IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received September 19, 2019, accepted November 25, 2019, date of publication December 3, 2019,
date of current version December 16, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2957220

Test Case Selection for All-Uses Criterion-Based Regression Testing of Composite Service

SHUNHUI JI[1], (Member, IEEE), BIXIN LI[2], AND PENGCHENG ZHANG[1], (Member, IEEE)
[1]College of Computer and Information, Hohai University, Nanjing 211100, China
[2]School of Computer Science and Engineering, Southeast University, Nanjing 211189, China

Corresponding author: Shunhui Ji (shunhuiji@hhu.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61702159, and in part by the Natural Science Foundation of Jiangsu Province under Grant BK20170893.

ABSTRACT Composite services evolve for various reasons. Test case selection in the regression testing is an effective technique to ensure the correctness of modified versions meanwhile to reduce the cost of testing. However, few work has studied the test case selection problem based on the data flow testing criteria. In addition, there are three observable kinds of changes during the evolution, including *Process change*, *Binding change* and *Interface change*, which all bring impact to the data flow. To address these issues, a test case selection approach is proposed for regression testing of BPEL (Business Process Execution Language) composite service where all-uses criterion is satisfied and all the three change types are involved. BPEL composite service is modeled with a two-level model in which XCFG (eXtended Control Flow Graph) describes the behavior of BPEL process in the first level and WSDM (Web Service Description Model) depicts the interface information of composite service and partner services in the second level. Change impact analysis is performed to identify the affected definition-use pairs by comparing and analyzing two-level models of the baseline and evolved versions. And testing paths are generated to cover the affected definition-use pairs and select test cases based on the path condition analysis. Empirical result shows that the proposed approach is effective.

INDEX TERMS Regression testing, data flow testing, composite service, test case selection.

I. INTRODUCTION

With the development of cloud computing, service-oriented workflow has become a mainstream application to offer more complicated functions [1]. Service composition technology integrates existing services according to the composition mechanism to construct new services, which achieves services reusability and value-adding. In practice, composite services evolve a lot for reasons such as environment changing, bugs fixing and requirements enhancing [2]. In the whole lifetime, each version of composite service must be tested to ensure the correctness. Regression testing plays a very important role to check whether the modifications have brought any faults to the evolved version.

Many works have applied test case selection techniques to reuse test cases from existing test suite to test the modified part of evolved composite service [1], [3], [4]. However, less attention was paid on the data flow testing criteria oriented regression testing. For the correctness of composite service, the intuitive performance is that expected output can be acquired with any given input, which is realized by a series of definitions and uses of variables accompanied with control flow. The data flow correctness is an important and essential requirement. The all-uses criterion [5], which requires the test data to exercise at least one path going from each definition to each use reached by that definition, has been demonstrated to be practical and effective for the testing of composite services [6].

Composite service is a combination of composition process and partner services. We synthesized all possible changes into four types [7]: *Implementation Change*, *Process Change*, *Binding Change*, and *Interface Change*. The test case selection of composite service requires to identify as many modifications as possible in the evolved version. So one challenge is that not only the changes of composition behavior but also the observable changes of partner services need to be detected. Moreover, the various change types bring various impacts to the data flow. The affected definition-use (def-use)

The associate editor coordinating the review of this manuscript and approving it for publication was Zhangbing Zhou.

pairs need to be tested in the all-uses based regression testing of composite service. So another challenge is that, besides identifying the various changes, the various impact they bring to the def-use pairs also needs to be analyzed. We will discuss how to conquer the above challenges in this paper.

The main contribution of this paper with its preliminary version [8] is fourfold:

- We propose a two-level model to fully describe the BPEL composite service. EXtended Control Flow Graph XCFG) is constructed to model the composition process in the first level. Web Service Description Model (WSDM) is newly proposed to model the interface information of composite service and partner services in the second level. Besides, the model construction is provided as well.
- We update the classification of the types of affected def-use pairs by adding the ''Type Pairs'', and provide the graphical illustration of different types. The updated classification covers all the possible impact that various changes bring to the def-use pairs.
- We provide the change impact analysis algorithms to identify affected def-use pairs of different types caused by *process change*, *binding change*, and *interface change*. And the testing path generation for def-use pairs covering and test case selection are illustrated in detail.
- We explore five versions of carefully designed composite service to show how to select test cases for the evolved version. Our empirical result indicates that our approach is effective in selecting test cases for all-uses criterion based regression testing of BPEL composite services and can detect three kinds of change types, including *process Change*, *binding Change*, and *interface Change*.

The rest of the paper is organized as follows: Section 2 introduces WSDL (Web Service Description Language) and BPEL and gives a motivating example used to illustrate our idea; Section 3 introduces the classifications of composite service evolution and the impact types they bring to the data flow, and gives an overview of our approach; Section 4 illustrates the definition and construction of the two-level model for describing BPEL composite service; Section 5 discusses how to identify the affected def-use pairs by change impact analysis; Section 6 discusses how to performs test case selection for the affected def-use pairs; Section 7 performs some experiment and evaluation of our approach using the motivating example and its four modified versions; Section 8 compares the related works; Section 9 concludes the paper.

II. BACKGROUND

In this section, we present the prerequisite knowledge of WSDL and BPEL and provide a motivating example for convenient illustration in later sections.

A. WSDL

WSDL is an XML-based specification for describing Web services [9], including the operations the service provide, the interactive rules to use the service and the location to access the service.

A standard WSDL document is comprised of abstract part and concrete part. The abstract part is specified with `types`, `message`, `operation`, and `portType` to describe the functional interface of the service.

- `types`: it is a container for data type definitions. The data type can be an XML schema built-in type or a self-defined complex type.
- `message`: it defines the constitution of the message. A `message` is comprised of parts and each `part` is specified with `name` and `type`.
- `operation`: it provides an abstract description of a function supported by the service. The `operation` stipulates the message exchange pattern with `input` and `output` elements.
- `portType`: it is a set of `operations`.

The concrete part is specified with `binding`, `port`, and `service` to describe how and where to access the service.

- `binding`: it defines the rules for users interacting with the service, including concrete message format and transmission protocol for `portType`.
- `port`: it defines the endpoint with a network address corresponding to a `binding` so that users can get the location of the interface.
- `service`: it is a set of `ports`.

B. BPEL

BPEL is one of the standard service composition language [10]. It is an XML-based executable language for drawing up the workflow of composition. It specifies the interactions between composite service and partner services and orchestrates the interactions in some logical order so that the required business process can be constructed.

In BPEL composite service, `partnerLink` prescribes the interaction relationship between BPEL process and partner service, and `variable` provides the means for holding messages and data. In addition, `Process` written in BPEL is composed of many activities. The activities are classified into basic activity and structural activity. Basic activity, which can specify the interaction between services, exists independently or in a structural activity. The main basic activities defined in BPEL 2.0 specification are as follows:

- `invoke`: it is used to call partner service offered by service providers through the external interface exposed to users, with the variables referenced by `inputVariable` and `outputVariable` recording the request and response.
- `receive`: it is used to wait for a matching message to arrive the process.
- `reply`: it is used to send a response in reply to a request previously received.

- `assign`: it is used to update the values of variables. It can also be used to copy the content of `EndpointReference` to `partnerLink` so as to decide the service endpoint the process will bind.

Structural activity, which prescribes the execution order of activities with control flow logic, usually contains multiple activities. The main structural activities defined in BPEL 2.0 specification are as follows:

- `sequence`: it is used to define a set of activities to be performed sequentially according to the order they appear within `sequence` activity.
- `if/pick`: they are used to define the conditional behavior in which exactly one activity from a set of choices is selected to execute. The condition defined in `if` activity is a boolean expression, while it is an occurrence of one event in *pick* activity.
- `while/repeatUntil`: they are used to define the repetitive behavior in which the contained activity is executed repeatedly. The contained activity is executed as long as the specified `condition` being true in `while`, while it is executed until the specified `condition` being true in `repeatUntil`.
- `flow`: it is used to specify one or more activities to be performed concurrently, in which synchronization dependency between activities can be enabled with `link`.
- `forEach`: it is used to iterate the child scope activity $N + 1$ times where $N =< finalCounterValue > - < startCounterValue >$.
- `scope`: it is used to define a nested activity with its own context.

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

C. A MOTIVATING EXAMPLE

We choose the Loan Composite Service (*LCS*) [11] as a sample case. It is composed of process *LoanFlow* and three partner services including *CreditRatingService*, *UnitedLoanService*, and *StarLoanService*. *CreditRatingService* provides the synchronous function of computing loan grade for users. *UnitedLoanService* and *StarLoanService*, which share the same WSDL file, provide the asynchronous function of offering loan. *LoanFlow* starts with receiving the loan request from a client and calls *CreditRatingService* for confirming the client's loan grade. Then two concurrent tasks are activated: the process respectively calls *UnitedLoanService* and *StarLoanService* to get the loan result. The two results are compared and the one who has a smaller APR (Annual Percentage Rate) value is selected as the loan application goal. Finally, the chosen result is replied to the client. This version of *LCS*, which is denoted as *v1.0*, is set as the baseline version. The BPEL specifications of both *LCS v1.0* and its evolved version *v1.1* in which the content of *assign* in line 32 is modified are shown in Fig.1.

III. COMPOSITE SERVICE EVOLUTION

In this section, we introduce the evolution types of composite services that are taken into consideration in this paper. Then we propose the updated classification of the affected def-use pairs types. Finally, an overview of our approach for regression testing is provided.

A. CHANGE TYPES OF COMPOSITE SERVICE

A synthesized classification of all possible change types in composite services has been proposed in our earlier work [7], in which there are four change types including *Implementation Change*, *Process Change*, *Binding Change*, and *Interface Change*.

In this article, we will perform regression testing to BPEL composite service from the perspective of service integrator. Since the source code of parter services is non-observable for service integrator, *Implementation Change* is not considered. We will give a brief introduction to the other three change types with the help of the example shown in Fig.2, in which $S1$,$S2$, ... $S5$ denote composite services, $A1$, $A2$, ... $A7$ denote activities, $P1$ and $P2$ denote partner services, and $W1$ and $W2$ respectively denote WSDL specifications of composite service and partner service.

- *Process Change* includes change of activities, change of execution order, addition or deletion of partner services and change of variables. It usually happens when functional requirements change. In Fig.2, composite service $S1$ evolves to $S2$ by changing the activity $A4$ to $A7$.
- *Binding Change* denotes the change of endpoint address of partner service by replacing the partner service with another candidate service with the same functionality. It usually happens when the original service is unavailable or certain non-functional properties are violated. In Fig.2, $S1$ evolves to $S3$ since the partner service interacting with $A2$ has changed from $P1$ to $P2$.
- *Interface Change* denotes the change of interface defined in WSDL specification, including the change of `type`, `messages`, `operation`, `portType`, `binding`, `port`, and `service`. It involves the interface change of both composite service and partner service. In Fig.2, $S1$ evolves to $S4$ since the interface of composite service has changed from $W1$ to $W1'$; $S1$ evolves to $S5$ since the interface of partner service $P1$ has changed from $W2$ to $W2'$.

B. TYPES OF AFFECTED DEF-USE PAIRS

In the regression testing of evolved BPEL composite service, def-use pairs affected by *process Change*, *binding Change* and *interface Change* compared with the baseline version must be tested to satisfy the all-uses data flow testing criterion. Based on the classification of affected def-use pairs for the traditional program [12], [13], we categorize the affected def-use pairs of BPEL composite service into four types, i.e., *New Pairs*, *Value Pairs*, *Condition Pairs*, and *Type pairs*.

```
(1)  <process name="LoanFlow"...>
       <partnerLink>⋯</partnerLink>
       <variables>⋯</variables>
(3)    <sequence>
(5)      <receive name="receiveInput"  variable="input".../>
(6)      <scope name="GetCreditRating" ...>
(8)        <sequence>
(10)         <assign name="copySSN" />
               <copy><from variable="input" part="payload"⋯/>
                 <to variable="crInput" part="payload"⋯/> </copy>
               <copy><from>⋯
                 <Address>http://www.ratingService.com/⋯</from>
                 <to partnerLink="creditRatingService"/></copy></assign>
(11)         <invoke name="invokeCR"  inputVariable="crInput"
               outputVariable="crOutput"⋯/>
(12)         <assign name="copyRating">
               <copy><from variable="crOutput" part="payload"⋯/>
                 <to variable="input" part="payload"⋯/> </copy> </assign>
(9)        </sequence>
(7)      </scope>
(13)     <scope name="GetLoanOffer"...>
(15)       <sequence>
(17)         <assign name="prepareApplication">
               <copy><from variable="input" ⋯/>
                 <to variable="loanApplication" ⋯/> </copy></assign>
(18)         <flow name="collectOffers">
(20)           <sequence>
(22)             <invoke name="invokeUnitedLoan"
                   inputVariable="loanApplication"⋯/>
(23)             <receive name="receive_invokeUnitedLoan"
                   variable="loanOffer1"⋯/>
(21)           </sequence>
(24)           <sequence>
(26)             <invoke name="invokeStarLoan"
                   inputVariable="loanApplication"⋯/>
(27)             <receive name="receive_invokeStarLoan"
                   variable="loanOffer2"⋯/>
(25)           </sequence>
(19)         </flow>
(16)       </sequence>
(14)     </scope>
(28)     <scope name="SelectOffer" ⋯>
(30)       <if name="selectBestOffer">
             <condition>"bpws.getVariableData('loanOffer1','payload'⋯)>
                        bpws.getVariableData('loanOffer2','payload'⋯)"
(32)           <assign name="selectStarLoanOffer">
                 <copy><from variable="loanOffer2" ⋯ />
                   <to variable="selectedLoanOffer" ⋯/> </copy></assign>
             <else>
(33)           <assign name="selectUnitedLoanOffer">
                 <copy><from variable="loanOffer1" ⋯/>
                   <to variable="selectedLoanOffer" ⋯/> </copy></assign>
             </else>
(31)       </if>
(29)     </scope>
(34)     <invoke name="replyOutput" partnerLink="client"
           inputVariable="selectedLoanOffer"⋯/>
(4)    </sequence>
(2)  </process>                                                    v1.0
```

```
(1)  <process name="LoanFlow"...>
       <partnerLink>⋯</partnerLink>
       <variables>⋯</variables>
(3)    <sequence>
(5)      <receive name="receiveInput"  variable="input".../>
(6)      <scope name="GetCreditRating" ...>
(8)        <sequence>
(10)         <assign name="copySSN" />
               <copy><from variable="input" part="payload"⋯/>
                 <to variable="crInput" part="payload"⋯/> </copy>
               <copy><from>⋯
                 <Address>http://www.ratingService.com/⋯</from>
                 <to partnerLink="creditRatingService"/></copy></assign>
(11)         <invoke name="invokeCR"  inputVariable="crInput"
               outputVariable="crOutput"⋯/>
(12)         <assign name="copyRating">
               <copy><from variable="crOutput" part="payload"⋯/>
                 <to variable="input" part="payload"⋯/> </copy> </assign>
(9)        </sequence>
(7)      </scope>
(13)     <scope name="GetLoanOffer"...>
(15)       <sequence>
(17)         <assign name="prepareApplication">
               <copy><from variable="input" ⋯/>
                 <to variable="loanApplication" ⋯/> </copy> </assign>
(18)         <flow name="collectOffers">
(20)           <sequence>
(22)             <invoke name="invokeUnitedLoan"
                   inputVariable="loanApplication"⋯/>
(23)             <receive name="receive_invokeUnitedLoan"
                   variable="loanOffer1"⋯/>
(21)           </sequence>
(24)           <sequence>
(26)             <invoke name="invokeStarLoan"
                   inputVariable="loanApplication"⋯/>
(27)             <receive name="receive_invokeStarLoan"
                   variable="loanOffer2"⋯/>
(25)           </sequence>
(19)         </flow>
(16)       </sequence>
(14)     </scope>
(28)     <scope name="SelectOffer" ⋯>
(30)       <if name="selectBestOffer">
             <condition>"bpws.getVariableData('loanOffer1','payload'⋯)>
                        bpws.getVariableData('loanOffer2','payload'⋯)"
(32) (35)      <assign name="selectStarLoanOffer">
                 <copy><from variable="loanOffer1" ⋯/>
                   <to variable="selectedLoanOffer" ⋯/> </copy></assign>
             <else>
(33)           <assign name="selectUnitedLoanOffer">
                 <copy><from variable="loanOffer1" ⋯/>
                   <to variable="selectedLoanOffer" ⋯/> </copy></assign>
             </else>
(31)       </if>
(29)     </scope>
(34)     <invoke name="replyOutput" partnerLink="client"
           inputVariable="selectedLoanOffer"⋯/>
(4)    </sequence>
(2)  </process>                                                    v1.1
```

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

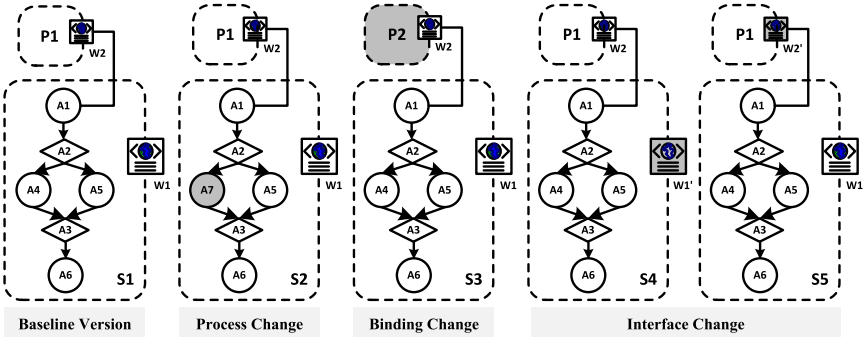


FIGURE 2. Change types of composite service.

