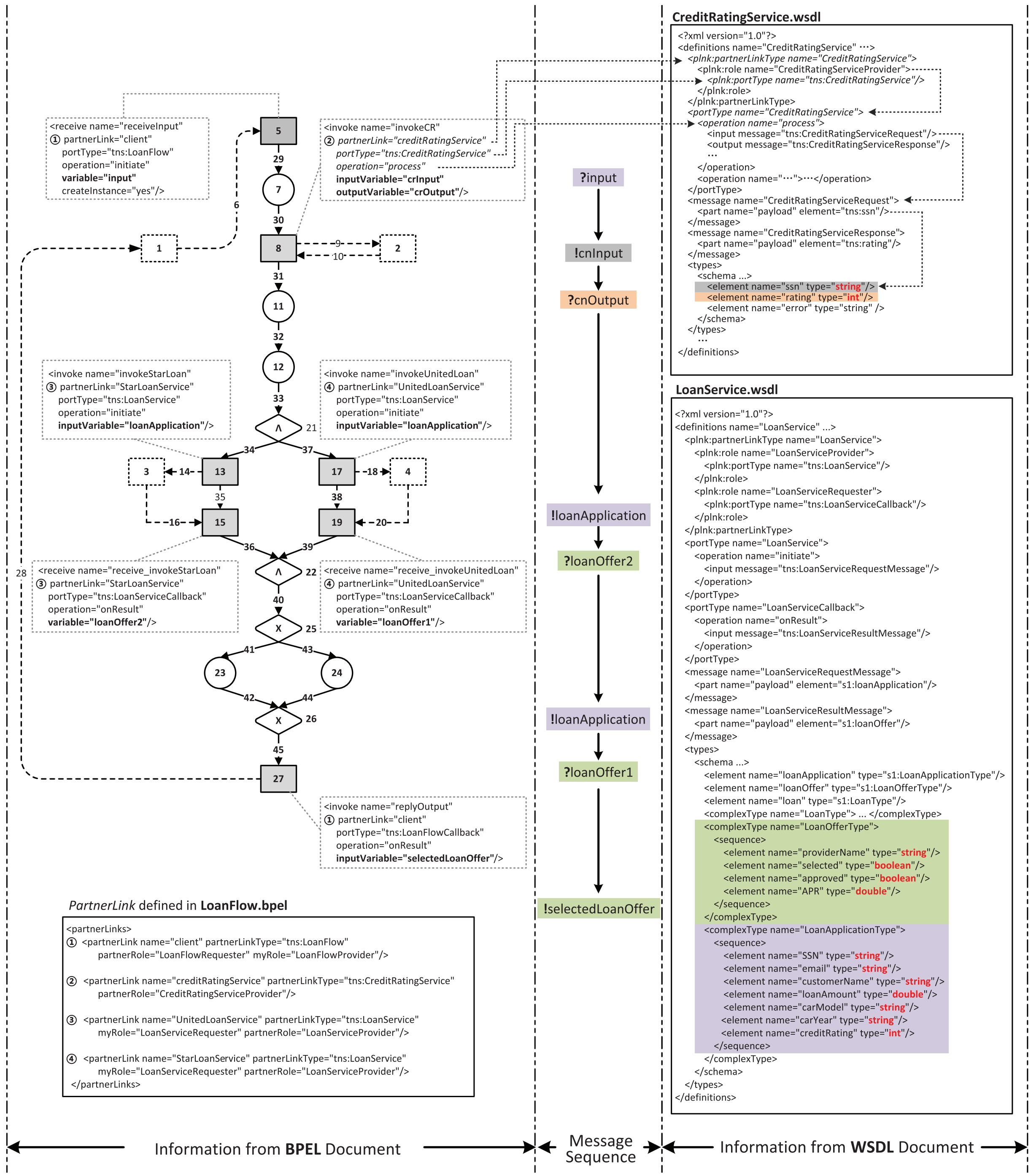


Fig. 8. The process of generating the message sequence from BPEL and WSDL document.

specification of cnInput. The dashed arrow shows the process of locating message in Fig. 8. Then, by further analysis of the `part` in the located `message` CreditRatingServiceRequest, we can get the type information that cnInput is a string in XSD build-in types where both of them are shown in gray background. All related information of `inputVariable` cnInput is listed in the second row of Table 3. The other `message`s and their concrete type are also marked in the same color in Fig. 8 and their detailed information is also listed in Table 3. (For interpretation of the references to color in text, the reader is referred to the web version of this article.)

It must be noted that the parsed message types could be used to supplement the fields *InMsg* and *OutMsg* of each *IN* which will be important for *XBFG message sequence* generation later.

An *XBFG message sequence* is attached to each *XBFG path* as a supplementary description of interactive behavior. It records the concrete message streams delivered between process and partner services. *Message sequence* is defined as follows:

Definition 7 *(Message sequence).* Let *ms* be a set that is composed of *messages*. It is called a *message sequence* for the corresponding XBFG path *p* if the following conditions are satisfied:

- $\forall m \in ms$, $\exists xe \in p$ and $xe \in IN$, $m = xe\,.\,InMsg$ or $m = xe\,.\,OutMsg$

The process to generate *XBFG message sequence* is not very complex. For each *XBFG path*, we search all the *IN*s in this path and

Table 3
Detailed message type of message sequence of $v1.0$.

Variable	PartnerLink	PortType	Operation	Message	Type
?input	client	LoanFlow	initiate	LoanFlowRequestMessage	(String, Boolean, Boolean, Double)
!crInput	creditRatingService	CreditRatingService	process	CreditRatingServiceRequestMessage	String
?crOutput	creditRatingService	CreditRatingService	process	CreditRatingServiceResponseMesage	Int
!loanApplication	StarLoanService	LoanService	initiate	LoanServiceRequestMessage	(String, Boolean, Boolean, Double)
?loanOffer2	StarLoanService	LoanServiceCallback	onResult	LoanServiceResultMessage	(String, String, String, Double, String, String, Int)
!loanApplication	UnitedLoanService	LoanService	initiate	LoanServiceRequestMessage	(String, Boolean, Boolean, Double)
?loanOffer1	UnitedLoanService	LoanServiceCallback	onResult	LoanServiceResultMessage	(String, String, String, Double, String, String, Int)
!selectedLoanOffer	client	LoanFlowCallback	onResult	LoanFlowResultMessage	(String, String, String, Double, String, String, Int)
Message sequence:	?(String,Boolean,Boolean,Double), !String, ?Int, !(String,Boolean,Boolean,Double), ?(String, String, String, Double, String, String, Int), !(String, Boolean, Boolean, Double), ?(String, String, String, Double, String, String, Int),!(String, String, String, Double, String, String, Int)				

collect the information of fields *InMsg* and *OutMsg* according to the occurrence order of *IN*. Table 3 shows the message sequence for both XBFG paths in $v1.0$ of *LCS* where "?" represents the *IN-Message* type and "!" represents the *Out-Message* type. Detailed description of each message in this message sequence is shown in this table where the type with parentheses represents user-defined complex type and the one without parentheses represents XSD build-in type.

5. Test case selection

In this section, we will discuss how to select test cases for a change of composite service. Any change of composite service can be reflected in the change of XBFG paths. For example, *process change* can be detected by comparing the fields of XBFG elements in different versions. *Binding change* can be found by checking the *endpoint* filed of corresponding *SN*s. So the problem of test case selection is transformed into a XBFG comparison problem which is consisted of *XBFG path comparison* and corresponding *message sequence comparison* where the former covers the *process change* and *binding change* while the latter covers the *interface change*. The goal of XBFG comparison is to identify the *modified path* that is affected by the service modification. *Modified path* involves two kinds of paths, one is *old path* on which test cases in baseline version can be selected to re-run, and the other is *new path* which requires new test cases to be generated. After the comparison, *path condition analysis* is introduced to reduce the number of generated test cases for *new path* since many test cases in the baseline version can be adopted if the path condition of *new path* equals to that of an *old path*.

To define the test case selection, some formal notations are introduced first.

5.1. Notations and basic ideas

Let $S[1], \ldots, S[i], \ldots, S[n]$ denote n different versions of a composite service, respectively, and $\Delta S[i]$ denote the change from $S[i]$ to $S[i+1]$, then we have

$$S[i+1] = S[i] + \Delta S[i] \quad (1 \le i \le n-1)$$

Let $G[i]$ be the XBFG model generated from $S[i]$ and $\Delta G[i]$ denote the change from $G[i]$ to $G[i+1]$, then according to the mapping relation from composite service to XBFG we have

$$G[i+1] = G[i] + \Delta G[i] \quad (1 \le i \le n-1)$$

Furthermore, let $\Delta S[i]_{pc}$, $\Delta S[i]_{bc}$ and $\Delta S[i]_{ic}$ denote *process change*, *binding change* and *interface change* from S_i to S_{i+1}, respectively, then we have

$$\Delta S[i] = \Delta S[i]_{pc} \cup \Delta S[i]_{bc} \cup \Delta S[i]_{ic}$$

Accordingly, let $\Delta G[i]_{pc}$, $\Delta G[i]_{bc}$ and $\Delta G[i]_{ic}$ denote corresponding types of changes from $G[i]$ to $G[i+1]$, respectively, in XBFG model, then we have

$$\Delta G[i] = \Delta G[i]_{pc} \cup \Delta G[i]_{bc} \cup \Delta G[i]_{ic}$$

Suppose $P[i]$ represent all XBFG paths of $G[i]$ where $P[i].size$ represents the number of $P[i]$ paths. For each XBFG path $p[i] \in P[i]$, it is consisted of a *in-process path* $inp[i]$ and a set of *out-process paths* $outp[i]_k (0 \le k \le n)$. We define *out-process path set* $outp[i] = outp[i]_1 \cup \cdots \cup outp[i]_n$. So we have $p[i] = inp[i] \cup outp[i]$. Let $ms[i]$ be the message sequence for $p[i]$, we have

$$ms[i] = \bigcup_{\forall e \in p[i]}^{e.category=IN} (e.InMsg \cup e.OutMsg)$$

Suppose $P[i+1]$ represent all XBFG paths of $G[i+1]$ and $P[i+1]^s$ is the *modified path* of $G[i+1]$. Obviously, we have $P[i+1]^s \subseteq P[i+1]$. Let $P[i+1]^s_{pc}$, $P[i+1]^s_{bc}$ and $P[i+1]^s_{ic}$ denote the set of XBFG paths of $G[i+1]$ influenced by *process change*, *binding change* and *interface change*, respectively, then we have

$$P[i+1]^s = P[i+1]^s_{pc} \cup P[i+1]^s_{bc} \cup P[i+1]^s_{ic}$$

Consider $S[i]$ and $S[i+1]$ are two versions of composite services where $P[i]$ is all XBFG paths of $G[i]$ and $P[i+1]$ is all XBFG paths of $G[i+1]$. For some XBFG path $p_k[i] \in P[i]$ and the corresponding path $p_k[i+1] \in P[i+1]$, there exist three *path relations* as follows:

- $p_k[i+1]$ is *control equals* with $p_k[i]$ iff for each XBFG element $e[i] \in p_k[i]$ and corresponding element $e[i+1] \in p_k[i+1]$, $e[i] = e[i+1]$, which is marked as $p_k[i+1] \overset{c}{=} p_k[i]$.
- $p_k[i+1]$ is *message equals* with $p_k[i]$ iff for each message type $m[i] \in ms_k[i]$ and corresponding element $m[i+1] \in ms_k[i+1]$, $m[i] = m[i+1]$, which is marked as $p_k[i+1] \overset{m}{=} p_k[i]$.
- $p[i+1]$ is *equals* with $p[i]$ iff $p[i+1] \overset{c}{=} p[i]$ and $p[i+1] \overset{m}{=} p[i]$ which is marked as $p[i+1] \equiv p[i]$.

