Static inter-BPEL program slicing for web services

Chengying Mao

School of Software and Communication Engineering,
Jiangxi University of Finance and Economics,
Nanchang, Jiangxi Province, 330013, China

and

The State Key Laboratory of Software Engineering,
Wuhan University, Wuhan, Hubei Province, 430072, China
E-mail: maochy@yeah.net

Abstract: Analysis and maintenance of BPEL programs play a vital role in assuring the quality of Web service software. In the paper, the concept of inter-service control flow graph (ISCFG) is proposed to represent the interaction behaviours between service units. Meanwhile, the traditional data flow analysis is extended to handle combinatorial structure of variables in BPEL program. Then, a method for constructing BPEL program dependence graph (BPDG) is addressed according to the above control and data dependence representations. Finally, a static inter-BPEL program slicing algorithm for Web service compositions is proposed, and has been validated via a real-world Web service application.

Keywords: web services; BPEL; SCFG; ISCFG; dependence analysis; BPEL Program Dependence Graph; BPDG; slicing algorithm.

Reference to this paper should be made as follows: Mao, C. (2012) 'Static inter-BPEL program slicing for web services', *Int. J. Simulation and Process Modelling*, Vol.7, No. 3, pp.204–216.

Biographical note: Chengying Mao received the BS in Computer Science and Technology from Central South University, Changsha, China, in 2001, and the PhD in Computer Software and Theory from Huazhong University of Science and Technology, Wuhan, China, in 2006. He worked as a Post-Doctor in the College of Management of Huazhong University of Science and Technology from July 2006 to September 2008. He is an Associate Professor of the School of Software and Communication Engineering in Jiangxi University of Finance and Economics, Nanchang, China. His current research interests include software engineering and data mining. He has published more than 30 papers in the areas of software engineering, SOA and data mining. He is a member of the China Computer Federation (CCF), ACM, IEEE and IEEE CS, also the member of programme committee of GrC'08/09/10/11, ICCIT'09, NCM'09, ICIS'10 and ICIW'10/11.

1 Introduction

As a rapidly emerging technology, web services offer a brand-new mechanism for program-to-program interactions over the internet. This new technology brings great convenience to the construction of distributed, heterogeneous and loose-coupling systems. web service units can be developed by several kinds of programming languages, and run on different platforms. WSS is usually built by composing the service units over the network with some constraints such as QoS, reliability and usability. This software development paradigm takes a full use of software reuse strategy to greatly improve its development efficiency. However, it brings a great challenge to the maintenance activities in the latter stage (Canfora and Penta, 2009).

In general, web service is a well-encapsulated component unit. Therefore, its source code is merely visible to service developers, but invisible to its users, i.e., developers of WSS. When some failures have occurred during the testing process, it is so hard for system maintainers to precisely locate the position of fault. Among all fault localisation and diagnosis methods, program slicing is the typical one and has strong ability of revealing errors. In the paper, we attempt to introduce this technology to debug WSS.

In WSS, its business workflow is usually modelled by using process languages like WS-BPEL (OASIS WSBPEL Technical Committee, 2007), BPML (BPMI, 2002) and WSFL (Leymann, 2001). Among them, BPEL has become the de-facto standard and been widely adopted in industrial applications. Therefore, quality assurance of WSS is mainly to ensure the corrective interaction of BPEL programs in fact. Naturally, we mainly concern on analysing the WSDL (World Wide Web Consortium (W3C), 2001) and BPEL files to produce precise slices for debugging.

In our previous work (Mao, 2009), the slicing algorithm for a single BPEL program is addressed. Here, we extend the existing technique to handle the inter-service invoking problem. At first, control and data flow between multiple BPEL programs are analysed and represented. Then, the system dependence graph of whole WSS is constructed. On the basis of the above-mentioned representation, a static slicing algorithm is proposed. The paper attempts to provide a practical slicing algorithm for WSS and its contributions are addressed as follows:

- The denotations for representing control flow and data flow are provided in our work.
- The dependence relations, including control dependence and data dependence, are analysed for BPEL programs. Meanwhile, construction algorithm for the corresponding dependence graph (i.e., BPDG) is addressed in details.
- A static slicing algorithm based on the above-mentioned dependence analysis is also described.