Suppose $def(A)$ and $use(A)$ respectively denote the variables defined and used in activity A, and (x, A, B) denotes a def-use pair where B uses the definition of x in A. Fig.3 shows the different types of affected

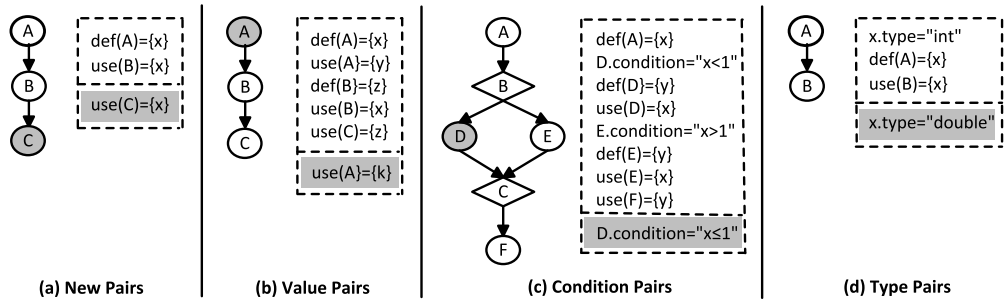


FIGURE 3. Types of affected def-use pairs.

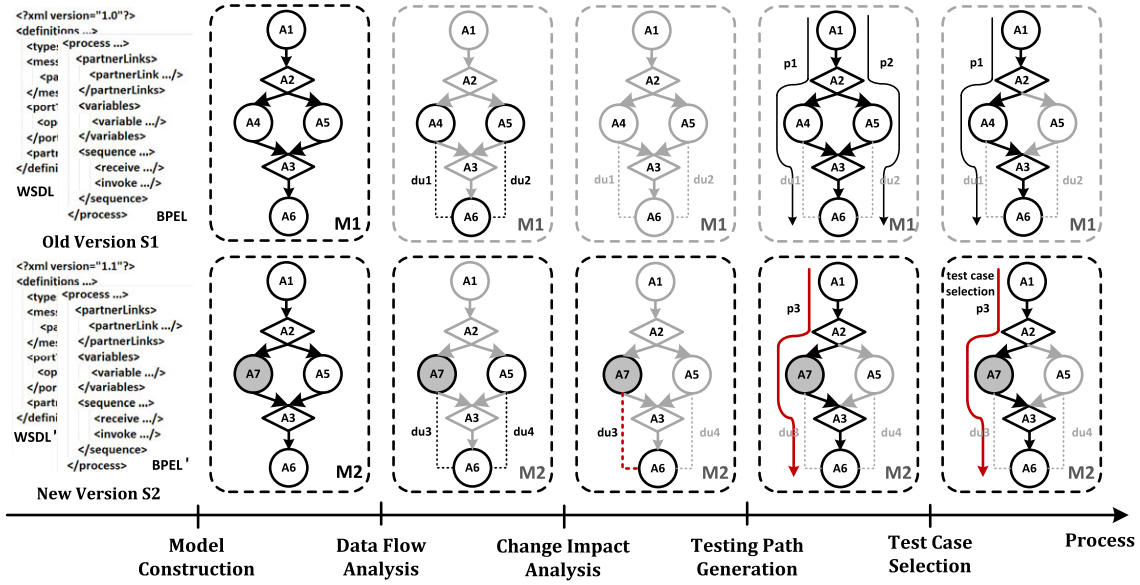


FIGURE 4. Outline of our approach.

def-use pairs, in which changes to each case are marked as grey.

- *New Pairs* are new def-use pairs created because of the insertion/deletion of definitions and uses or order change of activities. In Fig.3(a), adding activity C introduces a new use of variable x. Def-use pairs consisting of definitions of x that reach the use of x in C must be tested. The *new pair* is (x, A, C).
- *Value Pairs* are the def-use pairs whose computed values may have changed. In Fig.3(b), replacing the used variable in A introduces the changed value of x in this activity. The def-use pairs that depend on the new value of x must be retested. Since B uses the definition of x in A, def-use pair (x, A, B) is a *value pair*. The new value of x causes the computed value of z in B to be changed, and also (z, B, C) is a *value pair*. The process of identifying *value pairs* continues with the use of these variables whose computed values may have changed.
- *Condition Pairs* are the def-use pairs whose control dependency conditions have changed, where the condition can be affected by an explicit change to the predicate statement or by the value change of some variable in the predicate. In BPEL composite service, activity A is control dependent on condition c if and only if whether A will be executed or not depends on the value of the condition. And def-use pair (x, A, B) is control dependent on condition c if and only if either A is control dependent on c or B is control dependent on c. In Fig.3(c), changing the control dependency condition of D introduces *condition pairs* (x, A, D) and (y, D, F), which are control dependent on this condition. The *condition pairs* must be retested.
- *Type Pairs* are the def-use pairs whose variable types have changed, and they require retesting. In Fig.3(d), changing the type of x makes def-use pair (x, A, B) be affected. The *type pair* is (x, A, B).

C. OUTLINE OF OUR APPROACH

We propose a new approach to solve the all-ueses based regression test case selection problem of BPEL-based composite service. We will use the *process change* in Fig.4 (from $A4$ to $A7$) as an example to explain our approach. There are five key steps:

- **Model construction**. For BPEL-based composite service, a two-level model consisting of XCFG and WSDMs is created to describe the complete

composite service, where data access information is attached to XCFG elements for data flow analysis. In Fig.4, the visual XCFG models of both old version $S1$ and new version $S2$ are constructed as $M1$ and $M2$, respectively. Take $M1$ as an example, it consists of activities in process (such as $A1$) and control flow relation (such as solid line between $A1$ and $A2$).

- **Data flow analysis**. Based on generated XCFG, data flow is analyzed to calculate all def-use pairs for satisfying the all-uses criterion in the testing. In Fig.4, both def-use pairs of $M1$ and $M2$ are calculated and only two pairs in each version ($du1$ and $du2$ for $M1$, $du3$ and $du4$ for $M2$) are shown as representative in this figure, which are represented with dashed lines.
- **Change impact analysis**. Two-level model comparison based analysis is performed to detect the def-use pairs affected by *process change*, *binding change* and *interface change* to determine which def-use pairs need to be checked in the regression testing. In Fig.4, we perform change impact analysis and find out that def-use pair $du3$ is a *new pair* since *process change* has occurred in the new version.
- **Testing path generation**. With the def-use pairs identified to be tested in the all-uses based data flow testing, the data flow paths that cover these pairs are constructed as testing paths for the selection of test cases. In Fig.4, both testing paths of $M1$ and $M2$ are calculated, where paths $p1$ and $p2$ for $M1$ cover $du1$ and $du2$ and path $p3$ for $M2$ covers $du3$.
- **Test case selection**. Finally, path condition analysis is performed to determine which paths can be tested with the test cases of the baseline version. And test cases that can be reused on the new version are identified according to the mapping relation between path and test suit. In Fig.4, test cases attached with $p1$ are selected to test $p3$.

IV. TWO-LEVEL MODEL

In this section, we discuss how to define and construct the two-level model.

The control flow model XCFG was introduced to describe BPEL process in the previous work [14], [15]. It supports concurrent control flow and synchronization dependency compared with the traditional control flow graph (CFG) and can be used to identify *process change*. However, BPEL composite service is composed of process and partner services. The absence of expressing partner services makes the original XCFG model not suitable for detecting *binding change*, and *interface change*.

In order to perform change impact analysis on composite service rather than just the process, we propose the two-level model, which is composed of the revised XCFG and WSDMs. It has the following advantages: (1) the revised XCFG can describe the variables and partnerLinks defined in process; (2) the modeling of loop activity is optimized in the revised XCFG; (3) WSDMs model the WSDL documents of composite service and partner services to provide the necessary interface information.

A. MODEL DEFINITION

In the two-level model, the first level is XCFG for modeling BPEL process, and the second level is WSDM for modeling related WSDL interfaces. We will give the definitions of XCFG and WSDM respectively.

Definition 1 (XCFG): XCFG is defined as a quintuple (N, E, PL, V, F), where N is a set of XCFG nodes, E is a set of XCFG edges, PL is a set of partnerLinks, V is a set of variables, and F is the field of XCFG element. $N = N_B \cup N_S \cup N_E \cup N_C$ where N_B, N_S, N_E, N_E denote *Basic Node*, *Sequence Node*, *Exclusive Node* and *Concurrent Node*, respectively; $E = E_C \cup E_L$ where E_C and E_L denote *Control Edge* and *Link Edge*, respectively.

In BPEL, both basic activities and structural activities are transformed into XCFG nodes. N is classified into four types to describe different activities:

- *Basic Node* (N_B). It is created for basic activities in BPEL, such as `receive`, `reply`, `assign` and so on. Additionally, it is also created for `onMessage` and `onAlarm` in `pick`.
- *Sequence Node* (N_S). It is created for sequential activities in BPEL, including `process`, `sequence`, and `scope`. It has two sub types *Sequence Start Node* (N_{SS}) and *Sequence End Node* (N_{SE}) to respectively represent the start and end of sequence construct.
- *Exclusive Node* (N_E). It is created for conditional activities, including `if`, `pick`, `while`, and `repeatUntil`. N_E is divided into *Exclusive Decision Node* (N_{ED}) and *Exclusive Merge Node* (N_{EM}) to respectively represent the start and end of choice construct.
- *Concurrent Node* (N_C). It is created for `flow` activity. N_C also has *Concurrent Branch Node* (N_{CB}) and *Concurrent Merge Node* (N_{CM}) to respectively represent the start and end of concurrency construct.

The following fields are included in F for XCFG nodes to support further analysis:

- *id*: represents the node's unique identification. Its value is a natural number, which can be used to represent a XCFG node for short.
- *name*: records the name of corresponding BPEL activity. Its value is `name` attribute of the activity.
- *condition*: denotes the predicate constraint of XCFG node. It exists in the first child node of each choice branch to record the condition of corresponding branch, and its value comes from `condition` sub element of corresponding BPEL structural activity. It also exists in the target node of `link`, and its value is `joinCondition` sub element of corresponding BPEL activity.
- *outEdges*: represents the set of outgoing control edges of XCFG node. For the end node of process, the size of

outEdges equals to 0; for N_B, N_S, N_{EM}, and N_{CM}, the size equals to 1; for N_{ED} and N_{CB}, the size is greater than 1.
- *inEdges*: represents the set of incoming control edges of XCFG node. For the start node of process, the size of *inEdges* equals to 0; for N_B, N_S, N_{ED}, and N_{CB}, the size equals to 1; for N_{EM} and N_{CM}, the size is greater than 1.
- *outLinks*: denotes the set of outgoing link edges of XCFG node. It exists in node corresponding to the child activity of `flow` construct who has `source` sub element.
- *inLinks*: denotes the set of incoming link edges of XCFG node. It exists in node corresponding to the child activity of `flow` construct who has `target` sub element.
- *cReadVars*: records the computation-use (c-use) variables of the node. It exists in nodes corresponding to `assign` and interaction activity which have read access on some variable.
- *pReadVars*: records the predicate-use (p-use) variables of the node. It exists in nodes that have `condition` field, where the variables in the boolean expression are extracted.
- *writeVars*: records the definition variables of the node. It exists in nodes corresponding to `assign` and interaction activity that have write access on variable.
- *partnerLink*/*portType*/*operation*: denote the partner service the node interact with and the interface used in the interaction. They only exist in node corresponding to interaction activity. And their values are `partnerLink`, `portType` and `operation` attributes of the activity.

In addition, the connection between activities defined in BPEL can be transformed into XCFG edges. E is classified into following two types:

- *Control Edge* (E_C). It is created for control flow of XCFG nodes to represent the execution logic of activities.
- *Link Edge* (E_L). It is created for *links* between XCFG nodes to represent synchronization dependencies among concurrent activities.

The following fields are included in F for XCFG edges:

- *id*: represents the edge's unique identification. Its value is a natural number, which can be used to represent a XCFG edge for short.
- *source*: records the set of precedent nodes of both E_C and E_L.
- *target*: records the set of subsequent nodes of both E_C and E_L.
- *type*: denotes the type of E_C, where the types of control edges are distinguished into *SEQUENCE*, *CHOICE*, and *CONCURRENCY* to respectively represent sequence, choice and concurrency dependency between two activities.
- *name*: represents the name of E_L. Its value is the *name* attribute of corresponding `link`.
- *condition*: records `transitionCondition` of E_L, which determines the status of corresponding `link`.

PL describes the `partnerLinks` defined in BPEL process, for which the following fields are required in F:
- *name*: denotes the name of partnerLink. Its value is the `name` attribute of corresponding `partnerLink`.
- *endpoint*: denotes the binding address of partnerLink which determines the actual partner service.

V describes the `variables` defined and used in BPEL process, for which the following fields are required in F:
- *name*: denotes the name of variable. Its value is the `name` attribute of corresponding *variable*.
- *type*: denotes the type of variable. The type can be WSDL message type, XML Schema built-in type or user-defined complex type.
- *location*: denotes the namespace URI that contains the type definition of variable.

Definition 2 (WSDM): WSDM is defined as a quadruple (M, EM, CT, F), where M is a set of messages, EM is a set of elements, CT is a set of complexTypes, and F is the field of WSDM model and WSDM element.

For WSDM model, the following fields are required in F:
- *name*: records the name of WSDM model. Its value is the `name` attribute in `definitions` root element of corresponding WSDL document.
- *namespace*: denotes the namespace of WSDM model. Its value is the `targetNamespace` attribute in corresponding WSDL document.

The `messages` defined in WSDL are characterized by M. The following fields are included in F of M:
- *name*: records the name of WSDM message. Its value is the `name` attribute of corresponding `message`.
- *elements*: denotes the type of WSDM message whose `parts` are defined with `element` attribute in WSDL. Its value is a set of `element` attributes of `parts`.
- *types*: denotes the type of WSDM message whose `parts` are defined with *type* attribute in WSDL. Its value is a set of `type` attributes of `parts`.

The `elements` defined in `types` are characterized by EM. The following fields are included in F of EM:
- *name*: records the name of WSDM element. Its value is the `name` attribute of corresponding `element`.
- *type*: denotes the type of WSDM element. Its value is the `type` attribute of corresponding `element`.

The `complexTypes` defined in `types` are characterized by CT. The following fields are included in F of CT:
- *name*: records the name of WSDM complexType. Its value is the `name` attribute of corresponding `complexType`.
- *types*: denotes the type of WSDM complexType which are defined with `type` attribute in WSDL. Its value is a set of `type` attributes contained in corresponding `complexType`.
- *refs*: denotes the type of WSDM complexType which are defined with `ref` attribute in WSDL. Its value

is a set of `ref` attributes contained in corresponding `complexType`.

B. MODEL CONSTRUCTION

For the two-level model, XCFG and WSDMs are constructed separately.

The modeling of XCFG is performed with the help of open source package *org.apache.ode.bpel.compiler.bom* which can automatically parse BPEL composite service to BOM (BPEL Object Model). Based on BOM, the process of XCFG construction consists of the following four steps:

(1) Create *PL* and *V* for all partnerLinks and variables respectively;

(2) Create XCFG nodes for BPEL activities;

(3) Create E_C according to the execution order of activities;

(4) Create E_L according to the synchronization dependency between concurrent activities.

In the first step, `partnerLinks` and `variables` are transformed to *PL* and *V* respectively. For `partnerLink`, a *pl* is created where the value of *pl*.*endpoint* can be acquired by the analysis of *assign* activity which copies endpoint reference to `partnerLink`. For `variable`, a *v* is created where *v.type* is `messageType` attribute, or `type` attribute, or `element` attribute of `variable`.

Then, XCFG nodes and edges are constructed to depict the behavior of BPEL process. The construction of edges are accompanied with the construction of nodes. And the fields of XCFG nodes can be acquired by invoking the corresponding operations BOM classes provide. The transformation methods for activities are described below and Fig.5 gives some typical transformations of BPEL activity snippets for better understanding. We first give the transformation methods of following basic activities.

- `invoke` activity. A N_B *n* is created where values of *n.name*, *n.cReadVars*, *n.writeVars*, *n.partnerLink*, *n.portType* and *n.operation* are attributes `name`, `inputVariable`, `outputVariable`, `partnerLink`, `portType`, and `operation` of `invoke`, respectively.
- `receive` activity. A N_B *n* is created where value of *n.writeVars* is `variable` attribute of `receive` and *n.cReadVars* is ∅ since `receive` does not have read access on any variable.
- `reply` activity. A N_B *n* is created where value of *n.cReadVars* is `variable` attribute of `reply` and *n.writeVars* is ∅ since *reply* does not have write access on any variable.
- `assign` activity. A N_B *n* is created. If it is used for assignment of variable, then the values of *n.cReadVars* and *n.writeVars* are respectively set with the variables in `from` and `to` of each `copy` element. If it is used for assignment of partnerLink, then for *pl* corresponding to the `partnerLink` attribute in `to`, the value of *pl.endpoint* is updated with the address in `from`.

For structural activities, the transformation is focused on the structure itself. We give the transformation methods of following structural activities.