Now, the three kinds of change type mentioned in Section 3.2 can be defined in terms of the *path relation*s.

- *Process Change* happens when the following alternative conditions are satisfied:

Table 4
Test tables for test case selection.

(a) Mapping from one test case to multi XBFG paths

Test case	XBFG paths
$t_\alpha[i]$	$P_\alpha[i]=\{p_a[i], p_b[i], \cdots, p_m[i]\}$
$t_\beta[i]$	$P_\beta[i]=\{p_p[i], p_q[i], \cdots, p_z[i]\}$
...	...

(b) Mapping from one XBFG path to multi test cases

XBFG path	Test cases
$p_a[i]$	$T_a[i]=\{t_\alpha[i], t_\gamma[i], \cdots, t_\mu[i]\}$
$p_b[i]$	$T_b[i]=\{t_\beta[i], t_\delta[i], \cdots, t_\varphi[i]\}$
...	...

– $P[i+1].size \neq P[i].size$
– $\exists p_k[i+1] \in P[i+1], p_k[i] \in P[i]$, s.t. $p_k[i+1] \overset{c}{\neq} p_k[i]$.

- *Binding Change* happens when $\exists e_l[i+1] \in outp_k[i+1] \in p[i+1] \in P[i+1]$, $\exists e_l[i] \in outp_k[i] \in p[i] \in P[i]$, s.t. $e_l[i+1].partnerLink \neq e_l[i].partnerLink$
- *Interface Change* happens when $\exists p_k[i+1] \in P[i+1]$, $p_k[i] \in P[i]$, s.t. $p_k[i+1] \overset{m}{\neq} p_k[i]$

Since all three types of changes can be reflected in the changes of XBFG paths, there must exist a mapping φ that satisfies:

$$P[i+1]^s_{pc} = \varphi(\Delta P[i]_{pc})$$

$$P[i+1]^s_{bc} = \varphi(\Delta P[i]_{bc})$$

$$P[i+1]^s_{ic} = \varphi(\Delta P[i]_{ic})$$

As discussed before, *modified paths* $P[i+1]^s$ may come from *old paths* in $G[i]$, denoted as $P[i+1]^{so}$, and *new paths* in $G[i+1]$, denoted as $P[i+1]^{sn}$. Suppose $T[i]$ is the test suite of $S[i]$, $T[i+1]$ consists of two groups: $T[i+1]^{so}$ that re-test $P[i+1]^{so}$ selected from $T[i]$ and $T[i+1]^{sn}$ that test $P[i+1]^{sn}$ which need to be newly generated. So we have

$$T[i+1] = T[i+1]^{so} \cup T[i+1]^{sn}$$

where $T[i+1]^{so} \in T[i]$.

For each test case $t_k[i] \in T[i]$, $P_k[i]$ is a path set that $t_k[i]$ covers, where $P_k[i] \in P[i]$. This mapping relation can be described by a test table (Benedusi et al., 2002) as shown in Table 4(a). We can also easily obtain the reverse mapping from XBFG path to test cases, as shown in Table 4(b). For each XBFG path $p_k[i] \in P[i]$, a test suite $T_k[i]$ is attached to test $p_k[i]$ where $T_k[i] \in T[i]$.

Therefore, there should be a test suite $T_k[i+1]^{so}$ for each path $p_k[i+1]^{so}$ to be retested in $G[i+1]$, where $T_k[i+1]^{so} \in T[i+1]$. So we have

$$T[i+1]^{so} = \bigcup_{\forall p_k[i+1]^{so}}^{p_k[i+1]^{so} \in P[i+1]^{so}} T_k[i+1]^{so}$$

For each XBFG path $p_k[i+1]^{so} \in P[i+1]$, we can find a corresponding XBFG path $p_j[i]$ that $p_j[i] \equiv p_k[i+1]$. So the test suite $T_k[i+1]$ for $p_k[i+1]^{so}$ can be totally obtained from test suite $T_j[i]$ for $p_j[i]$. That is,

$$T[i+1]^{so} = \bigcup_{\forall p_k[i+1]^{so} \in P[i+1]^{so}, \exists p_j[i] \in P[i]}^{p_j[i] \equiv \in P_k[i+1]^{so}} T_j[i]^{so}$$

The steps for performing test case selection on $S[i+1]$ against $S[i]$ are as follows:

1) **XBFG path comparison**. The main task is to compare the paths in $P[i]$ and $P[i+1]$ one by one to get $P[i+1]^s_{pc}$ and $P[i+1]^s_{bc}$. Some of the paths in $P[i+1]^s_{pc}$ are *old paths* in $P[i]$, denoted as $P[i+1]^{so}_{pc}$, others are *new paths*, denoted as $P[i+1]^{sn}_{pc}$. All the paths in $P[i+1]^s_{bc}$ are the same as paths in $P[i]$. So move all members in $P[i+1]^{so}_{pc}$ and $P[i+1]^s_{bc}$ into $P[i+1]^{so}$, and all members of $P[i+1]^{sn}_{pc}$ into $P[i+1]^{sn}$.
2) **Message sequence comparison**. The main task is to compare the interfaces of each path in $P[i+1]^{so}$ with the corresponding path in $P[i]$ one by one. If some of the interfaces are found different, which indicates that new test cases are required for testing these paths, the corresponding paths need to be moved to $P[i+1]^{sn}$. Then, do comparison on the interfaces of each path in $\{P[i+1] - P[i+1]^{so} - P[i+1]^{sn}\}$ with the corresponding path in $P[i]$ (e.g. $\forall p[i+1] \in P[i+1] - P[i+1]^{so} - P[i+1]^{sn}$ and corresponding $p[i] \in P[i]$, $p[i+1] \overset{c}{=} p[i]$) to find out paths with interfaces change and move them to $P[i+1]^{sn}$.
3) **Path condition analysis**. The main task is to generate path condition for $P[i]$ and $P[i+1]^{sn}$ which has been updated in step (2). If $\exists\, p_k[i+1] \in P[i+1]^{sn}$ and $\exists\, p_l[i] \in P[i]$ where path condition of $p_k[i+1]$ is the same as one of $p_l[i]$, test suite $T_l[i]$ that is attached to $p_l[i]$ can also be used as the test suite for $p_k[i+1]$. So move all $p_k[i+1]$ from $P[i+1]^{sn}$ to $P[i+1]^{so}$.
4) **Test case selection**. The main task is to select test cases from $T[i]$. We can search all paths $p_k[i+1] \in P[i+1]^{so}$ in *test table* to find all re-usable test cases and add them into $T[i+1]^{so}$.

The following three sections will present the first three steps in more details.

5.2. XBFG path comparison

The purpose of *XBFG path comparison* is to find the path affected by *process change* and *binding change*. The path that has *process change*, namely *new path*, should be moved to $P[i+1]^{sn}$. And the path that has *binding change* but no *process change*, namely *old path*, should be moved to $P[i+1]^{so}$.

As each XBFG element has an identity field *ID*, the set of modified elements can be obtained by comparing *ID* of each element e in $N[i]$ (the set of elements in $G[i]$) and $N[i+1]$ (the set of elements in $G[i+1]$).

There are two cases for modified elements:

- If element $e \notin N[i] \wedge e \in N[i+1]$, we regard e as a new element that has been added into the new version.
- If element $e \in N[i] \wedge e \notin N[i+1]$, we regard e as an element that has been deleted from the old version.

Let $N[i+1]_{add}$ be a set of new elements added to $N[i+1]$ and $N[i+1]'_{add}$ be copy of $N[i+1]_{add}$ without elements whose *category* field is *SN* or *ME*. For each element $e_{add} \in N[i+1]_{add}$, search all the paths which contain element e_{add} and move them into $P[i+1]^s$. For each element $e'_{add} \in N[i+1]'_{add}$, search all the paths which contain element e'_{add} and move them into $P[i+1]^{sn}$. So we can get $P[i+1]^{so} = P[i+1]^s - P[i+1]^{sn}$. Let $N[i]_{del}$ be a set of elements deleted from $N[i]$. For each $p[i] \in P[i]$, let $N^{p[i]}_{del}$ denote the set of all elements deleted from $p[i]$, then we have $N[i]_{del} = \cup_{\forall p[i] \in P[i]} N^{p[i]}_{del}$. If $p[i+1] = p[i] - N^{p[i]}_{del}$ and $p[i+1] \in P[i+1]$, add $p[i+1]$ into $P[i+1]^{sn}$. Algorithm 2 describes the process for computing $P[i+1]^{so}$ and $P[i+1]^{sn}$. Suppose the number of XBFG elements in $p[i]$ is n and in $p[i+1]$ is m (n and m are in the same order of magnitude since the number of modification from the old version to the new version should be limited), the time complexity of Algorithm 2 is $O(2*n)+O(n^2)+O(n^2)+O(n^2) \approx O(n^2)$.

Algorithm 2. Path comparison algorithm.