The remainder of this paper is organised as follows. In Section 2, we introduce the basic constructs and semantic information of BPEL programs. At the same time, a running example program is demonstrated for the following analysis. The control flow analysis and its representation for BPEL programs are described in Section 3. In Section 4, we discuss the control and data dependence relations and their representations. A static slicing method for WSS is addressed in Section 5. The related work and discussion about event handler are addressed in Section 6, and Section 7 concludes the paper.

2 Background

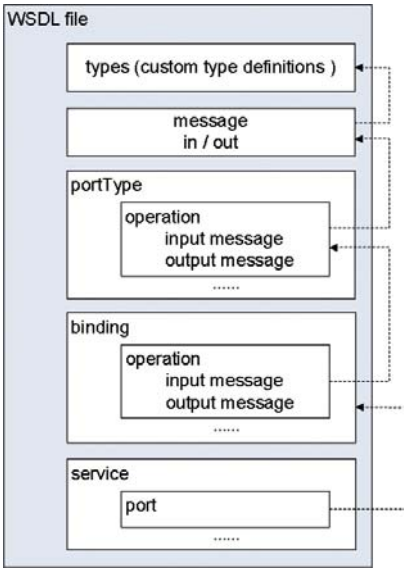
2.1 Web services analysis

A web service is defined by the W3C as "a software system designed to support interoperable machine-to-machine interaction over a network" (World Wide Web Consortium (W3C), 2002). It can also be treated as an internet-based application fulfilling a specific task or a set of tasks, which can be combined with other web services to maintain workflow or business transactions (Aoyama et al., 2002; IBM, 2000).

In fact, web services are usually web Application Programming Interfaces (APIs) that can be accessed over a network such as the internet, and executed on a remote system hosting the requested services (Mao, 2009). The services or workflow templates registered in service agency are implemented via service providers, and their interface information is written in standard description languages such as WSDL. The external developers cannot make use of them before they are published on service registration site.

Web Services Description Language (WSDL) is essentially an XML file, which describes the interface of a web service using an XML standard. As shown in Figure 1, a typical WSDL file mainly contains the following five elements: **types**, **message**, **portType**, **binding** and **service**. **Types** are the data types defined by customer and are exchanged on the interface to a web service. **Message** is the interface information of a service unit, and usually includes two kinds of input and output. The **portType** is a logical grouping of operations, used to identify operations that are intended to be available at the same network location. The **binding** is an association between a portType and a protocol binding such as SOAP over HTTP. Obviously, **service** is the declaration of a service and the ports on which it is available.

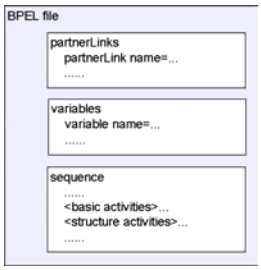
Figure 1 The basic elements of WSDL file



In general, a WSS is a program body composed with several service units. The execution rule of whole WSS is defined by the workflow that is described in BPEL. In fact, BPEL is an XML document conforming to a specific XML schema format, which can describe different aspects of a business process, such as roles, port types and orchestration (OASIS WSBPEL Technical Committee, 2007). From the perspective of the structure at the interface level (as shown in Figure 2), a BPEL process is mainly composed of activities, which can be separated into basic activities, such as **receive**, **reply**, **invoke**, **assign**, **throw** and **exist** as well as structure activities such as **sequence**, **if**, **while**, **repeatuntil**, **pick** and **flow**. It should be noted that the flow (`<flow>`) activity provides concurrency and synchronisation, and is used to define a set of activities that will be invoked in parallel. In general, an operation that is the source of parallel work to be performed is called a concurrent (fork) operation. An operation that is the target of parallel work is called a synchronised (join) operation.

In BPEL, the structural activities with the feature of concurrency and synchronisation cannot be expressed by traditional Control Flow Graph (CFG). Therefore, the graphical representation of BPEL program should introduce some special notations like synchronised node and concurrent node to represent concurrent features.