- `sequence` activity. A pair of N_S *ssn* and *sen* is created. The values of *ssn.name* and *sen.name* are respectively assigned as *name_start* and *name_end* where *name* is the `name` attribute of `sequence`. The child activities contained in `sequence` are traversed sequentially. E_C *ecs* of type *SEQUENCE* are built up to describe the sequential relation among *ssn*, nodes corresponding to child activities, and *sen*. For `process` and `scope`, they are processed the same way as `sequence`.
- `if` activity. A pair of N_E *edn* and *emn* is created. The values of *edn.name* and *emn.name* are respectively assigned as *name_start* and *name_end* where *name* is the `name` attribute of `if`. The condition of each branch in *if* is recorded in *condition* field of the first node in each branch. E_C *ecs* of type *CHOICE* are built up according to the branch relations among *edn*, branch nodes and *emn*. Activity `pick` is processed the same way as `if`.
- `while` activity. To simplify the analysis of loop, meanwhile to keep record of the full data flow information in the model, `while` is molded as choice structure with three branches. The three branches respectively describe that child activities of `while` be executed for zero time, one time and two times. For a `while` activity whose `condition` attribute is *c*, a pair of N_E *edn* and *emn* is created. The values of *edn.name* and *emn.name* are respectively assigned as *name_start* and *name_end* where *name* is the `name` attribute of `while`. In the first branch, a N_B corresponding to `empty` activity is created and the *condition* field is set as !*c*. In the second branch, nodes corresponding to child activities in `while` are created and the *condition* field of the first node is set as *c*. In the third branch, nodes in the second branch are doubly created and the *condition* field of the first node is *c*. Finally E_C *ecs* of type *CHOICE* are built up among *edn*, branch nodes and *emn*.
- `repeatUntil` activity. The child activities contained in `repeatUntil` will be at least executed for once no matter the condition is satisfied or not. For a `repeatUntil` activity whose `condition` attribute is *c*, nodes corresponding to the child activities of `repeatUntil` are created firstly. Then a pair of N_E *edn* and *emn* is created with two branches. The values of *edn.name* and *emn.name* are respectively assigned as *name_start* and *name_end* where *name* is the `name` attribute of `repeatUntil`. In the first branch, a N_B corresponding to `empty` activity is created and the *condition* field is set as *c*. In the second branch, nodes corresponding to child activities in `repeatUntil` are created and *condition* field of the first node is set as !*c*. Finally E_C *ecs* of type *CHOICE* are built up among *edn*, branch nodes and *emn*.
- `forEach` activity. It is dealt with in the same way with `while` activity, except that the condition of child

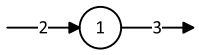
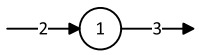
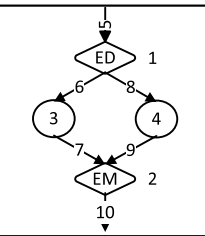
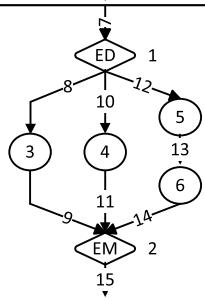
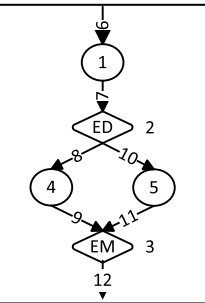
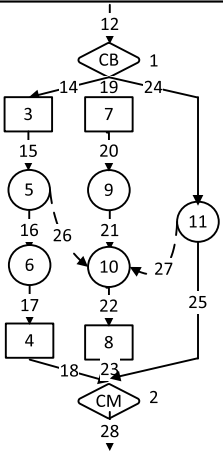
	BPEL Activity	XCFG Construction	Field
Basic Activity	**<invoke** name="A"⋯ inputVariable="X" outputVariable="Y"/>		The field of *invoke:* id=1, name=A, outEdges={3}, inEdges={2}, cReadVars={X}, writeVars={Y}
	<assign name="A"> <copy> <from>X</from> <to>Y</to><copy> </assign>		The field of *assign:* id=1, name=A, outEdges={3}, inEdges={2},cReadVars={X}, writeVars={Y}
Structural Activity	**<if** name="A"> <condition>e</condition> <activity name="B">⋯</activity> <else> <activity name="C"> ⋯ </activity> </else> </if> Note: The same is applied to <pick>.		The field of branches in if: 1)id=3, name=A, outEdges={7}, inEdges={6}, condition=e 2)id=4, name=B, outEdges={9}, inEdges={8}, condition=!e The field of edges from ED: 1)id=6, source=1, target=3, type=CHOICE 2)id=8, source=1, target=4, type=CHOICE
	<while name="A"> <condiiton>e</condition> <activity name="B"> ⋯⋯ </activity> </while>		The field of branches in while: 1)id=3, outEdges={9}, inEdges={8}, condition=!e 2)id=4, name=B, outEdges={11}, inEdges={10}, condition=e 3)id=5, name=B, outEdges={13}, inEdges={12}, condition=e 4)id=6, name=B, outEdges={14}, inEdges={13} The field of edges from ED: 1)id=8, source=1, target=3, type=CHOICE 2)id=10, source=1, target=4, type=CHOICE 3)id=12, source=1, target=5, type=CHOICE 4)id=13, source=5, target=6, type=SEQUENCE
	<repeatUntil name="A"> <activity name="B"> ⋯⋯ </activity> <condiiton>e</condition> </repeatUntil>		The field of node 1: id=1, name=B, outEdges={7}, inEdges={6} The field of branches in repeatUntil: 1)id=4, outEdges={9}, inEdges={8}, condition=e 2)id=5, name=B, outEdges={11}, inEdges={10}, condition=!e The field of edges from ED: 1)id=7, source=1, target=2, type=SEQUENCE 2)id=8, source=2, target=4, type=CHOICE 3)id=10, source=2, target=5, type=CHOICE
	<flow name="A"> <links> <link name="BtoF"/> <link name="GtoF"/> </links> <sequence name="B"> <activity name="C"⋯> <source linkName="BtoF" transitionConsition="e1"/> </activity> <activity name="C"⋯/> </sequence> <sequence name="D"> <activity name="E"⋯/> <activity name="F"⋯/> < joinCondition="e1 or e2"/> <target linkName="BtoF"/> <target linkName="GtoF"/> </activity> </sequence> <activity name="G"⋯> <source linkName="GtoF" transitionCondition="e2"/> </activity> </flow>		The field of node CB: id=1, name=A_start, outEdges={14,19,24},inEdges={12} The field of activity F: id=10, name=F, outEdges={22}, inEdges={21}, inLinks={26,27}, condition=e1 or e2 The field of edges from CB: 1)id=14, source=1, target=3, tyep=CONCURRENCY 2)id=19, source=1, target=7, type=CONCURRENCY 3)id=24, source=1, target=7, type=CONCURRENCY The field of link "BtoF": id=26, name=BtoF, source=5, target=10, condition=e1
Legend	Control Edge; Link Edge	Normal Node; Sequence Node	Exclusive Decision Node; Exclusive Merge Node; Concurrency Branch Node; Concurrency Merge Node

FIGURE 5. XCFG construction for BPEL activities.

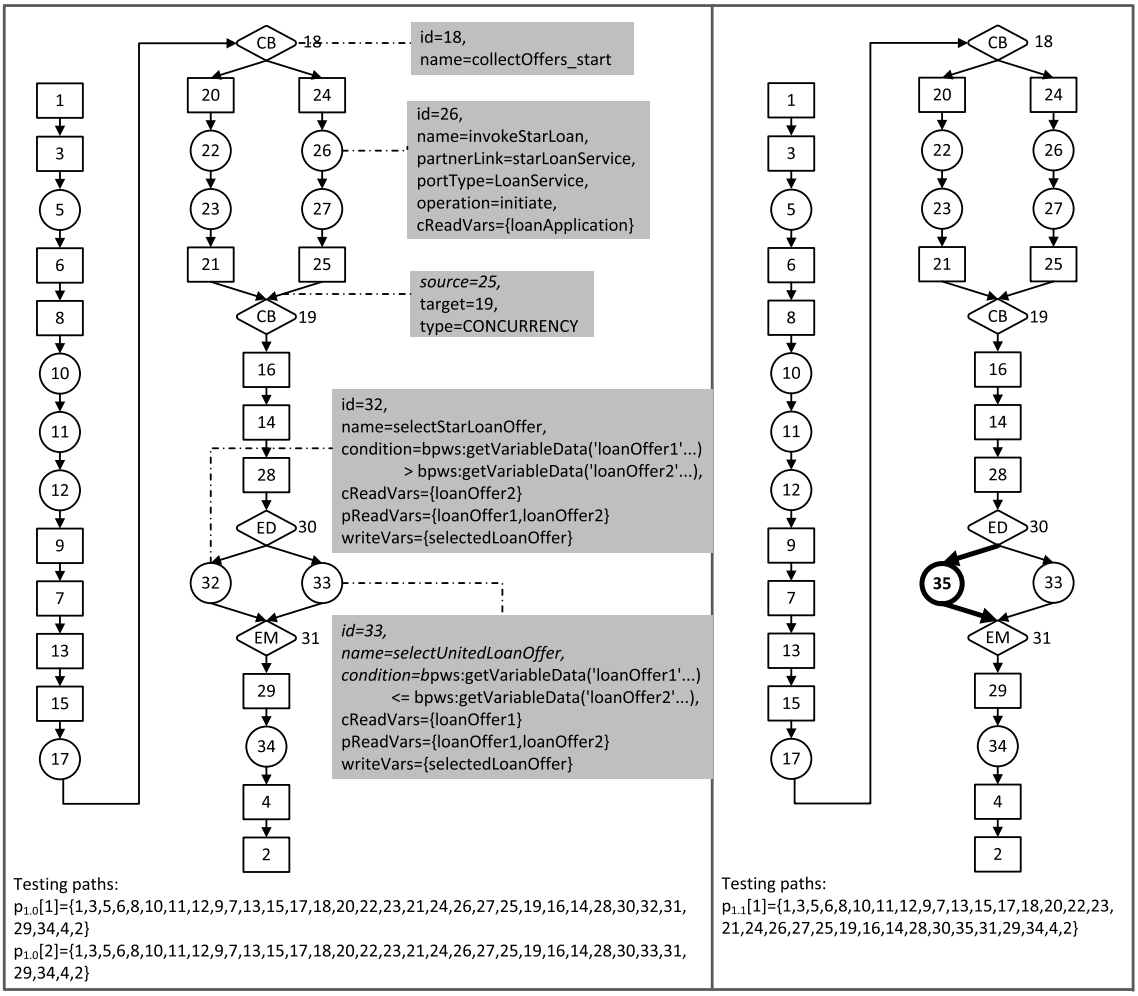


FIGURE 6. The graphical XCFG of LCS v1.0 and v1.1.

activity is *startCounterValue* ≤ *counterName* ≤ *finalCounterValue*.

- `flow` activity. A pair of N_C *cbn* and *cmn* is created. The values of *cbn.name* and *cmn.name* are respectively assigned as *name_start* and *name_end* where *name* is the `name` attribute of `flow`. Concurrency branches are constructed according to the child activities of `flow`. And E_C *ecs* of type *CONCURRENCY* are built up among *cbn*, branch nodes and *cmn*. Finally, E_L *el*s are constructed for *link*s defined in `flow`. The source node and target node of *el* can be identified through the analysis of each activity since BOM provides *getLinkSources()* and *getLinkTargets()* operations to respectively get *outLinks* and *inLinks* fields of corresponding node.

Fig.6(a) shows the XCFG of *LCS v1.0*. The number attached to each XCFG node is its *id* field which corresponds to the line number of BPEL document in Fig.1. And the detailed fields of some typical XCFG elements are listed in gray boxes, which are extracted from corresponding BPEL activities according to our transformation algorithms. The XCFG of *LCS v1.1* is also shown in Fig.6(b), where the modification of `assign` activity in *v1.0* is reflected with the bold XCFG elements.

The modeling of WSDM is performed with the help of open source package *wsdl4j.jar* which can automatically parse WSDL document. Based on the parsing, *name* and *namespace* fields of the WSDM can be acquired by identifying `name` and `targetNamespace` attributes defined in the WSDL. Also, elements defined in WSDL, including `message`, `element`, `complexType`, and so on, can be identified through the corresponding operations that *wsdl4j.jar* provides.

- For each `message` defined in WSDL, a *m* ($m \in M$) is created where *name* field can be acquired through *getQName()* operation of `message`, and *elements* and *types* fields can be acquired by collecting the values of corresponding attributes in `parts` enclosed in the `message`.
- For each `element` enclosed in `types`, a *em* ($em \in EM$) is created where *name* and *type* fields can be

acquired through *getNodeValue()* operation of corresponding attribute.

- For each `complexType` enclosed in `types`, a *ct* ($ct \in CT$) is created where *name* field can be acquired through *getNodeValue()* of `name` attribute, *types* field and *refs* field can be acquired by collecting the values of `type` attributes and `ref` attributes in `elements` enclosed in the `complexType` respectively.

C. DATA FLOW ANALYSIS

Based on the definition of XCFG in the two-level model, all the def-use pairs defined in BPEL process can be computed by reaching definitions based data flow analysis. The definitions that may reach a program point along some path are known as reaching definitions [16]. For traditional program, transfer equation and control-flow equation are provided to calculate the reaching definitions based on CFG.

However, the equations cannot support XCFG based reaching definitions analysis of BPEL process. CFG based control-flow equation only considers sequence and choice control flow, while concurrency and synchronization dependency between concurrent activities are included in BPEL specification and corresponding XCFG model. For this reason, the improved equations for XCFG based reaching definitions are defined as follows:

$$Out(n) = Gen(n) \cup (In(n) - Kill(n))$$
$$In(n) = \bigcup_{p \in pred(n)} Out(p) - \bigcup_{m \in K} Kill(m)$$

where $In(n)$ and $Out(n)$ denote the reaching definitions at the point before and after node n, $Gen(n)$ denotes definitions generated by n, $Kill(n)$ denotes the definitions killed by the definition in n, $pred(n)$ denotes predecessor nodes of n, and K is especially proposed for the end node n of concurrency structure and the target node n of link to denote nodes that are contained in the concurrency structure and will definitely be executed before n.

Detailed algorithm to get the solutions and correctness analysis of the improved equations can be referred to [17].

Suppose (x, n) denotes a definition which means variable x is defined in node n. With the reaching definitions of each node, def-use pair (x, n, n') is constructed if definition (x, n) can reach the point before n' meanwhile it is used in n'. For any variable x, all its def-use pairs $du - pairs(x)$ can be computed according to the following equation, where $def(n)$ and $use(n)$ denote the variables defined and used in node n respectively.

$$du - pairs(x) = \{(x, n, n') |$$
$$x \in def(n) \wedge (x, n) \in In(n') \wedge x \in use(n')\}$$

Table 2 shows the def-use pairs in *LCS v1.0* and *v1.1*, in which filed *id* is used as the representation of each XCFG node. In addition, the def-use pairs are classified into two types for further change impact analysis: *Computation* and *Predication*. For def-use pair $du = (x, n, n')$, if variable occurrence in n' is computation-use, then du is of type *Computation*. If variable occurrence in n' is predicate-use, then du is of type *Predicate*.

V. CHANGE IMPACT ANALYSIS

Any change of composite service can bring impact to def-use pairs. In this section, we will discuss how to perform the change impact analysis to identify the affected def-use pairs, including *new pairs*, *value pairs*, *condition pairs* and *type pairs*, which covers *process change*, *binding change* and *interface change*.

A. BASIC IDEA

Let $S_1, S_2 \ldots, S_n$ denote n different versions of composite service, and $\triangle S_i$ denote the changes from S_i to S_{i+1}, then we have

$$S_{i+1} = S_i + \triangle S_i \quad (1 \leq i \leq n-1)$$

Let M_i denote the two level model of S_i and $\triangle M_i$ denote the changes from M_i to M_{i+1}, then we have

$$M_{i+1} = M_i + \triangle M_i \quad (1 \leq i \leq n-1)$$

Furthermore, let $\triangle S_i^p$, $\triangle S_i^b$, and $\triangle S_i^i$ respectively denote *process change*, *binding change* and *interface change* from S_i to S_{i+1}, then we have

$$\triangle S_i = \triangle S_i^p \cup \triangle S_i^b \cup \triangle S_i^i$$

Accordingly, let $\triangle M_i^p$, $\triangle M_i^b$, and $\triangle M_i^i$ denote corresponding types of changes from M_i to M_{i+1}, then we have

$$\triangle M_i = \triangle M_i^p \cup \triangle M_i^b \cup \triangle M_i^i$$

Suppose DU_{i+1} and DU_{i+1}^s respectively denote all the def-use pairs and the modified def-use pairs of M_{i+1}, where $DU_{i+1}^s \subseteq DU_{i+1}$. Let DU_{i+1}^{sp}, DU_{i+1}^{sb}, and DU_{i+1}^{si} respectively denote the def-use pairs of M_{i+1} influenced by *process change*, *binding change*, and *interface change*, then we have

$$DU_{i+1}^s = DU_{i+1}^{sp} \cup DU_{i+1}^{sb} \cup DU_{i+1}^{si}$$

- *process change*: Change of activities and change of execution order will cause *new pairs*, *value pairs* and *condition pairs*. In addition, change of variables will make *type pairs* when the change happens to the type of the variable. Let DU_{i+1}^{spn}, DU_{i+1}^{spv}, DU_{i+1}^{spc} and DU_{i+1}^{spt} denote *new pairs*, *value pairs*, *condition pairs* and *type pairs* caused by *process change*, then we have

$$DU_{i+1}^{sp} = DU_{i+1}^{spn} \cup DU_{i+1}^{spv} \cup DU_{i+1}^{spc} \cup DU_{i+1}^{spt}$$

- *binding change*: Change of endpoint address of partner service will make *value pairs* since it may reply new value to some variable in the interaction between process of the partner service. And it may further cause *condition pairs* since the variable value may be changed in the *condition* element. Let DU_{i+1}^{sbv} and DU_{i+1}^{sbc} denote *value*

pairs and *condition pairs* caused by *binding change*, then we have

$$DU_{i+1}^{sb} = DU_{i+1}^{sbv} \cup DU_{i+1}^{sbc}$$

- *interface change*: Change of operations and portTypes that will be reflected in activities of BPEL process is taken into consideration in the analysis of *process change*. If change of some operations and portTypes is not reflected in BPEL process, which means it is irrelative with the composition process, then there is no need to handle with this kind of change. For *interface change*, we mainly focus the change of information that is related with the composition process and not available in BPEL document, which consists of `types` change and `message` change. It will make *type pairs* since the concrete type of variable may be influenced. Let DU_{i+1}^{sit} denote the *type pairs* caused by *interface change*, then we have

$$DU_{i+1}^{si} = DU_{i+1}^{sit}$$

Let DU_{i+1}^{sn}, DU_{i+1}^{sv}, DU_{i+1}^{sc} and DU_{i+1}^{st} denote *new pairs*, *value pairs*, *condition pairs* and *type pairs* caused by changes in the evolution, then we have

$$DU_{i+1}^{s} = DU_{i+1}^{sn} \cup DU_{i+1}^{sv} \cup DU_{i+1}^{sc} \cup DU_{i+1}^{st}$$

The steps of change impact analysis is comprised of *new pairs* identification, *value pairs* identification, *condition pairs* identification and *type pairs* identification. According to above analysis, DU_{i+1}^{sn}, DU_{i+1}^{sv}, DU_{i+1}^{sc} and DU_{i+1}^{st} come from one or two of *process change*, *binding change* and *interface change*, as shown in the following equations. So change impact analysis is performed based on the two-level model comparison which can reveal the three kinds of changes.

$$\begin{aligned} DU_{i+1}^{sn} &= DU_{i+1}^{spn} \\ DU_{i+1}^{sv} &= DU_{i+1}^{spv} \cup DU_{i+1}^{sbv} \\ DU_{i+1}^{sc} &= DU_{i+1}^{spc} \cup DU_{i+1}^{sbc} \\ DU_{i+1}^{st} &= DU_{i+1}^{spt} \cup DU_{i+1}^{sit} \end{aligned}$$

The following sections will present how to identify the affected def-use pairs DU_{i+1}^{s} in S_{i+1} compared with the baseline version S_i.