Input $P[i]$, $P[i+1]$: set of paths to be compared;
Input $N[i]$, $N[i+1]$: set of elements in $P[i]$ and $P[i+1]$;
Output $P[i+1]^{so}$: set of *old paths*;
Output $P[i+1]^{sn}$: set of *new paths*;
PathComparison($P[i]$, $P[i+1]$, $N[i]$, $N[i+1]$)
Let $N_{all} = N[i] \cup N[i+1]$
for each element $e \in N_{all}$ **do**
  if $e \notin N[i] \wedge e \in N[i+1]$ **then**
    $N[i+1]_{add} = N[i+1]_{add} \cup \{e\}$
    if $n.category! = SN \& \& n.category! = ME$) then
      $N[i+1]'_{add} = N[i+1]'_{add} \cup \{e\}$
    end if
  else if $e \in N[i] \wedge e \notin N[i+1]$ **then**
    $N[i]_{del} = N[i]_{del} \cup \{e\}$
  end if
end for
for each element $n_{add} \in N[i+1]_{add}$ **do**
  for each element of $p[i+1] \in P[i+1]$ **do**
    if $n_{add} \in p[i+1]$ **then**
      $P[i+1]^{s} = P[i+1]^{s} \cup \{p[i+1]\}$
    end if
  end for
end for
for each element $n'_{add} \in N[i+1]'_{add}$ **do**
  for each element of $p[i+1] \in P[i+1]$ **do**
    if $n'_{add} \in p[i+1]$ **then**
      $P[i+1]^{sn} = P[i+1]^{sn} \cup \{p[i+1]\}$
    end if
  end for
end for
$P[i+1]^{so} = P[i+1]^{s} \setminus P[i+1]^{sn}$
for each $p[i] \in P[i]$ **do**
  for each $n[i]_{del} \in N[i]_{del}$ **do**
    if $n[i]_{del} \in p[i]$ **then**
      $N^{p[i]}_{del} = N^{p[i]}_{del} \cup \{n[i]_{del}\}$
    end if
  end for
  if $p[i+1] = (p[i] - N^{p[i]}_{del}) \in P[i+1]$
    $P[i+1]^{sn} = P[i+1]^{sn} \cup \{p[i+1]\}$
  end if
end for
Return $P[i+1]^{so}$, $P[i+1]^{sn}$

5.3. Message sequence comparison

The purpose of *message sequence comparison* is to find the XBFG path affected by *interface change* as *XBFG path comparison* can only find those influenced by *process change* and *binding change*. As is discussed in Section 4.4, *message sequence comparison* is actually a procedure of comparing message types of each XBFG path. Let $p[i]$ be a path of $G[i]$ and $p[i+1]$ be a path of $G[i+1]$. Consider the path pair $p[i]$ and $p[i+1]$, each of which is composed of the same XBFG elements. After *XBFG path comparison*, we have $p_k[i+1] \stackrel{c}{=} p_k[i]$. Let $ms[i]$ and $ms[i+1]$ denote *message sequence* of $p[i]$ and $p[i+1]$, respectively. Suppose $ms[i] = m_1[i]m_2[i] \ldots m_i[i] \ldots m_n[i]$ where $m_i[i]$ represents the *message* in *message sequence* of $p[i]$. Then the *message sequence comparison* can be performed by checking the type of all message pairs $m_k[i]$ and $m_k[i+1]$ from $ms[i]$ and $ms[i+1]$, respectively. The comparison will terminate if the type of one message is different from that of the other one in the pair. If the result indicates that $ms[i]$ is not equal to $ms[i+1]$, we need to move the corresponding XBFG path $p[i+1]$ to $P[i+1]^{sn}$.

Algorithm 3 presents the details of message sequence generation and comparison, where the message sequence generation is based on Section 4.4. Suppose the number of XBFG elements in $p[i]$ is n, the time complexity of Algorithm 3 is $O(n)$ obviously.

Algorithm 3. Message sequence generation and comparison algorithm.

Input $p[i]$, $p[i+1]$: two XBFG paths to be compared based on message sequences
Output *result*: comparison result of $p[i]$ and $p[i+1]$
MessageSequenceComparison($p[i]$, $p[i+1]$)
// Message Sequence Generation
for each element $e[i] \in p[i]$ **do**
  if $e[i].category == IN$ **then**
    if $e[i].InMsg! = null$ **then**
      $ms[i] = ms[i] \cup \{e[i].InMsg\}$
    end if
    if $e[i].OutMsg! = null$ **then**
      $ms[i] = ms[i] \cup \{e[i].OutMsg\}$
    end if
  end if
end for
for each element $e[i+1] \in p[i+1]$ **do**
  if $e[i+1].category == IN$ **then**
    if $e[i+1].InMsg! = null$ **then**
      $ms[i+1] = ms[i+1] \cup \{e[i+1].InMsg\}$
    end if
    if $e[i+1].OutMsg! = null$ **then**
      $ms[i+1] = ms[i+1] \cup \{e[i+1].OutMsg\}$
    end if
  end if
end for
// Message Sequence Comparison
$result = false$
for each message pair ($m_k[i] \in ms[i]$, $m_k[i+1] \in ms[i+1]$) **do**
if $m_k[i]! = m_k[i+1]$
  $result = false$
  break
end if
end for
return *result*

5.4. Path condition analysis

After XBFG paths and message sequences are compared, paths to be retested have been divided into two groups: (1) *old path* in $P[i+1]^{so}$ that does not need new test cases for regression testing; (2) *new path* in $P[i+1]^{sn}$ that needs new test cases for regression testing. In order to make full use of test cases from the baseline version and avoid redundant test case generation, we adopt *the principle of predicate logic* and compare *path conditions* of two versions. If they can be proven to be identical, the test cases required for *new path* can be selected from those in the baseline version.

In CFG of traditional structural program, the predicate constraints of a path come from branch nodes. It is effective to analyze predicates of all branch nodes when we want to determine the predicate constraint expression of a path, A BPEL program is in fact also a structured program while a BPEL flow is more complex because some new mechanisms, such as control dependencies and dead path elimination are introduced[3]. The predicate constraints in BPEL come from not only branch nodes, but also merge nodes.

The *condition* field of a XBFG element records predicate constraint (also called *prc* in short). It is composed of two parts, i.e., *expressions* and *operands*. In BPEL, *expression* may be a variable or a function. Let *exp* denote the expression, *op* denote the operand and $op \in \{=, >, >=, <, \le, !=, !\}$. There are three kinds of predicate constraints in XML Schema:

[3] When a target activity is not performed due to the value of the ⟨joinCondition⟩ (implicit or explicit) being false, its outgoing links MUST be assigned a false status according to some rules and Link Semantics. This has the effect of propagating false link status transitively along entire paths formed by successive links until a join condition is reached that evaluates to true. This approach is called Dead-Path Elimination (DPE) (Alves et al., 2007).

Table 5
Subject program and statistics.

Version	Loc of BPEL	Activity	WSDL spec	Message
$v1.0$	274	14	2	8
$v1.1$	274	14	2	8
$v1.2$	274	14	2	8
$v1.3$	274	14	2	8
$v2.0$	490	24	5	13

1) *Boolean prc*. Its general format is $op\ exp_1$, where exp_1 is an expression, the value of which is a Boolean data, $op \in \{ ! \}$.
2) *Numeric prc*. Its general format is $exp_1\ op\ exp_2$, where exp_1 and exp_2 are expressions, the values of which are numeric data, $op \in \{=, >, >=, <, <=, !=\}$.
3) *String prc*. Its general format is $exp_1\ op\ exp_2$, where exp_1 and exp_2 are expressions, the values of which are string data, $op \in \{=, !=\}$.

According to the classification of XBFG elements, *EDN*, *EMN*, *MBN*, *MMN*, *CBN*, *CMN* and *CE* may have *condition* fields. But predicate constraints only exist in those *CE*s whose sources are branch nodes and in those whose targets are merge nodes. Therefore, predicates can be further divided into *branch predicate* and *merge predicate*.

Branch predicate is defined as a condition expression attached with XBFG branch nodes for determining which *CE*s are used in the next execution step. As we mentioned in Section 2, the branch predicate is fetched from `condition` sub-element nested in `if`, `while` and `repeatUntil`. The predicate can be *Boolean*, *Numeric* or *String* type.

Merge predicate is defined as a condition expression attached with XBFG merge nodes for determining which *CE*s are merged. BPEL designs `joinCondition` for synchronization of several activities by evaluating *link* status. A *link* generally has three kinds of statuses: *true*, *false* and *unset*. *Merge predicate* is of the type *Boolean*, and its expression is the same as the name of *link*, which could be regarded as a variable.

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

Suppose $prc[i]$ and $prc[i+1]$ are two predicate constraints for XBFG path $p[i]$ and $p[i+1]$, respectively, $prc[i] = prc[i+1]$ iff $prc[i].EP == prc[i+1].EP$ && $prc[i].PT == prc[i+1].PT$ && $prc[i].F == prc[i+1].F$. That is, two predicate constraints are identical only if their constraint expressions, predicate types and conjunction are all the same.

Path condition (also called *pac*) is a vector containing predicate constraints of the path. The way of identifying path condition is first to collect all the predicate constraints in the path before combining them together. As predicate constraints are bound with *CE*, path condition can be collected by traversing all *CE*s in the XBFG paths. Let $pac[i]$ denote the path condition of $p[i]$, $ce_k[i]$ denote any *CE* in $p[i]$, then

$$pac[i] = \cup_{k=1}^{n} \{ce_k[i].prc[i] | ce_k[i] \in p[i]\}$$

where n denotes the number of *CE*s in $p[i]$.

Two path conditions $pac[i]$ and $pac[i+1]$ of XBFG path $p[i]$ and $p[i+1]$ are identical if and only if for each $prc[i+1]$ in $pac[i+1]$ and $prc[i]$ in $pac[i]$, $prc[i] = prc[i+1]$. Algorithm 4 describes the comparison of path conditions. As the number of XBFG elements in XBFG path $p[i]$ is n, the time complexity of Algorithm 4 is $O(n^2)$ (Table 5).

Algorithm 4. Path condition comparison algorithm.

Input $p[i]$, $p[i+1]$: paths to be compared based on path condition
Output *result*: comparison result of $p[i]$ and $p[i+1]$
PathConditionComparison(p[i], p[i+1])
if $p[i].pac[i].size != p[i+1].pac[i+1].size$ **then**
  Return *false*
end if
for each $prc[i]$ in $p[i].pac[i]$ **then**
  $prc[i].isMatch = false$
end for
for each $prc[i+1]$ in $p[i+1].pac[i+1]$ **then**
  $prc[i+1].isMatch = false$
end for
for each $prc[i]$ in $p[i].pac[i]$ **do**
  for each $prc[i+1]$ in $p[i+1].pac[i+1]$ **do**
    if $prc[i].isMatch = false$ && $prc[i].EP == prc[i+1].EP$ && $prc[i].PT == prc[i+1].PT$ && $prc[i].F == prc[i+1].F$ **then**
      $(prc[i+1].isMatch = prc[i].isMatch) = true$
    end if
  end for
end for
for each $prc[i+1]$ in $p[i+1].pac[i+1]$ **do**
  if $prc[i+1].isMatch == false$ **then**
    $result = false$
  end if
$result = true$
Return *result*
end for

5.5. A simple case study

For LCS, we have obtained all XBFG paths and corresponding message sequences of $v1.0$ and $v1.1$ based on our approach of Sections 4.3 and 4.4. Let $P[1.0] = \{p_1[1.0], p_2[1.0]\}$ denotes the set of XBFG paths in $v1.0$, the details of $p_1[1.0]$ and $p_2[1.0]$ are shown in Table 6. Let $MS[1.0] = \{ms_1[1.0], ms_2[1.0]\}$ denote the set of XBFG message sequences in $v1.0$, where $ms_1[1.0]$ and $ms_2[1.0]$ are corresponding message sequences of $p_1[1.0]$ and $p_2[1.0]$, respectively. The details of $ms_1[1.0]$ and $ms_2[1.0]$ are shown in Table 7 where we can see that $ms_1[1.0] = ms_2[1.0]$. Similarly, let $PC[1.0] = \{pc_1[1.0], pc_2[1.0]\}$ denote the set of path conditions in $v1.0$, where $pc_1[1.0]$ and $pc_2[1.0]$ are corresponding path conditions of $p_1[1.0]$ and $p_2[1.0]$. Details of $pc_1[1.0]$ and $pc_2[1.0]$ are shown in Table 8. For $v1.1$ of LCS, we use the similar naming rules to label all XBFG paths, message sequences and path conditions. Details of $P[1.1] = \{p_1[1.1], p_2[1.1]\}$, $MS[1.1] = \{ms_1[1.1], ms_2[1.1]\}$ and $PC[1.1] = \{pc_1[1.1], pc_2[1.1]\}$ are also shown in Tables 6, 7 and 8, respectively.