Figure 2 The typical structure of BPEL file



2.2 Subject system

To better elaborate our slicing method for BPEL programs in WSS, a typical WSS whose workflow process is described via BPEL is introduced here. The subject system is obtained from the samples of Oracle BPEL Process Manager (Oracle Corp., 2009), and its main process is depicted by the file LoanFlow.bpel as shown in code listing 1. This process invokes a synchronous credit rating service, which maybe throws NegativeCredit exceptions, then invokes two asynchronous loan processor services in parallel and finally selects the best loan offer received and returns it (asynchronously) to its caller.

Code Listing 1 The main BPEL program of the LoanDemo system

```
<process name="LoanFlow" … >
 <partnerLinks>
  ... <!-- Include client, creditRatingService, UnitedLoanService and
          StarLoanService -->
 </partnerLinks>
 <variables>
  <variable name="input" .../>
  <variable name="crInput" …/>
  <variable name="crOutput" …/>
  <variable name="crError" …/>
  <variable name="loanApplication" …/>
  <variable name="loanOffer1" …/>
  <variable name="loanOffer2" …/>
  <variable name="selectedLoanOffer" …/>
 </variables>

 <sequence>
01 <receive name="receiveInput" partnerLink="client"
    portType="tns:LoanFlow" operation="initiate" variable="input"
    createInstance="yes"/>
   <scope name="GetCreditRating" >
    <!-- Watch for faults (exceptions) being thrown from
creditRatingService -->
    <faultHandlers>
02    <catch faultName="services:NegativeCredit" faultVariable =
"crError">
03      <assign><copy> <from expression="number(-1000)"/>
          <to variable="input" part="creditRating" /> </copy>
        </assign>
04    </catch>
    </faultHandlers>
   <sequence>
05 <assign> <copy> <from variable="input" part="SSN" />
     <to variable="crInput" part="SSN" /> </copy>
    </assign>
06 <invoke name="invokeCR" partnerLink="creditRatingService"
     portType="services:CreditRatingService" operation="process"
```

Code Listing 1 The main BPEL program of the LoanDemo system (continued)

```
    inputVariable="crInput" outputVariable="crOutput"/>
07 <assign> <copy> <from variable="crOutput" part="rating" />
     <to variable="input" part="creditRating" /> </copy>
    </assign>
  </sequence>
08 </scope>
  <scope name="GetLoanOffer">
   <sequence>
09  <assign> <copy> <from variable="input" >
      <to variable="loanApplication" /> </copy>
    </assign>
10  <flow>
     <sequence>
11     <invoke name="invokeUnitedLoan"
         partnerLink="UnitedLoanService"
         portType="services:LoanService" operation="initiate"
         inputVariable="loanApplication"/>
12     <receive name="receive_invokeUnitedLoan"
         partnerLink="UnitedLoanService"
         portType="services:LoanServiceCallback"
         operation="onResult" variable="loanOffer1"/>
     </sequence>
     <sequence>
13     <invoke name="invokeStarLoan"
         partnerLink="StarLoanService"
         portType="services:LoanService" operation="initiate"
         inputVariable="loanApplication"/>
14     <receive name="receive_invokeStarLoan"
         partnerLink="StarLoanService"
         portType="services:LoanServiceCallback"
         operation="onResult" variable="loanOffer2"/>
     </sequence>
15  </flow>
   </sequence>
  </scope>

  <scope name="SelectOffer" variableAccessSerializable="no">
  <switch> <!-- If loanOffer1 is greater (worse) than loanOffer2 -->
16  <case condition="bpws:getVariableData('loanOffer1','APR') >
      bpws:getVariableData('loanOffer2','APR') ">
17    <assign> <copy> <from variable="loanOffer2" part="APR/>
        <to variable="selectedLoanOffer" part="APR"/> </copy>
      </assign>
     </case>
     <otherwise>
18    <assign> <copy> <from variable="loanOffer1" part="APR"/>
       <to variable="selectedLoanOffer" part="APR"/> </copy>
      </assign>
     </otherwise>
19 </switch>
  </scope>

20 <invoke name="replyOutput" partnerLink="client"
     portType="tns:LoanFlowCallback" operation="onResult"
     inputVariable="selectedLoanOffer"/>
 </sequence>
</process>
```