B. NEW PAIRS IDENTIFICATION

New pairs caused by *process change* are identified by comparing the def-use pairs in DU_i and DU_{i+1} one by one, in which the compared pairs are of the same type: *Computation* or *Predicate*. The def-use pairs that exist in DU_{i+1} while not in DU_i are *new pairs*. So we have

$$DU_{i+1}^{sn} = \{du | (du \in DU_{i+1}) \wedge !(du \in DU_i)\}$$

Distinguishing the types of def-use pairs is required in the def-use pair comparison. *New pairs* are comprised of *Computation* type and *Predicate* type, which are denoted as DU_{i+1}^{snc} and DU_{i+1}^{snp} respectively. We have

$$DU_{i+1}^{sn} = DU_{i+1}^{snc} \cup DU_{i+1}^{snp}$$

Suppose there are corresponding def-use pairs $du_i = (x_i, d_i, u_i)$ and $du_{i+1} = (x_{i+1}, d_{i+1}, u_{i+1})$, where $du_i \in DU_i$ and $du_{i+1} \in DU_{i+1}$. If du_{i+1} and du_i are of *Computation* type, then du_{i+1} is equals with du_i iff the following condition is satisfied:

$$(x_{i+1} == x_i) \wedge (d_{i+1} == d_i) \wedge (u_{i+1} == u_i)$$

If du_{i+1} and du_i are of *Predicate* type, then du_{i+1} is equals with du_i iff the following condition is satisfied:

$$\begin{aligned} &(x_{i+1} == x_i) \wedge (d_{i+1} == d_i) \\ &\qquad \wedge (start_{i+1} == start_i) \wedge (num_{i+1} == num_i) \end{aligned}$$

where $start_i$ is the start node of the choice structure that contains u_i, and $num_i = k$ means u_i exists in the kth branch of the choice structure.

In the def-use pair comparison, variable comparison and node comparison are performed by comparing the fields of corresponding elements. For variables, x_{i+1} is equals with x_i iff the following condition is satisfied:

$$(x_{i+1}.name == x_i.name) \wedge (x_{i+1}.type == x_i.type)$$

And for nodes, n_{i+1} is equals with n_i iff the following condition is satisfied:

$$\begin{aligned} &(n_{i+1}.name == n_i.name) \\ &\quad \wedge (n_{i+1}.partnerLink == n_i.partnerLink) \\ &\quad \wedge (n_{i+1}.portType == n_i.portType) \\ &\quad \wedge (n_{i+1}.operation == n_i.operation) \\ &\quad \wedge (n_{i+1}.cReadVars == n_i.cReadVars) \\ &\quad \wedge (n_{i+1}.writeVars == n_i.writeVars) \end{aligned}$$

By comparing the def-use pairs of *LCS v1.1* with that of *v1.0* as shown in Table 2, *new pairs* of *Computation* type $(loanOffer1, 23, 35)$ and $(selectedLoanOffer, 35, 34)$ are found in *v1.1*.

C. VALUE PAIRS IDENTIFICATION

The main task of *value pairs* identification is to find the XCFG nodes in which the defined variable value is changed. Recording such nodes in the set *vNodes*, for any def-use pair $du = (x, n, n')$, where $du \in DU_{i+1}$ and $n \in vNodes$, du is a *value pair* and should be added into DU_{i+1}^{sv}.

There are three cases that will make the computed variable value be changed in node n:

- A new variable is used in the definition, which means $n.readVars$ is changed.
- A new function is taken for the assignment to the defined variable, which means $n.partnerLink$, $n.portType$ or $n.operation$ is changed or the three fields stay unchanged while the actual invoked object is changed.

- The value of the used variable is changed, which means $n.readVars$ stays unchanged while the value of some variable in $n.readVars$ is changed.

And the value change of computed variable will propagate along the definition and use relationship.

For *process change*, all the above three situations may happen. The first and second cases are reflected in the *new pairs* DU^{sn}_{i+1}. *New pairs* of *Computation* type will consequently cause the third case. With DU^{sn}_{i+1}, we can get the initial set *vNodes*, which are the nodes corresponding to the third case. For any def-use pair $(x, n, n') \in DU^{sn}_{i+1}$, if $n'.writeVars! = \emptyset$, then n' should be added into *vNodes* since the value of x used in n' is changed.

For *binding change*, although only partner service with the same functionality can be chosen to replace the original one, it is not definite whether the new partner service provide expected functionality in the composition process. So if *binding changes* happens and an activity defines some variable by interacting with the corresponding partner service, it is regarded as the second case. Let PL_i denote the set of partnerLinks in M_i. For some partnerLink $pl_{i+1} \in PL_{i+1}$ and corresponding partnerLink $pl_i \in PL_i$, *binding change* happens if the following condition is satisfied:

$$(pl_{i+1}.name == pl_i.name) \wedge (pl_{i+1}.endpoint! = pl_i.endpoint)$$

With the identified partnerLink $pl \in PL_{i+1}$ whose binding is changed, for some unchanged node n in XCFG model of M_{i+1}, n should be added into *vNodes* if the following condition is satisfied:

$$(n.partnerLink == pl) \wedge (n.writeVars! = \emptyset)$$

Finally, the propagation of value change needs to be analyzed to find all the nodes corresponding to the third case with the initial *vNodes*. For $\forall n \in vNodes$, if there exists some def-use pair $(x, n, n') \in DU_{i+1}$, meanwhile $n'.writeVars! = \emptyset$, then value change happens in n' too so that n' should be added into *vNodes*.

Algorithm 1 depicts the process of computing *nVnodes* and identifying the *value pairs* DU^{sv}_{i+1}. The direct impact brought by *process change* and *binding change* initializes *nVnodes* through analyzing DU^{sn}_{i+1} and *plChangeSet* which records the partnerLinks whose binding addresses are changed. Then iterative propagation analysis of value change is performed until all the nodes in which the value of used variable is changed are found. Meanwhile, the def-use pairs whose computed values are changed are identified.

D. CONDITION PAIRS IDENTIFICATION

The identification of *conditional pairs* DU^{sc}_{i+1} needs to recognize the changed conditions first. Suppose the set of nodes whose condition fields are changed is denoted as *cNodes*. Then control dependency analysis is performed to find the def-use pairs that are control dependent on the conditions in *cNodes*.

Algorithm 1 ValpairIdentification( )

Input M_i, M_{i+1}: models of S_i and S_{i+1};
Input DU_i, DU_{i+1}: def-use pairs of S_i and S_{i+1};
Output *vNodes*: nodes whose computed values are changed;
Output DU^{sv}_{i+1}: value pairs in S_{i+1};
compare DU_{i+1} and DU_i to compute *new pairs* in S_{i+1}: DU^{sn}_{i+1};
$DU_{i+1} = DU_{i+1} - DU^{sn}_{i+1}$;
if $DU_{i+1}! = \emptyset$ **then**
  for $du \in DU^{sn}_{i+1}$ **do**
    get the use node of *du*: *useNode*;
    if $useNode.writeVars! = \emptyset$ **then**
      $vNodes = vNodes \cup \{useNode\}$;
    end if
  end for
  get all the partnerLinks in XCFG models of M_i and M_{i+1}: PL_i and PL_{i+1};
  for $pl_i \in PL_i$ **do**
    for $pl_{i+1} \in PL_{i+1}$ **do**
      if $(pl_{i+1}.name == pl_i.name) \wedge (pl_{i+1}.endpoint! = pl_i.endpoint)$ **then**
        $plChangeSet = plChangeSet \cup \{pl_{i+1}\}$;
      end if
    end for
  end for
  get all the nodes in XCFG models of M_i and M_{i+1}: $nodeSet_i$ and $nodeSet_{i+1}$;
  for $n \in nodeSet_{i+1}$ **do**
    for $pl \in plChangeSet$ **do**
      if $!(n \in nodeSet_i) \wedge (n.partnerLink == pl.name) \wedge (n.writeVars! = \emptyset)$ **then**
        $vNodes = vNodes \cup \{n\}$;
      end if
    end for
  end for
  $temp = vNodes$;
  while $temp! = \emptyset$ **do**
    get a node from *temp* and delete it from *temp*: *n*;
    for $du \in DU_{i+1}$ **do**
      get the define node and use node of *du*: *defNode* and *useNode*;
      if $defNode == n$ **then**
        $DU^{sv}_{i+1} = DU^{sv}_{i+1} \cup \{du\}$;
        if $(du.type == \text{“}Computation\text{”}) \wedge (useNode.writeVars! = \emptyset) \wedge !(useNode \in vNodes)$ **then**
          $vNodes = vNodes \cup \{useNode\}$;
          $temp = temp \cup \{useNode\}$;
        end if
      end if
    end for
  end while
end if

There are two cases for the changed condition:

- The condition stays unchanged, while the value of some predicate variable in the condition is changed.
- The condition is changed explicitly in which the predicate variable is changed or the operator is changed.

The nodes corresponding to the first case can be easily found based on *new pairs* DU_{i+1}^{sn} and *value pairs* DU_{i+1}^{sv}. If def-use pair $du = (x, n, n')$ is of *Predicate* type, where $du \in (DU_{i+1}^{sn} \cup DU_{i+1}^{sv})$, then the use node n' is regarded as the node whose condition is changed because of the value change of predicate variable x. And n' should be added into *cNodes*. For the second case, comparison between conditions $cSet_{i+1}$ in M_{i+1} and $cSet_i$ in M_i is performed to find the conditions that have explicit changes. For condition $cond_{i+1} \in cSet_{i+1}$, if there is no corresponding condition in $cSet_i$, the node whose *condition* field is $cond_{i+1}$ should be added into *cNodes*. If there is corresponding condition $cond_i = cond_{i+1}$, it requires to check whether the nodes that control depend on $cond_i$ have been changed or not. For def-use pair $du = (x, n, n')$, where $du \in DU_{i+1}$, if node n or n' control depends on $cond_{i+1}$ while not on $cond_i$, which means the control dependency condition of du is changed, then du should be added into DU_{i+1}^{sc}. Finally, with *cNodes* computed through the analysis of the two cases, the nodes that control depend on the conditions in *cNodes* are computed so that *condition pairs* are identified. For the changed condition *cond*, it is recorded in the first branch node n of some choice structure. All the nodes in the branch are control dependent on *cond*, which can be found by depth-first traversal from n to the end node of the choice structure. The process of identifying *condition pairs* is depicted in Algorithm 2.

Algorithm 2 CondpairIdentification( )

Input M_i, M_{i+1}: models of S_i and S_{i+1};
Input DU_{i+1}, DU_{i+1}^{sn}, DU_{i+1}^{sv}: def-use pairs, *new pairs* and *value pairs* in S_{i+1};
Output DU_{i+1}^{sc}: condition pairs in S_{i+1};
$DU_{i+1} = DU_{i+1} - DU_{i+1}^{sv}$;
for $du \in (DU_{i+1}^{sn} \cup DU_{i+1}^{sv})$ **do**
  if $du.type ==$ "*Predicate*" **then**
    get the use node of du: *useNode*;
    $cNodes = cNodes \cup \{useNode\}$;
  end if
end for
get all the conditions in M_i and M_{i+1}: $cSet_i$ and $cSet_{i+1}$;
for $cond_{i+1} \in cSet_{i+1}$ **do**
  get the node whose *condition* field is *cond*: *node*;
  if there does not exist corresponding $cond_i \in cSet_i$ **then**
    $cNodes = cNodes \cup \{node\}$;
  else
    compute the nodes that control depend on $cond_i$ and $cond_{i+1}$: $depSet_i$ and $depSet_{i+1}$;
    for $du \in DU_{i+1}$ **do**
      get the define node and use node of du: *defNode* and *useNode*;
      if $((defNode \in depSet_{i+1}) \wedge !(defNode \in depSet_i)) \vee ((useNode \in depSet_{i+1}) \wedge !(useNode \in depSet_i))$ **then**
        $DU_{i+1}^{sc} = DU_{i+1}^{sc} \cup \{du\}$;
      end if
    end for
  end if
end for
for $n \in cNodes$ **do**
  computes the nodes that control depend on n: *depSet*;
  for $du \in DU_{i+1}$ **do**
    get the define node and use node of du: *defNode* and *useNode*;
    if $(defNode \in depSet) \vee (useNode \in depSet)$ **then**
      $DU_{i+1}^{sc} = DU_{i+1}^{sc} \cup \{du\}$;
    end if
  end for
end for

E. TYPE PAIRS IDENTIFICATION

The main task of *type pairs* identification is to find the variables whose types are changed. Suppose the set of such variables is denoted as *tVars*. For any variable $v \in tVars$, def-use pairs of v in DU_{i+1} should be added into DU_{i+1}^{st}.

There are two cases for the changed variable type:

- The variable type is explicitly changed in BPEL document, where the *name* field of variable stays unchanged and the *type* field is changed.
- The description of variable stays unchanged in BPEL document, while the concrete type of variable is changed.

The first case can be easily identified. Let V_i denote the set of variables in M_i. For some variable $v_{i+1} \in V_{i+1}$ and corresponding variable $v_i \in V_i$, if $v_{i+1}.name == v_i.name$ and $v_{i+1}.type != v_i.type$, then the type of v_{i+1} is changed and v_{i+1} should be added into *tVars*.

For the second case, the concrete type of variable is not available from BPEL process. It is necessary to extract the concrete information by importing corresponding WSDL documents. Fig.7 describes how to obtain the concrete type of variable, which takes variable *loanApplication* as an example to illustrate the process of locating concrete type by the labeled dashed arrows. Firstly, the namespace URI of WSDL document that contains the type specification of *loanApplication* is got with the namespace prefix *services*. Secondly, *LoanService.wsdl* is located by matching the value of `targetNamespace` attribute with the namespace URI. Thirdly, the definition of message *LoanServiceRequestMessage* is achieved according to `messageType` of *loanApplication*. Fourthly, element *loanApplication* is located by further analysis of `part` in `message` *LoanServiceRequestMessage*. Finally, the concrete type defined in `complexType` *LoanApplicationType* is found. The related information of variable *loanApplication*

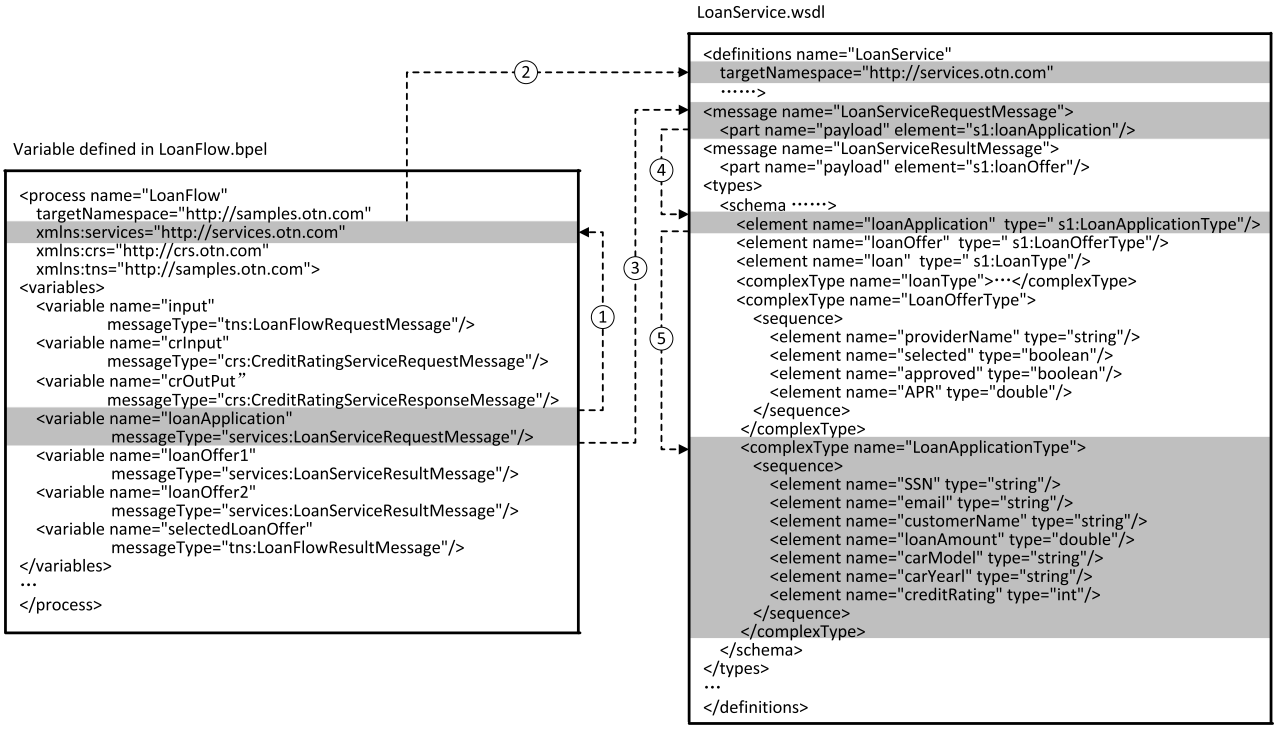


FIGURE 7. The process of finding concrete type of variable from BPEL and WSDL.