Applying Algorithm 2 of *XBFG path comparison*, we have

$$p_1[1.1] \overset{c}{\neq} p_1[1.0]$$

while

$$p_2[1.1] \overset{c}{=} p_2[1.0]$$

This means that the XBFG path $p_1[1.1]$ should be put into the *old path* set $P[1.1]^{so}$ and the *modified path* set $P[1.1]^{s}$ simultaneously. The next step is to find out whether any path in $P[1.1]^{so}$ should be

Table 6
XBFG paths of all five versions of LCS.

Ref.	XBFG path
p_1[1.0]	5,6,1,28,29,7,30,8,9,2,10,31,11,32,12,33,21,34,13,14,3,16,35,15,36,22,37,17,18,4,20,38,19,39,40,25,41,23,42,26,45,27
p_2[1.0]	5,6,1,28,29,7,30,8,9,10,2,31,11,32,12,33,21,34,13,14,3,16,35,15,36,22,37,17,18,4,20,38,19,39,40,25,43,24,44,26,45,27
p_1[1.1]	5,6,1,28,29,7,30,8,9,2,10,31,11,32,12,33,21,34,13,14,3,16,35,15,36,22,37,17,18,4,20,38,19,39,40,25,**47**,**46**,**48**,26,45,27
p_2[1.1]	5,6,1,28,29,7,30,8,9,2,10,31,11,32,12,33,21,34,13,14,3,16,35,15,36,22,37,17,18,4,20,38,19,39,40,25,43,24,44,26,45,27
p_1[1.2]	5,6,1,28,29,7,30,8,**47**,**46**,**48**,31,11,32,12,33,21,34,13,14,3,16,35,15,36,22,37,17,18,4,20,38,19,39,40,25,41,23,42,26,45,27
p_2[1.2]	5,6,1,28,29,7,30,8,**47**,**46**,**48**,31,11,32,12,33,21,34,13,14,3,16,35,15,36,22,37,17,18,4,20,38,19,39,40,25,43,24,44,26,45,27
p_1[1.3]	5,6,1,28,29,7,30,8,9,2,10,31,11,32,12,33,21,34,13,14,3,16,35,15,36,22,37,17,18,4,20,38,19,39,40,25,41,23,42,26,45,27
p_2[1.3]	5,6,1,28,29,7,30,8,9,10,2,31,11,32,12,33,21,34,13,14,3,16,35,15,36,22,37,17,18,4,20,38,19,39,40,25,43,24,44,26,45,27
p_1[2.0]	5,6,1,28,**66**,**48**,**67**,**49**,**50**,**46**,**51**,**68**,**52**,**69**,7,30,8,9,2,10,31,11,32,12,33,21,34,13,14,3,16,35,15,36,22,37,17,18,4,20,38,19,39,40,25,41,23,42,26,**70**,**55**,**71**,**56**, **72**,**57**,**58**,**47**,**59**,**61**,**73**,**60**,**74**,**64**,**75**,**62**,**76**,**65**,**79**,27
p_2[2.0]	5,6,1,28,**66**,**48**,**67**,**49**,**50**,**46**,**51**,**68**,**52**,**69**,7,30,8,9,2,10,31,11,32,12,33,21,34,13,14,3,16,35,15,36,22,37,17,18,4,20,38,19,39,40,25,43,24,44,26,**70**,**55**,**71**,**56**, **72**,**57**,**58**,**47**,**59**,**61**,**73**,**60**,**74**,**64**,**75**,**62**,**76**,**65**,**79**,27
p_3[2.0]	5,6,1,28,**66**,**48**,**67**,**49**,**50**,**46**,**51**,**68**,**52**,**69**,7,30,8,9,2,10,31,11,32,12,33,21,34,13,14,3,16,35,15,36,22,37,17,18,4,20,38,19,39,40,25,41,23,42,26,**70**,**55**,**71**,**56**, **72**,**57**,**58**,**47**,**59**,**61**,**73**,**60**,**74**,**64**,**77**,**63**,**78**,**65**,**79**,27
p_4[2.0]	5,6,1,28,**66**,**48**,**67**,**49**,**50**,**46**,**51**,**68**,**52**,**69**,7,30,8,9,2,10,31,11,32,12,33,21,34,13,14,3,16,35,15,36,22,37,17,18,4,20,38,19,39,40,25,43,24,44,26,**70**,**55**,**71**,**56**, **72**,**57**,**58**,**47**,**59**,**61**,**73**,**60**,**74**,**64**,**77**,**63**,**78**,**65**,**79**,27

Table 7
Message sequences for corresponding XBFG paths of all five versions.

Ref.[a]	Message sequence
ms_1[1.0]	?(String,Boolean,Boolean,Double) → !String → ?Int → !(String,Boolean,Boolean,Double) → !(String,Boolean,Boolean, Double) → ?(String,String, String,Double,String,String,Int) → ?(String,String,String,Double,String,String,Int) → !(String, String,String,Double,String,String,Int)
ms_2[1.0]	
ms_1[1.1]	?(String,Boolean,Boolean,Double) → !String → ?Int → !(String,Boolean,Boolean,Double) → !(String,Boolean,Boolean, Double) → ?(String,String, String,Double,String,String,Int) → ?(String,String,String,Double,String,String,Int) → !(String, String,String,Double,String,String,Int)
ms_2[1.1]	
ms_1[1.2]	?(String,Boolean,Boolean,Double) → !String → ?Int → !(String,Boolean,Boolean,Double) → !(String,Boolean,Boolean, Double) → ?(String,String, String,Double,String,String,Int) → ?(String,String,String,Double,String,String,Int) → !(String, String,String,Double,String,String,Int)
ms_2[1.2]	
ms_1[1.3]	?(String,Boolean,Boolean,Double) → !String → ?Int → !(String,Boolean,Boolean,Double) → !(String,Boolean,Boolean,Double) → ?(String,String, String,Double,String,String,**Double**) → ?(String,String,String,Double,String,String,**Double**) → !(String,String,String,Double,String,String,**Double**)
ms_2[1.3]	
ms_1[2.0]	?(String,Boolean,Boolean,Double) → !String → **?String** → **!String** → ?Int → !(String,Boolean,Boolean,Double) → !(String,Boolean,Boolean,Double) → ?(String,String,String,Double,String,String,Int) → ?(String,String,String,Double,String,String,Int) → **!initiateTaskMessage**[b] → ?**initiate- TaskResponseMessage**[b] → ?**taskMessage**[b] → !(String,String,String,Double,String,String,Int)
ms_2[2.0]	
ms_3[2.0]	
ms_4[2.0]	

[a] The reference of message sequence ms_i[*version*] is attached with XBFG path p_i[*version*].
[b] The concrete types of marked message are ignored here for saving the space since the definition of these `complexType` is too complicated.

Table 8
Path conditions for corresponding XBFG paths of all five versions.

Ref.[a]	Path condition			
	Predicate constraint	Constraint expression	Predicate type	F
pc_1[1.0]	prc_1[1.0]	getVariableData('loanOffer1')>getVariableData('loanOffer2')	Numeric	AND
pc_2[1.0]	prc_2[1.0]	getVariableData('loanOffer1')≤getVariableData('loanOffer2')	Numeric	AND
pc_1[1.1]	prc_1[1.1]	getVariableData('loanOffer1')>getVariableData('loanOffer2')	Numeric	AND
pc_2[1.1]	prc_2[1.1]	getVariableData('loanOffer1')≤getVariableData('loanOffer2')	Numeric	AND
pc_1[1.2]	prc_1[1.2]	getVariableData('loanOffer1')>getVariableData('loanOffer2')	Numeric	AND
pc_2[1.2]	prc_2[1.2]	getVariableData('loanOffer1')≤getVariableData('loanOffer2')	Numeric	AND
pc_1[1.3]	prc_1[1.3]	getVariableData('loanOffer1')>getVariableData('loanOffer2')	Numeric	AND
pc_2[1.3]	prc_2[1.3]	getVariableData('loanOffer1')≤getVariableData('loanOffer2')	Numeric	AND
pc_1[2.0]	prc_{11}[2.0]	getVariableData('loanOffer1')>getVariableData('loanOffer2')	Numeric	AND
	prc_{12}[2.0]	getVariableData('LoanOfferReview_globalVariable')='COMPLETE'	String	AND
	prc_{13}[2.0]	getVariableData('LoanOfferReview_globalVariable')='ACKNOWLEDGE'	String	AND
pc_2[2.0]	prc_{21}[2.0]	getVariableData('loanOffer1')≤getVariableData('loanOffer2')	Numeric	AND
	prc_{22}[2.0]	getVariableData('LoanOfferReview_globalVariable')='COMPLETE'	String	AND
	prc_{23}[2.0]	getVariableData('LoanOfferReview_globalVariable')='ACKNOWLEDGE'	String	AND
pc_3[2.0]	prc_{31}[2.0]	getVariableData('loanOffer1')>getVariableData('loanOffer2')	Numeric	AND
	prc_{32}[2.0]	getVariableData('LoanOfferReview_globalVariable') ≠ 'COMPLETE'	String	OR
	prc_{33}[2.0]	getVariableData('LoanOfferReview_globalVariable') ≠ 'ACKNOWLEDGE'	String	OR
pc_4[2.0]	prc_{41}[2.0]	getVariableData('loanOffer1')≤getVariableData('loanOffer2')	Numeric	AND
	prc_{42}[2.0]	getVariableData('LoanOfferReview_globalVariable') ≠ 'COMPLETE'	String	OR
	prc_{43}[2.0]	getVariableData('LoanOfferReview_globalVariable') ≠ 'ACKNOWLEDGE'	String	OR

[a] The reference of path condition pc_i[*version*] is attached with XBFG path p_i[*version*].

shifted to the *new path* set $P[1.1]^{sn}$. From *message sequence comparison* based on Algorithm 3, we have

$$p_1[1.1]\overset{m}{=}p_1[1.0]$$

It means that $p_1[1.1]$ should be still remained in $P[1.1]^{so}$ and we do not need to generate new test cases.

After both *XBFG path comparison* and *message sequence comparison*, we find that only one XBFG path $p_1[1.1]$ is affected by the modification when service evolves from $v1.0$ to $v1.1$ and the other path $p_2[1.1]$ does not need to be rested. Since $p_1[1.1]$ belongs to the *old path*, it is unnecessary to generate new test cases to retest this path. So *path condition analysis* is ignored here since $P[1.1]^{sn}$ is empty. We can use the test cases that are attached with $p_1[1.0]$ to validate the correctness of the modified version $v1.1$.

6. Experimental evaluation

In this section, we conduct an experimental study to evaluate the effectiveness of proposed approach by showing that it has a high change coverage rate using selected test cases. We will discuss how to set evaluation criterion, how to collect data and how to evaluate change coverage. In addition, we also discuss the threat to validity in this section.