TABLE 1. Concrete type of variables in *v1.0*.

Variable	Type	WSDL	Concrete Type
input	LoanFlowRequestMessage	LoanFlow.wsdl	(string, string, string, double, string, string, int)
crInput	CreditRatingServiceRequestMessage	CreditRatingService.wsdl	string
crOutput	CreditRatingServiceResponseMessage	CreditRatingService.wsdl	int
loanApplication	LoanServiceRequestMessage	LoanService.wsdl	(string, string, string, double, string, string, int)
loanOffer1	LoanServiceResultMessage	LoanService.wsdl	(string, boolean, boolean, double)
loanOffer2	LoanServiceResultMessage	LoanService.wsdl	(string, boolean, boolean, double)
selectedLoanOffer	LoanFlowResultMessage	LoanFlow.wsdl	(string, boolean, boolean, double)

is listed in the fifth row of Table 1 which also lists the other variables and their related information in *LCS v1.0*.

The procedure of obtaining concrete type of variable is performed with the help of WSDM models of composite service and partner services, as shown in Algorithm 3. For variable v, WSDM model DM of WSDL document that contains type specification of v is selected to find its concrete type by comparing the *namespace* field of DM with the *location* filed of v. In model DM, each complexType $ct \in CT$ is checked to perfect its *types* field. For ct which has *refs* field, EM is searched for each $ref \in ct.refs$ to find corresponding $elm \in EM$ where $elm.name == ref$. And $elm.type$ is compared with each $ct' \in CT$ to check whether $elm.type$ is a complex type or not. For elm whose type is complex type, *types* field of corresponding complexType is added to the *types* field of ct. Otherwise, $elm.type$ is added to the *types* field of ct. Then message msg whose *name* field equals with the *type* field of v is identified. On the one hand, if the *elements* field of msg is not $\emptyset$, EM is searched for each $element \in msg.elements$ to find corresponding $elm \in EM$ where $elm.name == element$. If $elm.type$ is complex type, *types* field of corresponding complexType is added to the concrete type of v. Otherwise, $elm.type$ is added to the concrete type of v. On the other hand, if the *types* field of msg is not $\emptyset$, each $t \in msg.types$ is checked whether t is complex type or not. According to the result, the corresponding type is added to the concrete type of v.

VI. TEST CASE SELECTION

In this section, we will discuss how to select test cases for the affected def-use pairs. Testing paths covering the def-use pairs that require to be tested are computed. Then path conditions of testing paths between two versions are compared to select the test cases in the baseline version.

A. TESTING PATH GENERATION

Testing paths consist of XCFG paths covering def-use pairs that require to be tested for satisfying the all-uses criterion. For the def-use pairs DU_i in S_i and DU^s_{i+1} in S_{i+1}, suppose P_i and P^s_{i+1} respectively denote the XCFG paths that cover DU_i

Algorithm 3 ConcreteTypeComputation( )

```
Input v: variable in XCFG model;
Input DMSet: WSDM models in the two-level model;
Output concreteType: the concrete type of variable v;
for wsdm ∈ DMSet do
  if v.location == wsdm.namespace then
    DM = wsdm;
    break;
  end if
end for
for ct ∈ CT do
  for ref ∈ ct.refs do
    for elm ∈ EM do
      if elm.name == ref then
        isComplex = false;
        for ct' ∈ CT do
          if elm.type == ct' then
            ct.types = ct.types ∪ {ct'.types};
            isComplex = true;
            break;
          end if
        end for
        if isComplex = false then
          ct.types = ct.types ∪ {elm.type};
        end if
        break;
      end if
    end for
  end for
end for
for message ∈ M do
  if v.type == message.name then
    msg = message;
    break;
  end if
end for
for element ∈ msg.elements do
  for elm ∈ EM do
    if elm.name == element then
      isComplex = false;
      for ct ∈ CT do
        if elm.type == ct then
          concreteType = concreteType ∪ {ct.types};
          isComplex = true;
          break;
        end if
      end for
      if isComplex = false then
        concreteType = concreteType ∪ {elm.type};
      end if
      break;
    end if
  end for
end for
for t ∈ msg.types do
  isComplex = false;
  for ct ∈ CT do
    if elm.type == ct then
      concreteType = concreteType ∪ {ct.types};
      isComplex = true;
      break;
    end if
  end for
  if isComplex = false then
    concreteType = concreteType ∪ {t};
  end if
end for
```

and DU^s_{i+1}. There exists a mapping σ that satisfies:

$$P_i = \sigma(DU_i)$$
$$P^s_{i+1} = \sigma(DU^s_{i+1})$$

Obviously, we have $P^s_{i+1} \in P_{i+1}$. For the regression testing of S_{i+1}, testing paths P^s_{i+1} should be computed.

Definition 3 (XCFG path): XCFG path is a sequence of nodes to record the execution trace of BPEL process. Let p be a set composed of XCFG nodes $(n_1, n_2, \ldots, n_k)$. It is called a XCFG path if the following conditions are satisfied:

- $\forall n_i \in p, n_i \in N$.
- n_1 and n_k are respectively the start node and end node of XCFG model.
- $\forall n_i \in p(1 \leq i \leq k-1), \exists n_j \in p \mapsto (n_i, n_j) \in E$.

The steps for generating testing paths P' for def-use pairs DU are as follows:

1) Generate all XCFG paths P based on XCFG model. XCFG path is constructed from the visited nodes according to their execution orders.
2) Select the subset P' ($P' \subseteq P$) so that $\forall du \in DU$, du is covered by some path $p \in P'$.

Firstly, we will discuss how to generate all XCFG paths. Fields *inEdges* and *outEdges* of XCFG node and fields *source* and *target* of XCFG edge are used to record the execution order of XCFG nodes. With this information, depth-first traversing for XCFG can be performed to find XCFG path. The whole generation process is depicted in Algorithm 4. It starts with taking $p[1]$ as the current path and the start node *process_start* of XCFG as the current node to call Algorithm *ProcessPath*(), which recursively traverses XCFG nodes. During the traversing, different types of XCFG nodes are dealt with in different ways in Algorithm 4:

- If the current node *node* is the end node *process_end* of XCFG, it is added into $p[count]$ and the recursion of the current path $p[count]$ terminates.
- If *node* is N_{CB}, all the concurrency branches will be added into $p[count]$ accompanied with the handling of corresponding N_{CM}. Although multiple execution sequences may exist for the child nodes of the concurrency structure, they have the same execution condition and expected output for a given input. So they are treated as one single path. After adding *node* into $p[count]$, one of its subsequent nodes *to* is processed recursively. Meanwhile, to make sure all the other concurrency branches be included in $p[count]$, array $waitSet[i][j]$ is used to record the edges that outgo from N_{CB} whose *id* field is j and wait for further handling in path $p[i]$.
- If *node* is N_{CM}, we check whether all the precedent nodes are included in $p[count]$ at first. If some precedent node is not included, it means there is still some concurrency structure waiting for handling. Then an edge in $waitSet[count][start.id]$ is taken to recursively trave its target node, where *start* is the N_{CB} corresponding to *node*. *node* cannot be added into $p[count]$ until all its precedent nodes are included in $p[count]$.

Algorithm 4 ProcessPath( )

Input $p[count]$: current XCFG path to be processed;
Input *node*: current XCFG node to be processed;
Output $p[count]$: all XCFG paths;
if $node == process\_end$ **then**
  $p[count] = p[count] \cup \{node\}$;
  return;
else if $node \in N_{CB}$ **then**
  $p[count] = p[count] \cup \{node\}$;
  get the edges *outEdges* outgoing from *node* and an edge *edge* from *outEdges*:;
  $outEdges = outEdges - edge$;
  $waitSet[count][node.id] = outEdges$;
  get the target node of *edge*: *to*;
  ProcessPath(p[count],to);
else if $node \in N_{CM}$ **then**
  boolean $isfinished = true$;
  get the edges incoming to *node*: *inEdges*;
  for $edge \in inEdges$ **do**
    get the source node of *edge*: *from*;
    if $!from \in p[count]$ **then**
      $isFinished = false$;
    end if
  end for
  if $isfinished == false$ **then**
    find the start node of the concurrency structure: *start*;
    if $waitSet[count][start.id]! = \emptyset$ **then**
      get an element from $waitSet[count][start.id]$: $edge'$;
      $waitSet[count][start.id] = waitSet[count][start.id] - \{edge'\}$;
      get the target node of $edge'$: *to*;
      ProcessPath(p[count],to);
    end if
  else
    $p[count] = p[count] \cup \{node\}$;
    get the subsequent node of *node*: *to*;
    ProcessPath(p[count],to);
  end if
else if $node \in N_{ED}$ **then**
  $p[count] = p[count] \cup \{node\}$;
  $pTemp = p[count]$;
  $wTemp = waitSet[count][1..size]$;
  get the edges outgoing from *node*: *outEdges*;
  for (int $i = 0$; $i < outEdges.size()$; $i++$) **do**
    get the ith edge *edge* and the target node *to* of *edge*;
    if $i > 0$ **then**
      $count++$;
      create a new XCFG path $p[count]$;
      $p[count] = pTemp$;
      $waitSet[count][1 \ldots size] = wTemp$;
    end if
    ProcessPath(p[count],to);
  end for
else
  $p[count] = p[count] \cup \{node\}$;
  get the subsequent node of *node*: *to*;
  ProcessPath(p[count],to);
end if

- If *node* is N_{ED} with k branches, copy $p[count]$ for $k-1$ times to create another $k-1$ paths. Meanwhile *waitSet* for $p[count]$ is copied for the $k-1$ paths in case *node* is contained in some concurrency structure. Then all subsequent nodes of *node* are processed recursively in k paths.
- If *node* is of other kinds, including N_B, N_{SS}, N_{SE} and N_{EM}, *node* is added into $p[count]$ and subsequent node of *node* is processed recursively.

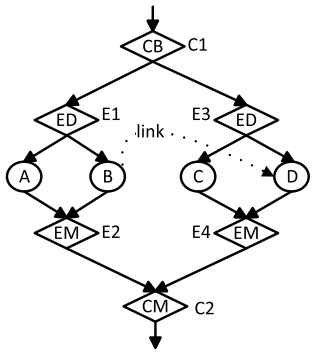


FIGURE 8. Unexecutable path caused by *link*.

However, the generation of XCFG paths in Algorithm 4 does not take the synchronization dependency defined with *link* into consideration, which may cause the unexecutable paths be included in the result. Suppose nodes n and n' are the source node and target node of *link*, if n' is contained in path p while n is not contained in p, then p is unexecutable in practice. In Fig.8, path $\{C1, E1, A, E2, E3, D, E4, C2\}$ is constructed with Algorithm 4. However, D will be in the status of waiting since its source activity B is not completed in the execution. The unexecutable paths should be get rid of from the XCFG paths P generated by Algorithm 4.

The process of deleting unexecutable paths from P is depicted in Algorithm 5. All the target nodes of *links* are identified and recorded in *targetSet*. For $\forall node \in targetSet$, its source nodes which are connected through *links* are identified and recorded in *sourceSet*. If some path p does not contain both *node* and its *sourceSet*, p is unexecutable and should be deleted from P.

Secondly, we will discuss how to select the subset P' from XCFG paths P calculated by Algorithm 5 for covering the def-use pairs DU. For a def-use pair (x, n, n'), XCFG path p covers it if the following conditions are satisfied:

- $n \in p$ and $n' \in p$.
- There is no definition of variable x in nodes $m_1 \ldots m_k$ where $(n, m_1, \ldots, m_k, n') \subseteq p$.

The key to selecting testing path p for (x, n, n') is to make sure that the nodes among $m_1, \ldots, m_k$ will not redefine x. Fig.9 shows the examples in which the redefinition of x exists. In Fig.9 (a), m_1 redefine x, which means only the path walking through m_2 can be selected. In Fig.9 (b), m_1 and m_2 redefine x and are contained in the same choice structure, which means only the path walking through m_3 can be selected. In Fig.9 (c), m_1 and m_3 redefine x and are contained in different choice structures, which means only the path walking through m_2 and m_4 can be selected. The nodes that will kill the definition (x, n) and cause (x, n) can not

Algorithm 5 RefinePath( )

Input P: all XCFG paths;
Input *xcfg*: XCFG model of BPEL process;
Output P: executable XCFG paths;
get the links in *xcfg*: *linkSet*;
for $link \in linkSet$ **do**
  get the target node of *link*: *target*;
  $targetSet = targetSet \cup \{target\}$;
end for
for $node \in targetSet$ **do**
  get the link incoming to *node*: *inLinks*;
  for $link \in inLinks$ **do**
    get the source node of *link*: *source*;
    $sourceSet = sourceSet \cup \{source\}$;
  end for
  for $p \in P$ **do**
    if $node \in p$ **then**
      if $!sourceSet \subseteq p$ **then**
        $P = P - \{p\}$;
      end if
    end if
  end for
end for
return P;

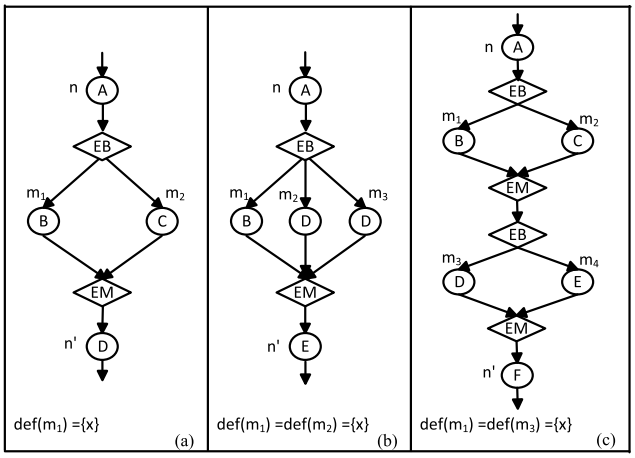


FIGURE 9. Redefinition of variable.

reach n' should be avoided in the testing path p for (x, n, n'). If some node m is executed after n and makes definition to x, meanwhile definition (x, m_i) can reach the point before n', then m should be kept away from the testing path. Algorithm 6 depicts the process of calculating the nodes that should be avoided while selecting testing path for def-use pair. With the *avoidSet*, p is the testing path for (x, n, n') only if the following condition is satisfied:

$$(p \in P) \wedge (n \in p) \wedge (n' \in p) \wedge !(avoidSet \subseteq p)$$

The process to select the subset P' from P is not very complex. We construct all the subsets of P and search these subsets in the order from small to large to check which one covers all def-use pairs to be tested, so that the minimal subset is acquired. With the above method, testing paths calculated for both *LCS v1.0* and *v1.1* are listed in Fig.6. For the affected def-use pairs (*loanOffer*1, 23, 35) and (*selectedLoanOffer*, 35, 34) in *v1.1*, only $p_{1.1}[1]$ requires to be tested.

Algorithm 6 CalcAvoidSet( )

Input (x, n, n'): def-use pair;
Output *avoidSet*: the nodes that need to be avoided in the testing path of (x, n, n');
get the reaching definitions at the point before n': *inSet*;
calculate the nodes that are executed after n: *afterSet*;
for $m \in inSet$ **do**
  if $m! = n \wedge x \in def(m)$ **then**
    if $m \in afterSet$ **then**
      $avoidSet = avoideSet \cup \{m\}$;
    end if
  end if
end for

B. PATH CONDITION ANALYSIS

The key to test case design for executing the chosen path is the satisfaction of execution condition of the path. For two paths p and p', if their path conditions are the same, meanwhile the types of test cases that drive their execution are the same, then the test cases attached to p can be used to test p'. In order to reduce unnecessary test case generation in regression testing, we perform path condition comparison to select the reusable test cases from the baseline version.

Suppose T_i is the test suite of S_i. In the testing paths P^s_{i+1}, some paths can reuse the test cases in T_i, denoted as P^{so}_{i+1}, while some need newly generated test cases, denoted as P^{sn}_{i+1}. Suppose T^{so}_{i+1} and T^{sn}_{i+1} are respectively taken for testing P^{so}_{i+1} and P^{sn}_{i+1}, we have

$$T_{i+1} = T^{so}_{i+1} \cup T^{sn}_{i+1} \quad (T^{so}_{i+1} \subseteq T_i)$$

For each XCFG path $p_{i+1}[k] \in P^s_{i+1}$, $p_{i+1}[k] \in P^{so}_{i+1}$ iff the following two conditions are satisfied:

- Compared with S_i, the type of input data in S_{i+1} stays unchanged.
- There exists a XCFG path $p_i[k] \in P_i$ so that the path conditions of $p_{i+1}[k]$ and $p_i[k]$ are the same.