6.1. Experimental design

6.1.1. Subject programs, versions

Suppose that *loan composite service* has passed through a continuous evolution and four versions are generated, including modified version $v1.1$ mentioned in Section 2. Fig. 9(a)–(c) shows the specific modification in BPEL code for the other three versions, respectively. In version 1.2 of *LCS* ($v1.2$ for short), the service integrator uses another candidate service *CreditRatingService* with the same functionality to replace the corresponding one in $v1.0$. In version 1.3 of *LCS* ($v1.3$ for short), the `message` LoanServiceRequestMessage defined in *LoanService.wsdl* evolves by changing the content of user-defined `complexType` LoanApplicationType. In version 2.0 of *LCS* ($v2.0$ for short), more modifications have been made on $v1.0$, where two additional partner services have been imported, one is *customService*, which provides the function of SSN querying, and the other is *taskService*, which provides the function of manual checking for users. Therefore two partnerLinks are added into the process, and some new interfaces are imported in this version. We set $v1.0$ as the baseline version of $v1.1$, $v1.2$, $v1.3$ and $v2.0$.

Table 5 shows the descriptive statistics of the subject program of different versions. The scale information of BPEL specification, including LOC of BPEL document and the number of activities defined in the process, are shown in the second and third column, respectively. The number of related WSDL specifications used in the composite service is listed in the fourth column. The number of message types used in communication between process and partner services is listed in the rightmost column of the table. For every version of subject program, we used the test case selection approach presented in Section 4 to determine the XBFG path sets. We then perform regression testing with some test case coming from the baseline version and collect the test results.

6.1.2. Evaluation criterion

In this section, we discuss how to analyze the change coverage. Suppose the actual numbers of changes of BPEL process (in fact, it is the changes of BPEL activities), bindings and interfaces are denoted as $|\Delta[i]_{pc}|$, $|\Delta[i]_{bc}|$ and $|\Delta[i]_{ic}|$, respectively, and the number of each kind of change covered is denoted as num_{pc}, num_{bc} and num_{ic}, respectively, then the coverage rate of *process changes* is evaluated as follows:

$$\rho[i]_{pc} = \frac{num_{pc}}{\Delta[i]_{pc}} \times 100\,\%$$

The coverage rate of binding changes is evaluated as follows:

$$\rho[i]_{bc} = \frac{num_{bc}}{\Delta[i]_{bc}} \times 100\,\%$$

And the coverage rate of interface changes is evaluated as follows:

$$\rho[i]_{ic} = \frac{num_{ic}}{\Delta[i]_{ic}} \times 100\,\%$$

As Table 2 shows that *process change* and *binding change* occur in BPEL document only, and such changes behave as changes of *XBFG elements* in XBFG model, $\rho[i]_{pc}$ and $\rho[i]_{bc}$ can be represented as proportion between the covered changes and the actual changes in XBFG model. Let $|n_n|$, $|n_m|$ and $|n_d|$ denote the number of new XBFG elements, modified XBFG elements and deleted XBFG elements caused by service evolution, respectively. Let $|n'_d|$ denote the number of deleted elements that can be covered by the calculated *old path* set and *new path* set, then we define

$$\rho[i]_{pc} = \frac{num_{pc}}{\Delta[i]_{pc}} \times 100\,\% = \frac{|n_n| + |n_m| + |n'_d|}{|n_n| + |n_m| + |n_d|} \times 100\,\%$$

Similarly, let $|b_n|$, $|b_m|$ and $|b_d|$ denote the number of new bindings, modified bindings and deleted bindings, respectively, then we define

$$\rho[i]_{bc} = \frac{num_{bc}}{\Delta[i]_{bc}} \times 100\,\% = \frac{|b_n| + |b_m|}{|b_n| + |b_m| + |b_d|} \times 100\,\%$$

Although a WSDL document is composed of the definitions of `port`, `binding`, `portType`, `operation`, `message` and `type`, our approach covers only the `message` and `type` used by the composite service. Let $|P_d|$, $|B_d|$, $|PT_d|$, $|O_d|$, $|M_d|$ and $|T_d|$, respectively, denote the set of changed ports, bindings, portTypes, operations, messages and types that are have been defined in WSDL documents for both composite service and partner services, $|M_u|$ and $|V_u|$ denote the number of changed messages and variables that are covered, then we define

$$\begin{aligned}\rho[i]_{ic} &= \frac{num_{ic}}{\Delta[i]_{ic}} \times 100\,\% \\ &= \frac{|M_u| + |V_u|}{|P_d| + |B_d| + |PT_d| + |O_d| + |M_d| + |T_d|} \times 100\,\%\end{aligned}$$

From the three estimation equations above, it can be inferred that $\rho[i]_{pc}$ depends on the proportion between $|n'_d|$ and the actual changes of XBFG elements, $\rho[i]_{bc}$ depends on the proportion between $|b_d|$ and the actual changes of binding, and $\rho[i]_{ic}$ depends on the proportion between ($|M_u|+|V_u|$) and the number of actual changed elements in WSDL documents.

6.1.3. Prototype tool

We have developed a prototype tool, named RTGenius4BPEL (regression testing genius for BPEL-based service), for implementing the automatic regression testing of composite Web service. It has three main functions:

(1) **Test case selection**. Service integrators can determine which paths are *old paths* and select test cases from the baseline version.
(2) **New test cases generation**. Service integrators can determine which paths are *new paths* and generate new test cases

```
        ... ...
(1)     <partnerLink name="client".../>
(2)(46) <partnerLink name="creditRatingService".../>
(3)     <partnerLink name="UnitedLoanService".../>
        ... ...
          <sequence>
(7)           <assign name="copySSN">
              ...
                <copy>
                  <from> <literal><service-ref><EndpointReference>
                     <Address>http://www.creditRS/</Address>
                     <ServiceName>invoiceService</ServiceName>
                  </EndpointReference></service-ref></literal></from>
                  <to partnerLink="creditRatingService" />
                </copy>
(8)      ... ...
```

(a)Modified Part in ***LoanFlow.bpel*** for V1.2

```
... ...
<definitions name="LoanService" ...>
      ... ...
      <element name="loanApplication" type="s1:LoanApplicationType"/>
      <complexType name="LoanApplicationType">
        <sequence>
          ... ...
          <element name="carYear" type="string"/>
          <element name="creditRating" type="double"/>
        </sequence>
      </complexType>
... ...
</definitions>
```

(b)Modified Part in ***LoanService.wsdl*** for V1.3

```
        ... ...
(1)      <partnerLink name="client".../>
(46)     <partnerLink name="customerService".../>
(2,3,4)  ... ...
(47)     <partnerLink name="TaskService".../>
        ... ...
(5)      <receive name="receiveInput" partnerLink="client" portType="tns:LoanFlow"
              Operation="initiate" variable="input" createInstance="yes"/>
         <scope name="getCustomerSSN">
           <variables>...</variables>
           <sequence>
(48)         <assign name="assignRequest">...</assign>
(49)         <invoke name="getCustomerSSN" partnerLink="customerService" portType="tns:CustomerService"
                 operation="getCustomerSSN" inputVariable="getCustomerSSNRequest" outputVariable="getCustomerSSNResponse"/>
(52)         <assign name="assignResponse">...</assign>
           </sequence>
         </scope>
         <scope name="GetCreditRating" variableAccessSerializable="no"> ··· ··· </scope>
         <scope name="scope_collectOffers" variableAccessSerializable="no"> ··· ··· </scope>
         <scope name="LoanOfferReview"···>
           <variables>···</variables>
           <sequence>
(55)         <assign name="LoanOfferReview_AssignTaskAttributes">···</assign>
(56)         <assign name="LoanOfferReview_AssignSystemTaskAttributes">···</assign>
(57)         <invoke name="initiateTask" partnerLink="TaskService" portType="taskservice:TaskService"
                 operation="initiateTask" inputVariable="initiateTaskInput" outputVariable="initiateTaskResponseMessage">
             </invoke>
(60)         <receive name="receiveCompletedTask" partnerLink="TaskService" portType="taskservice:TaskServiceCallback"
                 Operation="onTaskCompleted" variable="LoanOfferReview_globalVariable" createInstance="no">
             </receive>
           </sequence>
         </scope>
(64)     <if name="taskSwitch">
           <condition>
             "bpws:getVariableData('LoanOfferReview_globalVariable', 'payload', '/task:task/task:systemAttributes/task:state') = 'COMPLETED' and
             "bpws:getVariableData('LoanOfferReview_globalVariable', 'payload', '/task:task/task:systemAttributes/task:outcome') = 'ACKNOWLEDGE'"
           </condition>
             ...
(62)       <sequence><assign>···</assign></sequence>
           <else>
             ...
(63)       <sequence><assign>···</assign></sequence>
           </else>
(65)     </if>
(27)     <invoke name="replyOutput" partnerLink="client" portType="tns:LoanFlowCallback"
             operation="onResult" inputVariable="selectedLoanOffer"/>
        ... ...
```

(c)Modified Part in ***LoanFlow.bpel*** for V2.0

Fig. 9. XBFGs of *LCS* for three modified versions.

automatically or semi-automatically, which is outside the scope of our study.

(3) **Change coverage analysis**: Service integrators can evaluate the coverage rate of all changes in different versions.

6.2. Data collection

Prior to test case selection for $v1.1$, $v1.2$, $v1.3$ and $v2.0$ of *LCS*, some preparation work should be finished, such as the construction of XBFG, XBGF path computation, message sequence calculation and path condition extraction. According to the transformation rules presented in Section 4.2, we can get XBFG models of all five versions, as shown in Fig. 10(a), (b), (c), (d) and (e), corresponding to $v1.0$, $v1.1$, $v1.2$, $v1.3$ and $v2.0$, respectively.

Let $P[v]$ denote the set of XBFG paths of version v and $P[v] = \{p_1[v], p_2[v], \ldots, p_k[v], \ldots\}$ $(k \geq 1)$, where $p_k[v]$ represents the kth XBFG path in $P[v]$. Let $MS[v]$ be the set of XBFG message sequences of version v and $MS[v] = \{ms_1[v], ms_2[v], \ldots, ms_k[v], \ldots\}$, where $ms_k[v]$ represents the message sequence corresponding to the XBFG path $p_k[v]$. Similarly, let $PC[v]$ be the set of XBFG Path conditions of version v and $PC[v] = \{pc_1[v], pc_2[v], \ldots, pc_k[v], \ldots\}$, where $pc_k[v]$ represents the path condition corresponding to the XBFG path $p_k[v]$.

The generated XBFG paths of all versions can be computed with Algorithm 1. Table 6 shows the construction of XBFG elements for each XBFG path where the field $ID.id$ is the representative notation of each element. In this table, the bold numbers indicate the modified part based on the baseline version by performing XBFG path comparison with Algorithm 2. Similarly, the XBFG message sequence for the corresponding XBFG path can be calculated according to the steps illustrated in Fig. 8. The generated message sequences are shown in Table 7 and the bold parts indicate the modified contents relative to the corresponding message sequence of the baseline version according to Algorithm 3. In addition, details of path condition of each XBFG path is provided in Table 8.