With the first condition satisfied, for each XCFG path $p_{i+1}[k] \in P^{so}_{i+1}$, corresponding $p_i[k]$ whose path condition is equals with that of $p_{i+1}[k]$ can be obtained. Suppose the test suite $T_i[k]$ is attached to test $p_i[k] \in P_i$, then T^{so}_{i+1} is calculated as follows:

$$T^{so}_{i+1} = \bigcup_{\forall p_{i+1}[k] \in P^{so}_{i+1}, \exists p_i[k] \in P_i}^{p_{i+1}[k].condition == p_i[k].condition} T_i[k]$$

The path condition is a collection of predicate constraints whose value determines whether this path will be executed or not. The predicate constraints are recorded in the *condition*

Algorithm 7 SelectTc( )

Input P_i: testing paths of S_i;
Input P^s_{i+1}: testing paths of S_{i+1};
Output P^{so}_{i+1}: paths that can reuse the test cases of S_i;
Output T^{so}_{i+1}: selected test cases for P^{so}_{i+1};
compte the path conditions for each path in P_i and P^s_{i+1};
for $p_{i+1} \in P^s_{i+1}$ **do**
  for $p_i \in P_i$ **do**
    if $pac_{i+1} == pac_i$ **then**
      $p_{i+1}.reuse = p_{i+1}.reuse \cup \{p_i\}$;
      $P^{so}_{i+1} = P^{so}_{i+1} \cup \{p_{i+1}\}$;
    end if
  end for
end for
for $p_{i+1} \in P^{so}_{i+1}$ **do**
  for $p_i \in p_{i+1}.reuse$ **do**
    $p_{i+1}.tc = p_{i+1}.tc \cup \{p_i.tc\}$;
    $T^{so}_{i+1} = T^{so}_{i+1} \cup \{p_i.tc\}$;
  end for
end for

field of XCFG nodes. In the first branch nodes of choice activities, such as *if*, *while* and *repeatUntil*, the attached *condition* fields are the predicate constraints for determining whether the branch nodes will be executed. In the target nodes of link edges E_Ls, the attached *condition* fields are the predicate constraints for determining whether the target nodes will be executed. The predicate constraints can be classified into three kinds [11]: *Boolean*, *Numeric*, and *String*.

Formally, predicate constraint is defined as a triple $prc =< EP, PT, F >$, where EP is the condition expression, PT is the predicate type, F denotes how prc is combined in the path condition and $F = \{AND, OR\}$. Since predicate constraints are recorded in XCFG nodes, path condition pac for XCFG path p is defined as follows:

$$pac = \bigcup_{i=1}^{k} \{n_i.prc | n_i \in p\}$$

where n_i denotes any node in p and k denotes the number of nodes in p.

For path conditions pac_{i+1} and pac_i of XCFG paths p_{i+1} and p_i, $pac_{i+1} == pac_i$ iff the following conditions is satisfied:

$$\forall prc_{i+1} \in pac_{i+1}, \quad \forall prc_i \in pac_i,\ prc_{i+1} == prc_i$$

Furthermore, $prc_{i+1} == prc_i$ iff the following conditions is satisfied:

$$prc_{i+1}.EP == prc_i.EP \wedge prc_{i+1}.PT == prc_i.PT \wedge prc_{i+1}.F == prc_i.F$$

The process of path condition comparison based test case selection is depicted in Algorithm 7. For each path p_{i+1} in P^s_{i+1}, paths in P_i are searched to find the paths whose path conditions are the same as that of p_{i+1}, which are recorded in $p_{i+1}.reuse$. Then the test cases attached to p_i ($p_i \in p_{i+1}.resue$) are reused as the test cases for p_{i+1}, denotes as $p_{i+1}.tc$. The path conditions of testing paths in *LCS v1.0* and *v1.1* are listed in Table 5. It is obvious that the path conditions of $p_{1.1}[1]$ and $p_{1.0}[1]$ are the same. So the test cases attached to $p_{1.0}[1]$ can be reused to test $p_{1.1}[1]$ in the regression testing of *LCS v1.1*.

VII. EXPERIMENTAL EVALUATION

In this section, the proposed approach is applied to four evolved versions of *LCS* to show its effectiveness by evaluating the coverage rate of three kinds of changes. We will introduce the experimental design, the experimental data and the evaluation result.

A. EXPERIMENTAL DESIGN

To show the effectiveness of the proposed approach, the evaluation criterion proposed in [11] are taken to analyze the change coverage in the regression testing with our approach, where the detailed computation is defined here according to the proposed approach in this paper.

Suppose $\triangle^{pc}_i$, $\triangle^{bc}_i$, and $\triangle^{ic}_i$ respectively denote the numbers of actual changes in BPEL process, bingdings, and interfaces, and num^{pc}_i, num^{bc}_i, and num^{ic}_i respectively denote the numbers of three kinds of covered changes with our approach. Then the coverage rate of *process change* (denoted as ρ^{pc}_i), the coverage rate of *bingding change* (denoted as ρ^{bc}_i) and the coverage rate of *interface change* (denoted as ρ^{ic}_i) are evaluated as follows:

$$\rho^{pc}_i = \frac{num^{pc}_i}{\triangle^{pc}_i} \times 100\%$$

$$\rho^{bc}_i = \frac{num^{bc}_i}{\triangle^{bc}_i} \times 100\%$$

$$\rho^{ic}_i = \frac{num^{ic}_i}{\triangle^{ic}_i} \times 100\%$$

For *process change*, any change is reflected in XCFG of the two-level model. So ρ^{pc}_i can be calculated by analyzing the changes of XCFG elements, including XCFG nodes, edges, partnerLinks and variables. Let n_n, n_m, and n_d respectively denote the number of new XCFG elements, modified XCFG elements and deleted XCFG elements, and n'_n, n'_m, and n'_d respectively denote the number of the three covered elements, then we have

$$\rho^{pc}_i = \frac{n^{pc}_i}{\triangle^{pc}_i} \times 100\% = \frac{n'_n + n'_m + n'_d}{n_n + n_m + n_d} \times 100\%$$

For *binding change*, ρ^{bc}_i can be calculated by analyzing the changes of binding. Let b_n, b_m, and b_d respectively denote the number of new bindings, modified bindings and deleted bindings, and b'_n, b'_m, and b'_d respectively denote the number of the three covered bindings, then we have

$$\rho^{bc}_i = \frac{n^{bc}_i}{\triangle^{bc}_i} \times 100\% = \frac{b'_n + b'_m + b'_d}{b_n + b_m + b_d} \times 100\%$$

```
……
(10)   <assign name="copySSN" />
         …
         <copy>
           <from>
             <literal><service-ref><EndpointReference>
               <Address>http://www.creditRS.com/</Address>
               <ServiceName>invoiseService</ServiceName>
             </EndpointReference></service-ref></literal>
           </from>
           <to partnerLink="creditRatingService"/>
         </copy>
       </assign>
……
```

(a)Modified part in **LoanFlow.bpel** for *v1.2*

```
<definition name="LoanService"... />
  ……
  <types>
    …
    <element name="loanApplication" type="S1:LoanApplicationType"/>
    <complexType name="LoanApplicationType">
      <sequence>
        …
        <element name="carYear" type="string"/>
        <element name="creditRating" type="double"/>
      </sequence>
    </complexType>
  ……
</definition>
```

(b)Modified part in **LoanFlow.wsdl** and **LoanService.wsdl** for *v1.3*

```
      <partnerLink name="customerService"⋯/>
      <partnerLink name="TaskService"⋯/>
……
(5)   <receive name="receiveInput" partnerLink="client"  portType="Tns:LoanFlow operation="initiate"  variable="input" createInstance="yes" />
(35)  <scope name="getCustomerSSN" ...>
        <variables>⋯</variables>
(37)    <sequence>
(39)      <assign name="assignRequest">
            <copy><from variable="input"⋯/>
            <to variable="getCustomerSSNRequest"⋯/></copy></assign>
(40)      <invoke name="getCustomerSSN" partnerLink="customerService" portType="cs:CustomerService" operation="getCustomerSSN"
                  inputVariable="getCustomerSSNRequest" outputVariable="getCustomerSSNResponse"/>
(41)      <assign name="assignResponse">
            <copy><from variable="getCustomerSSNResponse"⋯/>
            <to variable="input"⋯/></copy></assign>
(38)    </sequence>
(36)  </scope>
      <scope name="getCreditRating">⋯</scope>
      <scope name="GetLoanOffer">⋯</scope>
(42)  <scope name="LoanOfferReview"⋯>
        <variables>⋯</variables>
(44)    <sequence>
(46)      <assign name="LoanOfferReview_AssignTaskAttributes">⋯</assign>
(47)      <assign name="LoanOfferReview_AssignSystemTaskAttributes">⋯</assign>
(48)      <invoke name="initiateTask" parterLink="TaskService" portType="Taskservice:TaskService" operation="initiateTask"
                  inputVariable="initiateTaskInput" outputVariable="initiateTaskResponseMessage">
(49)      <receive name="receiveCompletedTask partnerLink="TaskService" portType="Taskservice:TaskServiceCallback"
                  operation="onTaskCompleted" variable="LoanOfferReview_globalVariable createInstance="no"/>
(45)    </sequence>
(43)  </scope>
(50)  <if name="taskSwitch">
        <condition>
          "Bpws:getVariableData('LoanOfferReview_globalVariable','payload','/task:task/task:systemAttibutes/task:state')='COMPLETED'" and
          "Bpws:getVariableData('LoanOfferReview_globalVariable','payload','/task:task/task:systemAttibutes/task:outcome')='ACKNOWLEDGE'"
        </condition>
(52)    <sequence>
(54)      <assign>
            <copy><from variable="LoanOfferReview_globalVariable⋯/>
            <to variable="selectedLoanOffer"⋯/></copy></assign>
(53)    </sequence>
        <else>
(55)      <sequence>
(57)        <assign>
              <copy><from variable="LoanOfferReview_globalVariable⋯/>
              <to variable="selectedLoanOffer"⋯/></copy> </assign>
(56)      </sequence>
        </else>
(51)  </if>
(34)  <invoke name="replyOutput" ... operation="onResult"  inputVariable="selectedLoanOffer"/>
```

(c)Modified part in **LoanFlow.bpel** for *v2.0*

FIGURE 10. **Modifications in the three evolved versions of *LCS*.**

For *interface change*, changes of `type` and `messages` are reflected in WSDM of the two-level model. Although `operation`, `portType`, `binding`, `port`, and `service` are not reflected in WSDM, the changed elements that are used by the composite service can be covered in the regression testing based on our approach. Let T_d, M_d, O_d, PT_d, B_d, P_d, and S_d respectively denote the number of changed `type`, `messages`, `operation`, `portType`, `binding`, `port`, and `service` in WSDL documents of both composite service and partner services, T_u, M_u, O_u, PT_u, B_u, P_u, and S_u respectively denote the number of covered changes of each element, then we have

$$\rho_i^{ic} = \frac{n_i^{ic}}{\triangle_i^{ic}} \times 100\% = \frac{T_u + M_u + O_u + PT_u + B_u + P_u + S_u}{T_d + M_d + O_d + PT_d + B_d + P_d + S_d} \times 100\%$$

BPEL composite service *LCS v1.0* and its four evolved versions *v1.1*, *v1.2*, *v1.3*, and *v2.0* are taken as the experimental objects, where *v1.0* is set as the baseline version of the four evolved versions in the experimental evaluation. The modifications in *v1.1* have been shown in Fig.1, and the modifications of *v1.2*, *v1.3*, and *v2.0* are shown in Fig.10. In *v1.2*,

the partner service *CreditRatingService* is replaced with a new one. In *v1.3*, the content of `complexType` *LoanApplicationType* is changed in *LoanFlow.wsdl* and *LoanService.wsdl*. In *v2.0*, partner services *customerService* and *TaskService* are imported to provide SSN(social security number) querying and loan offer checking in the process, and corresponding WSDL documents are imported.

B. EXPERIMENTAL DATA

We have developed a prototype tool XCFG4BPEL for implementing the automatic test case selection of BPEL composite service. For test case selection for *LCS v1.1*, *v1.2*, *v1.3* and *v2.0*, model construction, data flow analysis, change impact analysis, testing path computation and path condition computation are required.

According to the method presented in Section 4.2, we can get the two-level models of all five versions, where the XCFG models are shown in Fig.11. The modified elements caused by *process change*s in *v1.1* and *v2.0* are marked in bold. Although *binding change* in *v1.2* and *interface change* in *v1.3* can not be directly reflected in XCFG model, the XCFG nodes directly related with the change are marked in grey.

With the data flow analysis presented in Section 4.3, the def-use pairs of *LCS* for all versions are shown in Table 2, where the second and third columns respectively list the def-use pairs of *Computation* type and *Predicate* type. Then change impact analysis is performed to *LCS v1.1*, *v1.2*, *v1.3* and *v2.0*. Table 3 shows the concrete types of variables obtained by importing WSDM models with Algorithm 3 for type pairs identification in the change impact analysis. In Table 3, the bold parts indicate the modified types compared with that of the baseline version, and the concrete types of five newly added variables in *v2.0* are ignored since concrete type comparison with the baseline version is not required for them. The def-use pairs affected by *process change*, *binding change* and *interface change* that need to be tested in the regression testing of each evolved version are identified according to the steps illustrated in Section 5, which are marked as bold def-use pairs in Table 2.

Let P_v denote the set of testing paths of version v and $P_v = \{p_v[1], p_v[2], \ldots, p_v[k]\}(k \geq 1)$, where $p_v[k]$ is the kth path in P_v. The testing paths covering all def-use pairs that require to be tested can be computed with the method illustrated in Section 6.1, which are shown in Table 4. The path condition for each testing path $p_v[k]$ is denoted as $pac_v[k]$. Path condition $pac_v[k]$ is consisted of predicate constraints and $pac_v[k] = \{prc_v[k][1], prc_v[k][2], \ldots, prc_v[k][j]\}(j \geq 1)$, where $prc_v[k][j]$ is the jth predicate constraint in $pac_v[k]$. The path condition of each testing path for all the five versions is provided in Table 5.

Suppose the test suite for *v1.0* be $T_{1.0} = \{t_1, t_2, t_3, t_4, t_5, t_6\}$, where t_1, t_2, t_3 are attached to test $p_{1.0}[1]$, and t_4, t_5, t_6 are attached to test $p_{1.0}[2]$. Test cases selected for each testing path of the evolved versions are shown in Table 6, where the second column represents the XCFG paths that need to be tested in the regression testing, the third column represents the test cases that can be selected from the baseline version *v1.0* to test the testing path and the fourth column represents whether the testing path needs newly generated test cases. In *LCS v1.1* and *v1.2*, the path condition of the testing path is equals with that of $p_{1.0}[1]$, so the test cases attached to $p_{1.0}[1]$ can be selected as the test cases in the regression testing of the two evolved versions. In *v1.3*, the concrete type of the input variable *input* is changed, and new test cases need to be generated for $p_{1.3}[1]$. In *v2.0*, new test cases are required since the path conditions of the two testing paths are changed.

C. EVALUATION RESULT

In this section, the coverage rate of *process change*, *binding change* and *interface change* will be evaluated for the four evolved versions to show the effectiveness of our approach.

The changes of the four versions compared with the baseline version can be partly reflected in the two-level model of composite service. The statistics of the comparison based on the two-level model are provided in Table 7. The second and third columns respectively show the number of graphical XCFG elements, including XCFG nodes and XCFG edges, and the number of changed graphical XCFG elements. The fourth and fifth columns respectively show the number of partnerLinks in XCFG model and the number of changed partnerLinks. The sixth and seventh columns respectively show the number of variables in XCFG model and the number of changed variables. The eighth and ninth columns respectively show the number of messages in WSDM model and the number of changed messages. The tenth and eleventh columns respectively show the number of elements in WSDM model and the number of changed elements. The following two columns respectively show the number of complexTypes in WSDM model and the number of changed complexTypes. The number of def-use pairs is provided in the fourteenth column, while the number of affected def-use pairs that need to be tested in the regression testing is provided in the fifteenth column. The number of testing paths is shown in the sixteenth column. And the change types are shown in the last column. For example, *process change* happens during the evolution from *v1.0* to *v1.1*. Both *v1.0* and *v1.1* have 69 XCFG nodes and edges and *v1.1* has 3 changed graphical XCFG elements against *v1.0*. The other elements in the two-level model, including partnerLinks, variables, messages, elements and complexTypes, have no changes. In *v1.1*, only 2 def-use pairs are affected by the *process change* and 1 testing path is generated to cover the two def-use pairs.

In *v1.1*, test cases t_1, t_2, t_3 are reused to test $p_{1.1}[1]$ since the path conditions of $p_{1.1}[1]$ and $p_{1.0}[1]$ are the same. One activity is changed in *v1.1* against *v1.0*, which causes 1 XCFG nodes and 2 XCFG edges be changed in XCFG model. The change coverage of *process change* in *v1.1* is $\rho^{pc}_{1.1} = 3/3 = 100\%$, which is also shown in the first row of Table 8.

In *v1.2*, test cases t_1, t_2, t_3 are reused to test $p_{1.2}[1]$ since the path conditions of $p_{1.2}[1]$ and $p_{1.0}[1]$ are the same. One binding is changed in *v1.2* against *v1.0*. The change coverage of *binding change* in *v1.2* is $\rho^{bc}_{1.2} = 1/1 = 100\%$.