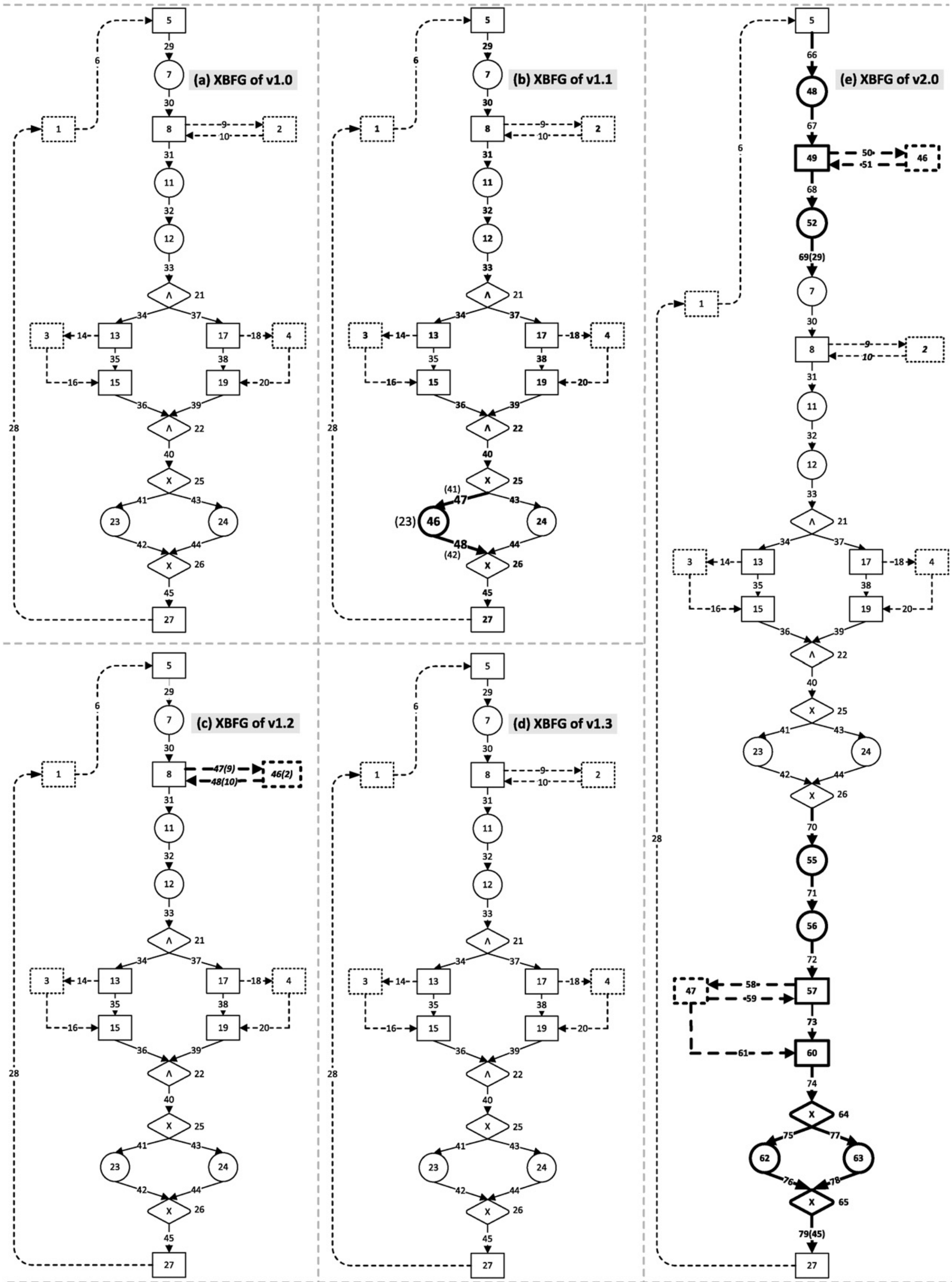


Fig. 10. The XBFGs of *LCS* for the initial version and four modified versions.

Table 9
Test case selection for four modified versions.

Ver.	Path	Old	New	Test selection from baseline	Total test	%
$v1.1$	$p_1[1.1]$	√		t_1, t_2, t_3	6	50
	$p_2[1.1]$					
$v1.2$	$p_1[1.2]$	√		t_1, t_2, t_3	6	100
	$p_2[1.2]$	√		t_4, t_5, t_6		
$v1.3$	$p_1[1.3]$		√	t_1, t_2, t_3	6	100
	$p_2[1.3]$		√	t_4, t_5, t_6		
$v2.0$	$p_1[2.0]$		√	t_1, t_2, t_3	6	100
	$p_2[2.0]$		√	t_4, t_5, t_6		
	$p_3[2.0]$		√	t_1, t_2, t_3		
	$p_4[2.0]$		√	t_4, t_5, t_6		

Table 10
Data statistics of all five versions.

Version	XBFG path	Changed path	XBFG element	Changed element	Message	Changed message	Variable	Changed variable	**Change type**
$v1.0$	2	-	45	-	8	-	4	-	-
$v1.1$	2	1	45	3	8	0	4	0	Process change
$v1.2$	2	2	45	3	8	0	4	0	Binding change
$v1.3$	2	2	45	3	8	3	4	1	Interface change (PS)
$v2.0$	4	4	75	32	13	5	7	5	Process change Binding change Interface change (PS)

A test suite for BEPL-based composite services can be generated automatically based on some decision coverage criterion against the XBFG path. Since it is not the emphasis of this article, we let the baseline test suite for $v1.0$ of *LCS* be $T[1.0] = \{t_1, t_2, t_3, t_4, t_5, t_6\}$, where the first three test cases (t_1, t_2, t_3) are bound to test XBFG path $p_1[1.0]$, while the last three (t_4, t_5 and t_6) are bound to test $p_2[1.0]$. With Tables 6, 7 and 8, we can perform test case selection and the result is shown in Table 9.

In Table 9, all generated XBFG paths for each version are listed in the second column. The third and fourth column represent whether the current XBFG path is an *old path* or a *new path*. Test cases selected from the baseline $v1.0$ are shown in the fifth column. The number of selected test cases and the usage statistics are shown in the last two columns.

In $v1.1$ of *LCS*, XBFG path $p_2[1.1]$ is not affected by the modification, which means that no test case is needed for $p_2[1.1]$. Only $p_1[1.1]$ is judged as an *old path* and those test cases mapped to test $p_1[1.0]$ can be selected for testing $p_1[1.1]$. In $v1.2$ of *LCS*, both paths $p_1[1.2]$ and $p_2[1.2]$ are considered to be *old paths* and test cases applied in $v1.0$ are all selected to guarantee the correctness of modified version $v1.2$. In $v1.3$ of *LCS*, though we have $p_1[1.3] = p_1[1.0]$ and $p_2[1.3] = p_2[1.0]$ after *XBFG path comparison*, for the corresponding message sequence in Table 7, we have $ms_1[1.3] \neq ms_1[1.0]$ and $ms_2[1.3] \neq ms_2[1.0]$, so both paths $p_1[1.3]$ and $p_2[1.3]$ are considered to be *new paths* on which new test cases must be generated to guarantee their correctness. On the other hand, from Table 8 we can see that $prc_1[1.3] = prc_1[1.0]$ and $prc_2[1.3] = prc_2[1.0]$, which means that the path conditions of both $p_1[1.3]$ and $p_2[1.3]$ equal to those of $p_1[1.0]$ and $p_2[1.0]$, respectively. So the test cases used to test $p_1[1.0]$ can also be applied in the regression testing in $v1.3$ although some new test cases are needed. In $v2.0$ of *LCS*, more modifications have been made to $v1.0$, including two additional partner services, which causes the number of XBFG paths to increase to 4. As all paths are classified as *new paths*, 6 test cases are selected as the final test suite for testing $v2.0$ according to the *path condition analysis*.

Table 11
Change coverage of four modified versions.

Version	$\rho[i]_{pc}$	$\rho[i]_{bc}$	$\rho[i]_{ic}$
$v1.1$	3/3 = 100%	–	–
$v1.2$	–	3/3 = 100%	–
$v1.3$	–	–	4/8 = 50%
$v2.0$	32/32 = 100%	2/2 = 100%	10/132 = 7.58%

6.3. Change coverage evaluation

In this section, we will present the coverage rate of four versions, and discuss the advantages and disadvantages of our approach based on the result.

Table 10 provides some statistics of XBFG model of each version and the comparison against the baseline version. The number of XBFG paths for each version is listed in the second column while the number of changed XBFG paths compared to the original version $v1.0$ is listed in the third column. In fourth and fifth column, the number of XBFG elements in the corresponding XBFG model and the number of the changed XBFG elements are provided, respectively. Similarly, the number of messages in XBFG model and the number of changed messages are shown in the sixth and seventh column, respectively. The number of variables used in the messages and the number of changed variables are shown in the eighth and ninth column, respectively. The last column presents the involved change types. For example, both $v1.0$ and $v1.1$ have 2 XBFG paths and $v1.1$ has one changed XBFG path over $v1.0$. Specifically, both $v1.0$ and v1.1 have 45 XBFG elements and $v1.1$ has 3 changed XBFG elements over $v1.0$ but both messages and variables in fact have no changes from $v1.0$ to v1.1. From the analysis of *XBFG path comparison* and *message sequence comparison*, we find that the change type from $v1.0$ to $v1.1$ is *process change*.

In Section 6.2, we have assumed that three test cases t_1, t_2 and t_3 are used to test XBFG path $p_1[1.0]$ while t_4, t_5 and t_6 are used to test $p_2[1.0]$. In $v1.1$, as $p_1[1.1]$ belongs to the *old path* under the *process change*, 3 test cases are used to test $p_1[1.1]$ and the number of generated test cases is 0. Since only one activity has been changed, the test case coverage is $\rho[1.1]_{pc} = 3/3 = 100\%$, which is also shown in the first row of Table 11.

In $v1.2$, as both $p_1[1.2]$ and $p_2[1.2]$ belong to the *old path* under the *binding change*, all 6 test cases can be used to test $P[1.2]$ and the number of generated test cases is also 0. Since only one binding has been changed, the test cases coverage is $\rho[1.2]_{bc} = 3/3 = 100\%$.

In $v1.3$, as both $p_1[1.3]$ and $p_2[1.3]$ belong to the *new path* under the *interface change* in partner service, new test cases must be generated. But from the *path condition analysis* above, we find that all 6 test cases can be used to test $P[1.3]$. In Table 10 we can see that one variable and three messages are changed during the evolution from $v1.0$ to $v1.3$, which causes change to definitions of one operation, one portType, one binding and one port. So the test cases coverage is $\rho[i]_{ic} = 4/8 = 50\%$.

During the evolution from $v1.0$ to v2.0, as all of the four XBFG paths in $v2.0$ 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 selected from $v1.0$ is 6. The coverage of test cases based on different change types are given as follows:

- *Process change*: 32 XBFG elements are added or modified and the experiment covers all 32 elements, so $\rho[2.0]_{pc} = 100\%$.
- *Binding change*: 2 bindings are added and the experiment covers both 2 bindings, so $\rho[2.0]_{bc} = 100\%$.
- *Interface change*: 132 XBFG elements are added or modified in *CustomerService.wsdl* and *TaskServiceWSIF.wsdl*. The experiment covers 10 of them (5 changed messages and 5 changed variables), so $\rho[2.0]_{ic} = 10/132^* 100\% = 7.58\%$. The 10 covered interface elements are all used by the composite service.