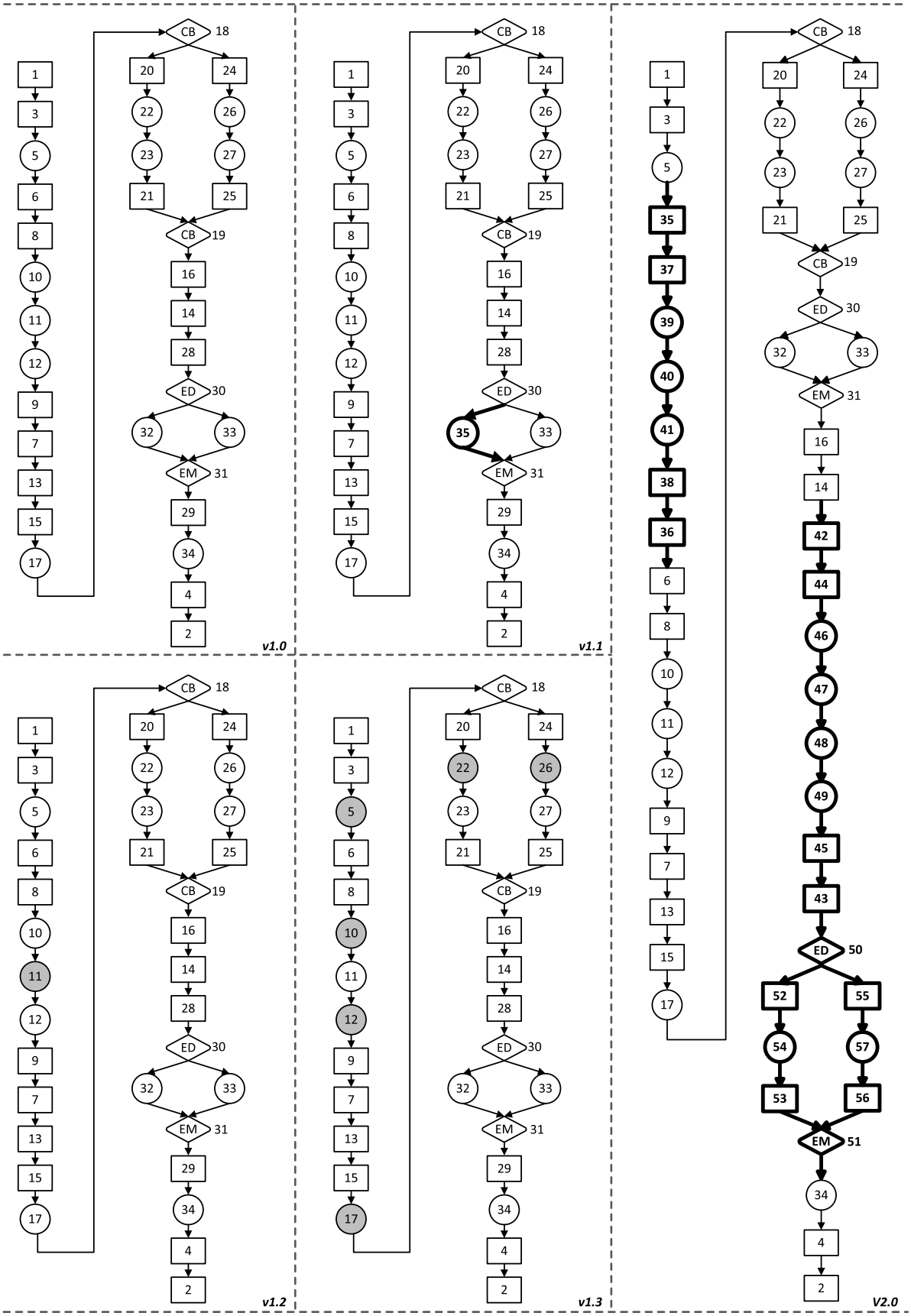


FIGURE 11. The XCFGs of *LCS* for the five versions.

TABLE 2. Def-use pairs of *LCS* for the five versions.

Version	Computation	Predicate
$v1.0$	$(input, 5, 10), (input, 12, 17), (crInput, 10, 11), (crOutput, 11, 12),$ $(loanApplication, 17, 22), (loanApplication, 17, 26), (loanOffer1, 23, 33),$ $(loanOffer2, 27, 32), (selectedLoanOffer, 32, 34), (selectedLoanOffer, 33, 34)$	$(loanOffer1, 23, 32), (loanOffer1, 23, 33),$ $(loanOffer2, 27, 32), (loanOffer2, 27, 33)$
$v1.1$	$(input, 5, 10), (input, 12, 17), (crInput, 10, 11), (crOutput, 11, 12),$ $(loanApplication, 17, 22), (loanApplication, 17, 26), (loanOffer1, 23, 33),$ **(loanOffer1,23,35),(selectedLoanOffer,35,34)**,$(selectedLoanOffer, 33, 34)$	$(loanOffer1, 23, 35), (loanOffer1, 23, 33),$ $(loanOffer2, 27, 35), (loanOffer2, 27, 33)$
$v1.2$	$(input, 5, 10)$,**(input,12,17)**,$(crInput, 10, 11)$,**(crOutput,11,12),** **(loanApplication,17,22)**,**(loanApplication,17,26)**,$(loanOffer1, 23, 33),$ $(loanOffer2, 27, 32), (selectedLoanOffer, 32, 34), (selectedLoanOffer, 33, 34)$	$(loanOffer1, 23, 32), (loanOffer1, 23, 33),$ $(loanOffer2, 27, 32), (loanOffer2, 27, 33)$
$v1.3$	**(input,5,10)**,**(input,12,17)**,$(crInput, 10, 11), (crOutput, 11, 12),$ **(loanApplication,17,22)**,**(loanApplication,17,26)**,$(loanOffer1, 23, 33),$ $(loanOffer2, 27, 32), (selectedLoanOffer, 32, 34), (selectedLoanOffer, 33, 34)$	$(loanOffer1, 23, 32), (loanOffer1, 23, 33),$ $(loanOffer2, 27, 32), (loanOffer2, 27, 33)$
$v2.0$	**(input,5,39)**,**(input,41,10)**,**(input,12,17)**, **(getCustomerSSNRequest,39,40)**,**(getCustomerSSNResponse,40,41)**, **(crInput,10,11)**,**(crOutput,11,12)**,**(loanApplication,17,22)**, **(loanApplication,17,26)**,$(loanOffer1, 23, 33), (loanOffer2, 27, 32),$ **(selectedLoanOffer,32,46)**,**(selectedLoanOffer,33,46)**, **(selectedLoanOffer,54,34)**, **(selectedLoanOffer,57,34)**, **(initiateTaskInput,47,48)**,**(LoanOfferReview_globalVariable,49,54)**, **(LoanOfferReview_globalVariable,49,57)**	$(loanOffer1, 23, 32), (loanOffer1, 23, 33),$ $(loanOffer2, 27, 32), (loanOffer2, 27, 33),$ **(LoanOfferReview_globalVariable,49,52)**, **(LoanOfferReview_globalVariable,49,55)**

TABLE 3. Types of variables in the five versions.

Variable	*v1.0*	*v1.1*	*v1.2*	*v1.3*	*v2.0*
input	(string, string, string, double, string, string, int)	(string, string, string, double, string, string, int)	(string, string, string, double, string, string, int)	(string, string, string, double, string, string, **double**)	(string, string, string, double, string, string, int)
crInput	string	string	string	string	string
crOutput	int	int	int	int	int
loanApplication	(string, string, string, double, string, string, int)	(string, string, string, double, string, string, int)	(string, string, string, double, string, string, int)	(string, string, string, double, string, string, **double**)	(string, string, string, double, string, string, int)
loanOffer1	(string, boolean, boolean, double)	(string, boolean, boolean, double)	(string, boolean, boolean, double)	(string, boolean, boolean, double)	(string, boolean, boolean, double)
loanOffer2	(string, boolean, boolean, double)	(string, boolean, boolean, double)	(string, boolean, boolean, double)	(string, boolean, boolean, double)	(string, boolean, boolean, double)
selectedLoanOffer	(string, boolean, boolean, double)	(string, boolean, boolean, double)	(string, boolean, boolean, double)	(string, boolean, boolean, double)	(string, boolean, boolean, double)
getCustomerSSNRequest	-	-	-	-	**GetCustomerSSNRequestMessage**
getCustomerSSNResponse	-	-	-	-	**GetCustomerSSNResponseMessage**
initiateTaskInput	-	-	-	-	**initiateTaskMessage**
initiateTaskResponseMessage	-	-	-	-	**initiateTaskResponseMessage**
LoanOfferReview _globalVariable	-	-	-	-	**taskMessage**

TABLE 4. Testing paths of *LCS* for the five versions.

Ref.	Testing path
$p_{1.0}[1]$	$\{1, 3, 5, 6, 8, 10, 11, 12, 9, 7, 13, 15, 17, 18, 20, 22, 23, 21, 24, 26, 27, 25, 19, 16, 14, 28, 30, 32, 31, 29, 34, 4, 2\}$
$p_{1.0}[2]$	$\{1, 3, 5, 6, 8, 10, 11, 12, 9, 7, 13, 15, 17, 18, 20, 22, 23, 21, 24, 26, 27, 25, 19, 16, 14, 28, 30, 33, 31, 29, 34, 4, 2\}$
$p_{1.1}[1]$	$\{1, 3, 5, 6, 8, 10, 11, 12, 9, 7, 13, 15, 17, 18, 20, 22, 23, 21, 24, 26, 27, 25, 19, 16, 14, 28, 30, 35, 31, 29, 34, 4, 2\}$
$p_{1.2}[1]$	$\{1, 3, 5, 6, 8, 10, 11, 12, 9, 7, 13, 15, 17, 18, 20, 22, 23, 21, 24, 26, 27, 25, 19, 16, 14, 28, 30, 32, 31, 29, 34, 4, 2\}$
$p_{1.3}[1]$	$\{1, 3, 5, 6, 8, 10, 11, 12, 9, 7, 13, 15, 17, 18, 20, 22, 23, 21, 24, 26, 27, 25, 19, 16, 14, 28, 30, 32, 31, 29, 34, 4, 2\}$
$p_{2.0}[1]$	$\{1, 3, 5, 35, 37, 39, 40, 41, 38, 36, 6, 8, 10, 11, 12, 9, 7, 13, 15, 17, 18, 20, 22, 23, 21, 24, 26, 27, 25, 19, 30, 32, 31, 16, 14, 42, 44, 46, 47, 48, 49, 45, 43, 50, 52, 54, 53, 51, 34, 4, 2\}$
$p_{2.0}[2]$	$\{1, 3, 5, 35, 37, 39, 40, 41, 38, 36, 6, 8, 10, 11, 12, 9, 7, 13, 15, 17, 18, 20, 22, 23, 21, 24, 26, 27, 25, 19, 30, 33, 31, 16, 14, 42, 44, 46, 47, 48, 49, 45, 43, 50, 55, 57, 56, 51, 34, 4, 2\}$

In *v1.3*, new test cases must be generated to test $p_{1.3}[1]$ since the data type of the input is changed. As shown in Table 7, 2 messages, 3 elements and 3 complexTypes are changed in related WSDLs, which causes 2 operations and 2 portTypes be changed in WSDLs. Four affected def-use pairs of 2 variables whose concrete types are changed are retested, and 2 changed complexTypes, 2 changed elements and 2 changed messages are covered. The testing of $p_{1.3}[1]$ also covers 2 changed operations and 2 changed portTypes. So the change coverage of *interface change* in *v1.3* is $\rho^{ic}_{1.3} = 10/12 = 83.33\%$.

In *v2.0*, 2 XCFG paths are tested for 18 affected def-use pairs because of all the three kinds of changes and they need new test cases. The coverage of different change types are provided as follows:

- *Process change*: 49 XCFG nodes and edges are changed, 2 partnerLinks are added and 5 variables are added in XCFG model corresponding to the *process changes* in *v2.0*. All the 56 changed elements are covered in the testing of 2 paths, so the change coverage of *process change* in *v2.0* is $\rho^{pc}_{2.0} = 100\%$.

TABLE 5. Path conditions of testing paths for the five versions.

Ref.	Path condition			
	Predicate constraint	EP	PT	F
$pac_{1.0}[1]$	$prc_{1.0}[1]$	$getVariableData('loanOffer1') > getVariableData('loanOffer2')$	Numeric	AND
$pac_{1.0}[2]$	$prc_{1.0}[2]$	$getVariableData('loanOffer1') \leq getVariableData('loanOffer2')$	Numeric	AND
$pac_{1.1}[1]$	$prc_{1.1}[1]$	$getVariableData('loanOffer1') > getVariableData('loanOffer2')$	Numeric	AND
$pac_{1.2}[1]$	$prc_{1.2}[1]$	$getVariableData('loanOffer1') > getVariableData('loanOffer2')$	Numeric	AND
$pac_{1.3}[1]$	$prc_{1.3}[1]$	$getVariableData('loanOffer1') > getVariableData('loanOffer2')$	Numeric	AND
$pac_{2.0}[1]$	$prc_{2.0}[1][1]$	$getVariableData('loanOffer1') > getVariableData('loanOffer2')$	Numeric	AND
	$prc_{2.0}[1][2]$	$getVariableData('LoanOfferReview\_globalVariable') =' COMPLETE'$	String	AND
	$prc_{2.0}[1][3]$	$getVariableData('LoanOfferReview\_globalVariable') =' ACKNOWLEDGE'$	String	AND
$pac_{2.0}[2]$	$prc_{1.0}[2][1]$	$getVariableData('loanOffer1') \leq getVariableData('loanOffer2')$	Numeric	AND
	$prc_{2.0}[2][2]$	$getVariableData('LoanOfferReview\_globalVariable') \neq' COMPLETE'$	String	OR
	$prc_{2.0}[2][3]$	$getVariableData('LoanOfferReview\_globalVariable') \neq' ACKNOWLEDGE'$	String	OR

TABLE 6. Test case selection for four evolved versions.

Version	Testing path	Reuse test cases	New test cases
$v1.1$	$p_{1.1}[1]$	t_1, t_2, t_3	
$v1.2$	$p_{1.2}[1]$	t_1, t_2, t_3	
$v1.3$	$p_{1.3}[1]$	-	$\surd$
$v2.0$	$p_{2.0}[1]$	-	$\surd$
	$p_{2.0}[2]$	-	$\surd$

- *Binding change*: 2 bindings are added and both of them are covered with our approach, so the change coverage of *binding change* in *v2.0* is $\rho^{bc}_{2.0} = 100\%$.
- *Interface change*: 3 WSDL documents are added in *v2.0*, including *CustomerService.wsdl*, *TaskServiceWSIF.wsdl* and *TaskServiceInterface.wsdl*, in which *TaskServiceInterface.wsdl* is imported in *TaskServiceWSIF.wsdl* to constitute the service definition of *TaskService*. In the new WSDLs, 50 messages and 2 elements are added, as shown in Table 7. In addition, 40 operations, 3 portTypes, 3 bindings, 3 ports and 3 services are added. In total 104 WSDL elements are changed, which is different with the statistical data in the preliminary version [8] since the imported *.xsd* files are excluded here. The testing of $p_{2.0}[1]$ and $p_{2.0}[2]$ can cover 22 of the changed elements, where $T_u = 2$, $M_u = 5$, $O_u = 3$, $PT_u = 3$, $B_u = 3$, $P_u = 3$ and $S_u = 3$. So the the change coverage of *interface change* in *v2.0* is $\rho^{ic}_{2.0} = 22/104 = 21.15\%$.

The evaluation result shows that the coverage of *interface change* is much lower than *process change* and *binding change*. The main reason is that only composite service related WSDL elements can be covered in the regression testing, while most elements in WSDL of *TaskService* are irrelative to BPEL composite service in *v2.0*. Since deleting the irrelative elements of WSDL won't influence the execution of BPEL process, we perform three experiments to consider the possibility of improving the coverage of *interface change* by changing the newly added WSDLs. In the first experiment, we delete 5 unused operations and 5 related messages in *TaskServiceInterface.wsdl*. The result is that the same 22 changed WSDL elements are covered, $\rho^{ic}_{2.0} = 22/94 = 23.4\%$. In the second experiment, another 4 unused operations and 7 related messages are deleted in *TaskServiceInterface.wsdl*. Still, 22 changed WSDL elements

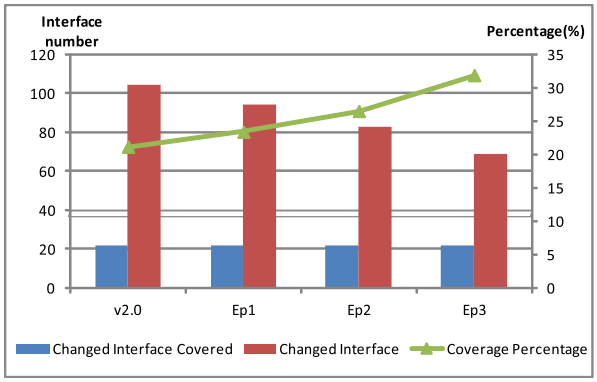


FIGURE 12. The coverage rate of interface change in *v2.0*.

are covered, and $\rho^{ic}_{2.0} = 22/83 = 26.51\%$. In the third experiment, another 7 unused operations and 7 related messages are deleted in *TaskServiceInterface.wsdl*. Still, 22 changed WSDL elements are covered, and $\rho^{ic}_{2.0} = 22/69 = 31.88\%$. Fig.12 shows that the coverage of *interface change* increases with the reduction of irrelevant elements in WSDLs of new partner services. The less irrelevant elements exist, the higher interface change coverage is.

VIII. RELATED WORK

Regression testing of composite service has been studied in many works, including three major branches: test case selection, test case prioritization and test suite minimization (reduction). Some studies worked on how to schedule the test cases to increase the effectiveness of regression testing [18], [19], and how to obtain a minimal subset of test suite [20]. Some studies worked on how to select test cases for basic services [21], [22]. In this section, we mainly discuss the related work about test case selection in the regression testing of composite service, as shown in Table 9.

Farooq *et al.* [3] presented a model-based regression test selection approach to provide the impact analysis earlier and early assessment of test effort. The model of business process consisted of BPMN (Business Process Modeling Notation) from the process view, UML (Unified Modeling Language) from structural view and UTP (UML Testing Profile) from test view. With dependency relations between models recorded, a set of impact rules were developed to forecast the

TABLE 7. Two-level model based statistics of the five versions.

Version	$N\&E$	$\triangle N\&E$	PL	$\triangle PL$	V	$\triangle V$	M	$\triangle M$	EM	$\triangle EM$	CT	$\triangle CT$	DU	$\triangle DU$	P	change type
v1.0	69	-	4	-	7	-	7	-	8	-	5	-	14	-	-	-
v1.1	69	3	4	0	7	0	7	0	8	0	5	0	14	2	1	*Process change*
v1.2	69	0	4	1	7	0	7	0	8	0	5	0	14	4	1	*Binding change*
v1.3	69	0	4	0	7	2	7	2	8	3	5	3	14	4	1	*Interface change*
v2.0	112	49	6	2	12	5	57	50	10	2	5	0	24	18	2	*Process change* *Binding change* *Interface change*

TABLE 8. Change coverage of four evolved versions.

Version	ρ_i^{pc}	ρ_i^{bc}	ρ_i^{ic}
$v1.1$	$3/3 = 100\%$	-	-
$v1.2$	-	$1/1 = 100\%$	-
$v1.3$	-	-	$10/12 = 83.33\%$
$v2.0$	$56/56 = 100\%$	$2/2 = 100\%$	$22/104 = 21.15\%$

impacts of changes and the resulting change propagation on test suite. And tests were analyzed and classified to determine which test cases were required for regression testing.

Some studies worked on the specification-based regression test selection. Khan and Heckel [4] proposed a TGTS (typed graph transformation systems) based approach at the level of interfaces. In TGTS, a set of rules with pre- and post conditions described the semantics of operations as visual contracts. The impact of changes of the signature, contract, or implementation of an operation was assessed by the analysis of conflicts and dependencies of the rules, which was used to select test cases for rerun. Sahoo and Ray [23] generated operation trees for both original and modified WSDL files so that change detection algorithm was used to identify the changes. And dynamic forward slicing algorithm was applied to the modified operation tree to find out the affected parts. Only the test cases that cover the modified and affected parts were selected. The two approaches could also be applied to basic services.

Most of the studies focused on the code-based regression test selection. Wang *et al.* [1] presented an all-activities coverage criterion oriented safe regression test selection approach guided by the behavior difference of activities. BPEL applications and their modified versions were transformed into universal BPEL forms with three rules respectively corresponding to DPE (dead path elimination), communication mechanism and multi-assignment. Then BPEL program dependency graphs were established to identify all affected activities and select corresponding tests by backward program slicing which computed the behavior of one activity. The approach could eliminate some unnecessary test cases to be selected from semantic perspective rather than from syntactic perspective.

Liu *et al.* [24] took the changes of concurrent control structures into consideration in the BFG (BPEL Flow Graph) based regression testing of BPEL composite service. An impact analysis rule for concurrent control structures was proposed to perform impact analysis. Based on the process change information and impact analysis result, the impacted test paths were selected and classified into modified, obsolete and new-structural paths. The approach could be applied even if the test paths were not generated. However, it had the limitation that it could not identify the new process structure. Li *et al.* [25] overcame the limitation based on the direct comparison of the two test case sets that were generated with test-path exploration for both the old and new BPEL processes.

Ruth *et al.* [26] applied a safe regression test selection technique to Java-based web services. The approach modeled Java-based web service with the Java-based control flow graph JIG (Java Interclass Graph), in which a simulation tool was used to transform the Java web services code into local Java programs to form a global JIG. The comparison between new JIG and old JIG was performed to recognize a set of dangerous edges so that which tests needed to be rerun were determined. The approach took into account the changes and effects of the back-end components.

Ruth *et al.* [27] required CFGs of participating services and applications rather than requiring a complete view of all the source code to construct the global CFG. Dual-traversal of modified CFG and original CFG was performed to identify dangerous edges, in which a number of issues due to concurrent changes were recognized and solved. The relationship of concurrent changes could be identified with the call graph and ''downstream changes first'' rule was followed to order the test case selection. Considering the situation that service providers were unwilling to expose CFGs, Ruth [28] employed privacy-preserving techniques (PPT) to protect the sensitive information contained within CFGs. The contents of individual nodes in CFG were sanitized with cryptographic hash functions. The shape of CFG was protected by CFG flattening. In addition, service providers may optionally contract or expand parts of CFGs prior to the flattening process. And the coverage information was updated accompanied by altering the shape of CFG. Ruth and Rayford [29] further presented a privacy-aware technique (PAT) using only locally available information of each service in which partial CFGs with ''call'' nodes were generated for composite service. A series of empirical studies [30] were undertaken which showed that PPT was more cost-effective than PAT.

Tarhini *et al.* [31] selected an adequate number of non-redundant test sequences aiming to find modification-related errors. A TLTS (Timed Labeled Transition System) based two-level model was proposed to represent the interaction of components with web application in the first level and the behavior of its composed components in the second level. With test set generated by traversing all acyclic

TABLE 9. Comparison of related work on regression test selection of composite service.

Reference	WS technique	Model	Test strategy	Test Coverage	Change types included
Farooq et al. [3]	BPMN	BPMN,UML,UTP	White-box	Modification coverage	Process change interface change
Khan and Heckel [4]	-	TGTS	Black-box	-	Interface change
Sahoo and Ray [23]	WSDL	Operation tree	Black-box	-	Interface change
Wang et al. [1]	BPEL	Dependency graph	White-box	Activity coverage	Process change
Liu et al. [24]	BPEL	BFG	White-box	Path coverage	Process change
Ruth et al. [26]	WSDL	JIG	White-box	Path coverage	Implementation change
Ruth et al. [27-29]	WSDL	Global CFG	White-box	Path coverage	Process change Implementation change
Tarhini et al. [31]	WSDL	TLTS	White-box	Modification coverage	Process change Binding change Implementation change
Li et al. [11]	BPEL,WSDL	XBFG	White-box	Path coverage	Process change Binding change Interface change
Our approach	BPEL,WSDL	XCFG,WSDM	White-box	Def-use pair coverage	Process change Binding change Interface change

paths of TLTS, the test selection algorithm compared the test sets of two versions to identify all modifications and select modification-related test cases, in which three kinds of modifications were discussed.

Li *et al.* [11] proposed an XBFG (eXtensible BPEL Flow Graph) based test case selection approach. Message sequence was appended to XBFG path in order to fully describe the behavior of composite service. XBFG path comparison was performed to find the XBFG paths influenced by process change and binding change. And message sequence comparison was performed to find the XBFG paths influenced by interface change. Then paths identified to be retested were divided into two parts after path condition analysis to distinguish which could reuse test cases while which required new test cases.

The code-based approaches [1], [11], [24]–[31] performed white-box testing, although some did not specify the web service technique for composition as shown in the second column in Table 9. Regarding the models used in the code based regression test selection approaches, dependency graph modeled the control dependence, data dependence, and aysn-invocation dependence of activities in the BPEL process [1]. TLTS modeled the interaction of components with the main application in the first level and modeled the state transition of each participant component to depict the internal behavior [31]. The others intuitively modeled the behavior of composite service with CFG and CFG-based extension [11], [24]–[30]. In addition, JIG [26] and Global CFG [27], [28] also contained the behavior of component services, and XBFG also modeled the interactions between the process and its partner services. The two-level model proposed in this paper not only models the control flow and data flow of BPEL process with XCFG but also models the interface of services with WSDM. Regarding the change types included in the regression testing, it requires to identify and handle as many changes as possible. Some works [26]–[31] took *Implementation change* into consideration, which relied on the collaboration of service providers. From the perspective of service integrator, only *process change*, *binding change*, and *interface change* are observable. Only the approaches provided by Li *et al.* [11] and proposed in this paper considered the three kinds of changes. Regarding the test coverage criterion, our approach is the only one that selects test cases based on the data flow testing criterion. Our work is meaningful since ensuring the data flow correctness is very important.

IX. CONCLUSION

Regression testing plays an important role to ensure the correctness of evolved composite services. In this paper, we proposed an all-use criterion based test case selection approach for BPEL composite service. The approach identifies and handles *process change*, *binding change* and *interface change* using a two-level model which consists of XCFG and WSDM to describe the composite process and the interfaces of participating services. The def-use pairs affected by the three kinds of changes which include *new pairs*, *value pairs*, *condition pairs*, and *type pairs* are detected by two-level model comparison based change impact analysis. Test cases are selected to run the affected def-use pairs of the evolved version by testing path generation and path condition analysis. We show the effectiveness of our approach through empirical study. This work is meaningful in the data flow criteria oriented regression testing of composite service.

Our research represents an initial work on the regression testing of composite services. There are a lot of interesting problems to be studied in future work, including: (1) Improve our approach to take the unique features of BPEL (such as dead path elimination) into consideration; (2) Enhance our tool to apply our approach to large-scale composite services; (3) Study how to perform test case selection on WS-CDL or OWL-S based composite services.

REFERENCES

[1] H. Wang, J. Xing, Q. Yang, P. Wang, X. Zhang, and D. Han, "Optimal control based regression test selection for service-oriented workflow applications," *J. Syst. Softw.*, vol. 124, pp. 274–288, Feb. 2017.

[2] W. Song and H.-A. Jacobsen, "Static and dynamic process change," *IEEE Trans. Serv. Comput.*, vol. 11, no. 1, pp. 215–231, Jan./Feb. 2016.

[3] Q.-U.-A. Farooq, S. Lehnert and M. Riebisch, "Analyzing model dependencies for rule-based regression test selection," in *Proc. Modellierung*, 2014, pp. 305–320.
[4] T. A. Khan and R. Heckel, "On model-based regression testing of Web-services using dependency analysis of visual contracts," in *Proc. 14th Int. Conf. Fundam. Approaches Softw. Eng., Joint Eur. Conf. Theory Pract. Softw.*, 2011, pp. 341–355.
[5] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE Trans. Softw. Eng.*, vol. SE-11, no. 4, pp. 367–375, Apr. 1985.
[6] L. Mei, W. K. Chan, T. H. Tse, and F.-C. Kuo, "An empirical study of the use of Frankl-Weyuker data flow testing criteria to test BPEL Web services," in *Proc. 33th Annu. IEEE Int. Conf. Comput. Softw. Appl.*, Jul. 2009, pp. 81–88.
[7] D. Qiu, B. Li, S. Ji, and H. Leung, "Regression testing of Web service: A systematic mapping study," *ACM Comput. Surveys*, vol. 47, no. 2, p. 21, 2014.
[8] S. Ji, B. Li, and P. Zhang, "Test case selection for data flow based regression testing of BPEL composite services," in *Proc. IEEE Int. Conf. Services Comput.*, Jun./Jul. 2016, pp. 547–554.
[9] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. (2001). Web services description language (WSDL) 1.1. W3C Note. [Online]. Available: http://www.w3.org/TR/2001/NOTE-wsdl-20010315
[10] A. Alves, A. Arkin, and S. Askary. (2007). Web services business process execution language version 2.0. OASIS Standard. [Online]. Available: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
[11] B. Li, D. Qiu, H. Leung, and D. Wang, "Automatic test case selection for regression testing of composite service based on extensible BPEL flow graph," *J. Syst. Softw.*, vol. 85, pp. 1300–1324, Jun. 2012.
[12] R. Gupta, M. J. Harrold, and M. L. Soffa, "An approach to regression testing using slicing," in *Proc. Int. Conf. Softw. Maintenance*, Nov. 1992, pp. 299–308.
[13] R. Gupta, M. J. Harrold, and M. L. Soffa, "Program slicing-based regression testing techniques," *J. Softw. Test. Verification Rel.*, vol. 6, no. 2, pp. 83–111, 1996.
[14] B. Li, S. Ji, D. Qiu, H. Leung, and G. Zhang, "Verifying the concurrent properties in BPEL based Web service composition process," *IEEE Trans. Netw. Service Manage.*, vol. 10, no. 4, pp. 410–424, Dec. 2013.
[15] S. Ji, B. Li, and D. Qiu, "Incremental verification of evolving BPEL-based Web composite service," *Chin. J. Electron.*, vol. 25, no. 1, pp. 6–12, 2016.
[16] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. New York, NY, USA: Addision-Wesley, 2006, pp. 583–705.
[17] S. Ji, B. Li, and P. Zhang, "XCFG based data flow analysis of business processes," in *Proc. 5th Int. Conf. Inf. Manage.*, Mar. 2019, pp. 71–76.
[18] H. Wang, J. Xing, Q. Yang, D. Han, and X. Zhang, "Modification impact analysis based test case prioritization for regression testing of service-oriented workflow applications," in *Proc. IEEE 39th Annu. Int. Comput., Softw. Appl. Conf.*, Jul. 2015, pp. 288–297.
[19] L. Mei, Y. Cai, C. Jia, B. Jiang, W. K. Chan, Z. Zhang, and T. H. Tse, "A subsumption hierarchy of test case prioritization for composite services," *IEEE Trans. Serv. Comput.*, vol. 8, no. 5, pp. 658–673, Sep./Oct. 2015.
[20] M. Bozkurt, "Cost-aware Pareto optimal test suite minimisation for service-centric systems," in *Proc. 15th Annu. Conf. Genetic Evol. Comput.*, 2013, pp. 1429–1436.
[21] A. Chaturvedi and D. Binkley, "Web service slicing: Intra and inter-operational analysis to test changes," *IEEE Trans. Service Comput.*, to be published, doi: 10.1109/TSC.2018.2821157.
[22] H. Zhong, L. Zhang, and S. Khurshid, "TestSage: Regression test selection for large-scale Web service testing," in *Proc. 12th IEEE Conf. Softw. Test., Validation Verification*, Apr. 2019, pp. 430–440.
[23] S. Sahoo and A. Ray, "A framework for optimization of regression testing of Web services using slicing," in *Proc. Int. Conf. Adv. Comput., Commun. Inform.*, Sep. 2017, pp. 1017–1022.
[24] H. Liu, Z. Li, J. Zhu, and H. Tan, "Business process regression testing," in *Proc. 5th Int. Conf. Service-Oriented Comput.*, 2007, pp. 157–168.
[25] Z. J. Li, H. F. Tan, H. H. Liu, J. Zhu, and N. M. Mitsumori, "Business-process-driven gray-box SOA testing," *IBM Syst. J.*, vol. 47, no. 3, pp. 457–472, 2008.
[26] M. Ruth, F. Lin, and S. Tu, "Applying safe regression test selection techniques to Java Web services," *Int. J. Web Services Pract.*, vol. 2, nos. 1–2, pp. 1–10, 2006.
[27] M. Ruth, S. Oh, A. Loup, B. Horton, O. Gallet, M. Mata, and S. Tu, "Towards automatic regression test selection for Web services," in *Proc. 31th Annu. Int. Comput. Softw. Appl. Conf.*, Jul. 2007, pp. 729–736.
[28] M. E. Ruth, "Employing privacy-preserving techniques to protect control-flow graphs in a decentralized, end-to-end regression test selection framework for Web services," in *Proc. 4th Int. Conf. Softw. Test., Verification Validation Workshops*, Mar. 2011, pp. 139–148.
[29] M. Ruth and C. Rayford, "A privacy-aware, end-to-end, CFG-based regression test selection framework for Web services using only local information," in *Proc. 4th Int. Conf. Appl. Digit. Inf. Web Technol.*, Aug. 2011, pp. 13–18.
[30] M. E. Ruth, "Empirical studies of privacy-preserving regression test selection techniques for Web services," in *Proc. IEEE Int. Conf. Softw. Test., Verification, Validation Workshops*, Mar./Apr. 2014, pp. 322–331.
[31] A. Tarhini, H. Fouchal, and N. Mansour, "Regression testing Web services-based applications," in *Proc. IEEE Int. Conf. Comput. Syst. Appl.*, Mar. 2006, pp. 163–170.

SHUNHUI JI received the B.S. degree in computer science and technology and the Ph.D. degree in computer software and theory from Southeast University, in 2008 and 2015, respectively. She is currently a Lecturer with the College of Computer and Information, Hohai University, Nanjing, China. Her research interests include service computing, cloud computing, software modeling, and analysis, testing, and verification. She is a Reviewer of some international conferences and journals.

BIXIN LI received the Ph.D. degree in software engineering from Nanjing University, in 2001. He is currently a Professor with the School of Computer Science and Engineering, Southeast University, Nanjing, China. He also leads the Software Engineering Institute, Southeast University, and over 20 young men and women are hard working on national and international projects. He has published over 90 articles in refereed conferences and journals. His main research interests include program slicing and its application, software evolution and maintenance, and software modeling, and analysis, testing, and verification. He is a Senior CCF Member.

PENGCHENG ZHANG received the Ph.D. degree in computer science from Southeast University, in 2010. He was a Visiting Scholar with San Jose State University. He is currently an Associate Professor with the College of Computer and Information, Hohai University, Nanjing, China. He has published in premiere or famous computer science journals, such as the IEEE TRANSACTIONS ON BIG DATA, the IEEE TRANSACTIONS ON EMERGING TOPICS IN COMPUTING, the IEEE TRANSACTIONS ON SERVICES COMPUTING, the *Information and Software Technology*, the *Journal of System and Software*, and the *Software: Practice and Experience*. His research interests include software engineering, service computing, and data science. He has served as the Technical Program Committee Member on various international conferences. He was the Co-Chair of the IEEE AI Testing 2019 Conference.