From the analysis result we can see that the interface change coverage is on the low side compared to the other two coverages. This may be due to the lack of full control over the interfaces of partner services from the perspective of *service integrator*. So when we perform regression testing for the *interface change*, only used messages and used variables can be covered while other definitions in interfaces, such as operations, portTypes, bindings and ports, are missed.

Considering the low coverage of interface change, we next consider the possibility of improving this coverage. Since not all the functionalities of the partner service are involved in the composite service, many interfaces of partner services actually are irrelevant to the tested composite service, which may directly reduce the change coverage of interfaces. We modify the interfaces of partner service in version 2.0 separately in three experiments. In experiment *Ep*1, we delete one new service and corresponding binding, one port type, one port and one operation in *CustomerService.wsdl*. The result is that 5 messages and 5 variables are covered, $\rho[2.0]_{ic} = 10/125^* 100\% = 8\%$; In experiment *Ep*2, we continue to delete 6 messages and their corresponding 6 variables of newly added service in *TaskServiceWSIF.wsdl*. The result is that 5 messages and 5 variables are covered, $\rho[2.0]_{ic} = 10/113^* 100\% = 8.85\%$; In experiment *Ep*3, we continue to delete 10 messages and their corresponding 10 variables of newly added service in *TaskServiceWSIF.wsdl*. The result is that 5 messages and 5 variables are still covered, $\rho[2.0]_{ic} = 10/93^* 100\% = 10.75\%$. Fig. 11 shows the trend curve of the changed interface coverage which increases steady with the reduction of irrelevant interfaces of partner services. It can be concluded that changed interface coverage is more precise if irrelevant interfaces can be eliminated.

In conclusion, our empirical study shows that our approach has more expressive capability than other approaches which only focus on process change. Furthermore, the selected test cases can cover most process changes and binding changes. The coverage of changed interface is not high since many unused functionalities of partner service are also included in the computation. If we can

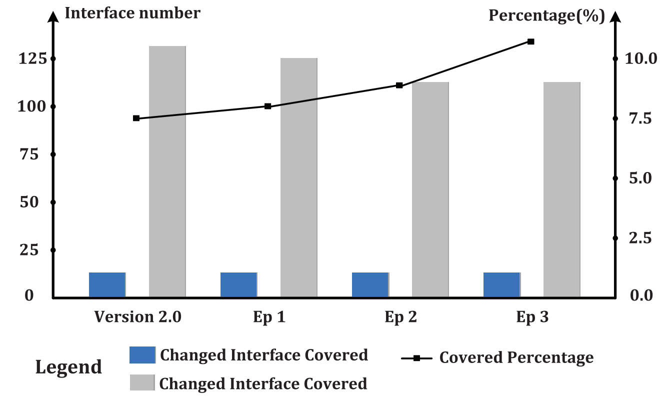


Fig. 11. Change coverage rate growth curve.

eliminate the useless interface, the interface change coverage can be increased.

6.4. Threats to validity

Threats to construct validity relates to the metrics used to evaluate the effectiveness of test case selection. In this experiment, we use the *change coverage* metric to evaluate the effectiveness of selected test cases by our approach. This is our newly defined metric since no such kind of existing metric is found in the related work of test case selection of web service. However, this metric may reduce the trustworthiness of our approach.

Threats to internal validity are the confounds that can affect the experimental results. When executing a test case on a service composition, the context of the entire composite service, especially the context of partner services, may change the execution path of each test case, which may also directly affect the result of change coverage evaluation. Therefore, before the execution of each test case, the prototype tool *RTGenius4BPEL* resets the context of all participating services to avoid impact caused by historical states.

Threats to external validity are concerned with whether the results are applicable to the general situation. First, only one subject service system has been selected. The particular feature of this system may affect our results. In addition, since it is really difficult to find publicly available benchmark programs with our required real-life modifications, the modified versions of $v1.1$, $v1.2$ and $v1.3$ are manually generated, which may also affect the generality of results. Second, the scale of selected subject system is not large enough to fully illustrate that our proposed approach is practical for testing large-scale BPEL-based systems. To reduce these two threats, we plan to collaborate with the industry to evaluate our proposed test case selection technique on existing large-scaled web services involving enough changes in real scenario under enterprise-level distributed computing environments.

7. Related work

Many researchers have studied the testing problem of Web services. In fact many methods and tools have been proposed to test basic service, composite service and even service-oriented application, such as unit testing (Lubke, 2007; Li et al., 2008a; Yan et al., 2006), model-based testing (Jose et al., 2006; Keum et al., 2006; Dong et al., 2006; Jeewani et al., 2006), regression testing (Liu et al., 2010; Penta et al., 2007), integration testing (Tarhini et al., 2006) and so on, which are mostly come from traditional software engineering. In the area of regression testing for Web services, many interesting methods or techniques have been proposed, as shown in Table 12.

Table 12
Comparison of related work on regression testing of composite service.

Reference	Perspective	Test object	WS technique	Test strategy	Approach	Change types included
Liu et al. (2010)	Service integrator	CS[a]	BPEL	White-Box	CFG	Process change
Tsai et al. (2009)	Service provider	S[b]	WSDL,OWL-S	Black-Box	CRM	–
Penta et al. (2007)	Service integrator	S	WSDL	Black-Box	Facet	Functional change Non-functional change
Tarhini et al. (2006)	Service integrator	CS	WSDL	White-Box	TPG,TLTS	Functional change Interface change
Ruth et al. (2007) and Ruth and Tu (2007)	Service integrator	S	WSDL	White-Box	Global CFG	Functional change
Mei et al. (2009)	Service integrator	CS	BPEL,WSDL	Black-Box	Tag	–
Chen et al. (2010)	Service integrator	CS	BPEL	Black-Box	BPFG	Process change
Our approach	Service integrator	CS	BPEL, WSDL	White-Box	XBFG	Process change Binding change Interface change

[a] CS is the abbreviation of composite service.
[b] S is the abbreviation of Service which is composed of basic service and composite service.

(1) *Among the many problems with regression testing of Web service, test object and the role of tester are both important, because they affect the test strategy and approach.*

Penta et al. discussed how to perform regression testing in detail in Penta et al. (2007), where they consider how the evolution of Web service were caused by *function change* and *non-function change* of complex service, and analyzed many testing methods and tools for all kinds of scenarios. But their test object of regression testing is mainly basic service because all their example scenarios involved the evolution caused by internal changes of a service itself.

Tarhini et al. (2006) showed how to obtain Web service as a two-level model, i.e., *interaction model* between basic services (or component), and *behavior model* of a basic service, used an input-complete TLTS (timed labeled transition system) to represent the two-level model, and further discussed all kinds of possible modifications of service system. The technique started from TLTS and needed to analyze the internal flow information of a basic service. However, this information is sometimes very difficult to obtain and even likely unavailable to typical service integrators and service users.

(2) *The second key problem is the selection of the "right" models to describe the complex BPEL process and interaction between BPEL process and partner services.*

For instance, CFG model is used for change impact analysis and regression testing path selection of BPEL process in Liu et al. (2010), where an impact analysis rule is proposed to identify the test paths affected by the change of BPEL concurrent control structures. Ginige et al. expressed BPEL control flow as algebraic expression using Kleen Algebra, then identified the changes of process by comparing algebraic expressions (Jeewani et al., 2006). Compared with CFG model, the algebraic expression model may encounter difficulties when it is used for expressing complex structures. Our proposed XBFG model is based on CFG, and can describe not only the behavior of BPEL process but also the interaction between process and partner services.

Khan et al. proposed a model-based approach for regression testing of Web service, where service interfaces are described by visual contracts, i.e., *pre-* and *post*-conditions expressed as graph transformation rules. The analysis of conflicts and dependencies between these rules allows them to assess the impact of a change of the signature, contract, or implementation of an operation on other operations, and thus to decide which of the test cases is required for re-execution. Apart from giving the conceptual foundations and justifications of the approach, they also evaluated it with a case study of a bug tracking service in several versions (Khan and Heckel, 2009, 2011).

(3) *The third important problem with regression testing of Web service is how to select and generate test cases for testing those changed services.*

Wang et al. (2008) and Li et al. (2010) proposed a XBFG-based regression testing framework of composite service, which is the early work of this article, where a prototype of XBFG model was introduced and only a high-level framework was discussed. In this article, we not only provide details on how to define and construct XBFG, but also introduce the concept of XBFG path based on two sub-types *in-process path* and *out-process path*, and further determine how to select test case based on the comparisons of paths and conditions. More experiments are also conducted to provide a stronger support of our approach.

Lallali et al. (2008) proposed a method to test composite Web service described in BPEL. As a first step, the BPEL specification is transformed into an intermediate format (IF) model that is based on timed automata, which enables modeling of timing constraints. They defined a conformance relation between two timed automata (of implementation and specification) and then proposed an algorithm to generate test cases. Test case generation is based on simulation where the exploration is guided by test purposes. The proposed method was implemented in a set of tools which were applied to a common Web service as a case study.

Based on the safe and efficient regression test selection technique proposed by Rothermel et al. (1997), Ruth et al. designed a regression testing selection algorithm using a global CFG which is integrated from many CFGs of partner services, and discussed mainly how *concurrent change* affects regression testing (Ruth et al., 2007; Ruth and Tu, 2007, 2007; Lin et al., 2006). Even though CFG analysis is a normal technique to select test case for regression testing, it is a bit difficult for representing the structure with data flow information and ineffective for generating new test case because CFG has no pre-condition constraint.

Li et al. (2008a,b) proposed a test-selection minimization algorithm based on Liu et al. (2010). Chen et al. (2010) proposed a dependence analysis based test case prioritization technique for Web Service regression testing. Tsai et al. (2009) presented a model-based adaptive testing (MAT) for multi-versioned software based the coverage relationship model which can be used to select and rank test cases. But their algorithm only considers *BPEL process*, which is just one part of composite service as we discussed in Section 1. *Partner service* and *interface* are ignored in Li et al. (2008b) and Liu et al. (2010), but they are included in our approach.

(4) *Change coverage analysis is also an important problem with regression testing of Web service, because it is desirable to cover as many changed services or paths as possible.*

Change impact analysis is another problem in the evolution of composite service. Xiao et al. proposed a method for supporting change impact analysis at the business process level and code level, where an IPG (impact propagation graph) has been constructed on the basis of analyzing all call graphs(Xiao et al., 2007).

(5) *The high cost of regression testing of Web service should be a major concern of testers.*

Canfora and Penta (2006) discussed how the cost and restrictions change when different shareholders, including service developer, service provider, service integrator, third-party organization and user performs Web service regression testing independently. But they did not provide a practical approach for regression testing of Web service.

The cost can be reduced by building service stubs to simulate behaviors of message exchanges between services against data collected by monitoring (Canfora and Penta, 2006).

8. Conclusion and future work

The new characteristics of Web service bring a great challenge to testing and maintaining service-centric software system. In this article, we proposed an XBFG-based regression testing approach to capture the influence caused by *process change*, *binding change* and *interface change*. The generated XBFG paths from BPEL process and partner services are divided into two parts, where the first part can be re-tested by selecting test cases used in the baseline version and the second part can be tested by generating new test cases after performing *XBFG path comparison*, *message sequence comparison*, and *path condition analysis*, which cover the main aspects of functional regression testing of service composition. Our approach has extended the study on regression testing to testing composite service, process and interaction between them.

Our research represents an initial work on regression testing service-centric software system, because we mainly concentrate on the composite service in an orchestration way and only consider how to regression testing composite service from the view of service integrators. There are a lot of interesting problems to be studied in our future work. It can be concluded as follows:

- Change of partner service includes change of its interface and implementation. Both types of change are uncontrollable for *service integrator*. In this article, only the former type is considered. Possible solutions are proposed in Penta et al. (2007) using the predetermined black-box strategy to perform the regression testing periodically to actively check whether the implementation has been modified.
- Partner services are generally coming from different service providers, and most of them will charge service users even when they just call services for testing. A large amount of service calls will increase the cost rapidly; additional, the possibility of being attacked increases when the messages exchange occurs frequently. One possible solution to these problems is to construct stub modules to simulate partner services and use monitors to collect messages for stub modules, to reduce the call times of services.
- In this article, we discuss how to retest composite service produced based on BPEL, which is just one of many service composition languages. If a composite service is composed based on WS-CDL (Kavantzas et al., 2012) or OWL-S (Martin et al., 2012), how to deal with the evolution and maintenance, and further how to perform regression testing are all in our future works.

Acknowledgements

This work is supported partially by National Natural Science Foundation of China under Grant No. 60973149, partially by the Open Funds of State Key Laboratory of Computer Science of Chinese Academy of Sciences under Grant No. SYSKF1110, partially by Doctoral Fund of Ministry of Education of China under Grant No. 20100092110022, partially by Department of Jiangsu Education, under Grant No. JHB2011-3, Shenzhen-Hong Kong Innovation Circle Sponsorship Scheme under grant No. ZYB200907060012A.

References

Alves, A., Arkin, A., Askary, S., et al., 2007. Web services business process execution language version 2.0. OASIS Standard, 11.

Benedusi, P., Cmitile, A., De Carlini, U., 2002. Post-maintenance testing based on path change analysis. In: Proceedings of the Conference on Software Maintenance, pp. 352–361.

Canfora, G., Penta, M.D., 2006. SOA testing and self-checking. In: Proceedings of International Workshop on Web Services-Modeling and Testing-WS-MaTE, pp. 3–12.

Canfora, G., Penta, M.D., 2006. Testing services and services-centric systems: challenges and opportunities. IT Professional 8, 10–17.

Canfora, G., Penta, M.D., 2009. Service-oriented architectures testing: a survey, software engineering. Lecture Notes in Computer Science 5413, 78–105.

Chen, L., Wang, Z., Xu, L., Lu, H., Xu, B., 2010. Test case prioritization for web service regression testing. In: Proceedings of the fifth IEEE International Symposium on Service Oriented System Engineering (SOSE), pp. 173–178.

Christensen, E., Curbera, F., Meredith, G., et al., 2001. Web services description language (WSDL) 1.1, W3C note, vol. 15.

Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N., Weerawarana, S., 2002. Unraveling the Web services web: an introduction to SOAP. WSDL, and UDDI, Internet Computing, IEEE 6 (2), 86–93.

Dong, W.L., Yu, H., Zhang, Y.B., 2006. Testing BPEL-based web service composition using high-level Petri nets. In: Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference, EDOC'06, pp. 441–444.

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

Gudgin, M., Hadley, M., Rogers, T., et al., 2006. Web Services Addressing 1. 0—WSDL Binding, http://www.w3.org/TR/ws-addr-wsdl.

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

Jeewani, A. Ginige, Uma Sirinivasan, Athula Ginige, 2006. 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, pp. 171–178.

Jose, G.F., Javier, T., Claudio, D.L.R., 2006. Generating test cases specifications for BPEL compositions of web services using SPIN. In: Proceedings of the International Workshop on Web Services-Modeling and Testing, pp. 83–94.

Kavantzas, N., Burdett, D., Ritzinger, G., et al. Web services choreography description language version 1.0. http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/.

Keum, C.S., Kang, S., Ko, I.-Y., et al., 2006. Generating test cases for web services using extended finite state machine. Testing of Communicating Systems, 103–117.

Khan, T.A., Heckel, R., 2009. A methodology for model-based regression testing of web services, taic-part. Proceedings of the Testing: Academic and Industrial Conference—Practice and Research Techniques, pp. 123–124.

Khan, T.A., Heckel, R., 2011. On model-based regression testing of web-services using dependency analysis of visual contracts, taic-part. In: Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering (FASE'11/ETAPS'11).

Lallali, M., Zaidi, F., Cavalli, A., Hwang, I., 2008. Automatic timed test case generation for web services composition. In: Proceedings of the IEEE Sixth European Conference on Web Services (ECOWS'08), November 12–14, 2008, pp. 53–62.

Lei, Y., Carver, R.H., 2006. Reachability testing of concurrent programs. IEEE Transactions on Software Engineering 32 (6), 382–403.

Li, Z.J., Sun, W., Du, B., 2008a. BPEL4WS unit testing: framework and implementation. International Journal of Business Process Integration and Management 3, 131–143.

Li, Z., Tan, H., Liu, H., Zhu, J., Mitsumori, N.M., 2008b. Business-process-driven gray-box SOA testing. IBM System Journal 47 (3), 457–472.

Li, B., Qiu, D., Ji, S., 2010. Automatic test case selection and generation for regression testing of composite service based on extensible BPEL flow graph. In: Proceedings of the IEEE International Conference on Software Maintenance (ICSM), pp. 1–10.

Lin, F., Ruth, M., Tu, S., 2006. Applying safe regression test selection techniques to Java web services. In: Proceedings of the International Conference on Next Generation Web Services Practices, NWeSP 2006, pp. 133–142.

Liu, H., Li, Z., Zhu, J., Tan, H., 2010. Business process regression testing. Service-Oriented Computing-ICSOC 2007, 157–168.

Lubke, D., 2007. Unit testing BPEL compositions. Test and Analysis of Web Services, 149–171.

Martin, D., Burstein, M., Hobbs, J., et al. OWL-S: Semantic Markup for Web Services. http://www.w3.org/Submission/OWL-S/.

Maruyama, H., Tamura, K., Uramoto, R. Digest values for DOM (DOMHASH), Network Working Group. http://www.ietf.org/rfc/rfc2803.

Mei, L., Chan, W.K., Tse, T.H., Merkel, R.G., 2009. Tag-based techniques for black-box test case prioritization for service testing. In: Ninth International Conference on Quality Software, pp. 21–30.

Mei, L., Chan, W.K., Tse, T.H., Merkel, R.G., 2011. XML-manipulating test case prioritization for XML-manipulating services. Journal of Systems and Software 84, 603–619.

Penta, M.D., Bruno, M., Esposito, G., Mazza, V., Canfora, G., 2007. Web services regression testing. Test and Analysis of Web Services, 205–234.

Rothermel, G., Harrold, M.J., Safe, A., 1997. Efficient regression test selection technique. ACM Transactions on Software Engineering and Methodology (TOSEM) 6, 173–210.

Ruth, M., Tu, S., 2007. 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, p. 47.

Ruth, M., Tu, S., 2007. Concurrency issues in automating RTS for web services. In: IEEE International Conference on Web Services, ICWS 2007, pp. 1142–1143.

Ruth, M., Oh, S., Loup, A., 2007. Towards automatic regression test selection for web services. In: Computer Software and Applications Conference, COMPSAC 2007, 31st Annual International, pp. 729–736.

Tarhini, A., Fouchal, H., Mansour, N., 2006. Regression testing web services-based applications. In: Proceedings of the IEEE International Conference on Computer Systems and Applications, pp. 163–170.

Tarhini, A., Fouchal, H., Mansour, N., 2006. A simple approach for testing Web service based applications. Innovative Internet Community Systems, 134–146.

Tsai, W.T., Xinyu Zhou, Raymond, A. Paul, Yinong Chen, Xiaoying Bai, 2009. A coverage relationship model for test case selection and ranking for multi-version software. High Assurance Services Computing, 285–311.

Wang, D., Li, B., Cai, J., 2008. Regression testing of composite service: an XBFG-based approach. In: Congress on Services Part II, SERVICES-2. IEEE, pp. 112–119.

Xiao, H., Guo, J., Zou, Y., 2007. Supporting change impact analysis for service oriented business applications. In: Proceedings of the International Workshop on Systems Development in SOA Environments, Minneapolis, pp. 116–121.

Yan, J., Li, Z.J., Yuan, Y., et al., 2006. BPEL4WS unit testing: test case generation using a concurrent path analysis approach. In: Proceedings of the 17th International Symposium on Software Reliability Engineering, ISSRE'06, pp. 75–84.

Yoo, S., Harman, M., 2010. Regression testing minimization, selection and prioritization: a survey. Software Testing, Verification and Reliability, doi:10.1002/stvr.430.

Yuan, Y., Li, Z., Sun, W., 2006. A graph-search based approach to BPEL4WS test generation. In: Proceedings of the International Conference on Software Engineering Advances, p. 14.

Bixin Li received his PhD degree in computer science from Nanjing University, in 2001, and now he is a professor of School of Computer Science and Engineering at the Southeast University, Nanjing, China. His research interests include: program slicing and its application; software evolution and maintenance; software modeling, analysis, testing and verification. He has published over 90 articles in refereed conferences and journals. He is also the director of Institute of Software Engineering at Southeast University (ISEU). He is awarded by "QinLan Porgram" of Jiangsu Province, the Program for New Century Excellent Talents in University of China, and CVIC SE talents award of 2011.

Dong Qiu is a second year PhD student in the ISEU at Southeast University, under the supervision of Prof. Bixin Li. He received the BSc from Southeast University, China. His PhD work mainly concerns on the regression testing and verification of web services.

Hareton Leung joined Hong Kong Polytechnic University in 1994 and is now director of the Lab for Software Development and Management. He serves on the Editorial Board of Software Quality Journal. He is a fellow of Hong Kong Computer Society, chairperson of its Quality Management Division (QMSID) and chairperson of HKSPIN. He previously held team leader positions at BNR, Nortel, and GeneralSoft Ltd. He is also an accomplished industry consultant, giving advice on software testing, quality assurance, process and quality improvement, system development, and providing expert witness and litigation support. His clients include large and medium-sized organizations and government departments throughout Hong Kong and China, such as Housing Authority, Social Welfare Dept, MPFA, OGCIO, MTR, HIT, Intellectual Property Department, VTech, Hong Kong Productivity Council, AIA Shanghai, and Chinese Academy of Science.

Di Wang received his master degree in computer science from southeast university, in 2009, and now he works in People's Procuratorate of Jiangsu Province. His work mainly concerns on the regression testing of web services.