In the above-mentioned main BPEL program, the details about some elements such as `<partnerLink>` and `<variable>` are omitted here for space consideration. The creditRatingService, UnitedLoanService and StarLoanService are three services invoked by the main process. Here, to demonstrate the invocation between two BPEL programs, we suppose StarLoanService is a composite service. The executable process code of StarLoanService is shown in code listing 2. In this composite service, the workflow firstly invokes the external service IDCheckService to check customer's identification through the SSN number. If the loan requester is a VIP customer and with high credit rate, then the StarLoanService will provide a low Annual Percentage Rate (APR) to the loan requester. Otherwise, the requester can only yield a high APR. Similarly, some unimportant code segments are also omitted here.

Code Listing 2 BPEL program of the invoked service (i.e., StarLoan-Service)

```
<process name="StarLoanService" ...
 <partnerLinks>
   <partnerLink name="LoanFlow" .../>
   <partnerLink name="IDCheckService" .../>
 </partnerLinks>
 <variables>
  <variable name="loanApp" .../>
  <varaible name="sNumber" .../>
  <varaible name="crdRate" .../>
  <variable name="apRate" .../>
  <variable name="loanOffer" .../>
 </variables>

 <sequence>
21 <receive name="receive_loanApplication" partnerLink="LoanFlow"
    portType="services:LoanService"
    operation="initiate" variable="loanApp" createInstance="yes"/>
   <assign>
22  <copy> <from variable="loanApp" part="SSN" />
        <to variable="sNumber" /> </copy>
23  <copy> <from variable="loanApp" part="creditRating" />
        <to variable="crdRate" /> </copy>
   </assign>
24 <invoke name="invokeIDCheck" partnerLink="idCheckService"
     portType="services:IDCheckService" operation="idCheck"
     inputVariable="sNumber" outputVariable="idStatus"/>
 </sequence>
 <sequence>
25<if> <condition idStatus = "vip" crdRate="high" />
26  <assign> <copy> <from expression="number(0.0015)"/>
      <to variable="apRate" /> </copy>
  </assign>
 <else>
27 <assign> <copy> <from expression="number(0.002)"/>
      <to variable="apRate" /> </copy>
  </assign>
28</if>
 <assign>
29  <copy> <from variable="apRate"/>
      <to variable="loanOffer" part="APR" /> </copy>
 </assign>

30  <invoke name="replyLoanService" partnerLink="LoanFlow"
      portType="services:LoanServiceCallback" operation="onResult"
      inputVariable="loanOffer"/>
 </sequence>
</process>
```

3 Control flow representation

For BPEL programs, we also analyse the dependence relationships in two ways: control dependence and data dependence. These dependences have been fully discussed for the traditional procedural and object-oriented programs. However, some features such as concurrence and distributed execution are introduced into WSC. Therefore, these new features should be attached great importance while performing dependence analysis on BPEL programs.

With the gradual improvement of WS-BPEL specification, more and more complex control constructs are added to the specification. As shown in the code of subject system, our dependence analysis method can tackle most of them at present, such as **synchronous** and **asynchronous request** and **fault handlers**. Meanwhile, the element about **event handler** does not exist in the above-mentioned BPEL program. Thus, we will mainly address the constructs analysis for the former three cases, and the treatments for event handler will be presented in the latter discussion section.

3.1 Subject system

Up to now, there are several kinds of notation methods for BPEL activities, among which UML activity diagram, Petri Net (Hamadi and Benatallah, 2003) and BPMN (White, 2004) are three typical representatives. With the direction of BPMN, several kinds of control flow representations including XBFG (Wang et al., 2008), BCFG (Liu et al., 2008) and annotated CFG (Hou et al., 2008) are proposed. To facilitate dependence analysis on BPEL programs, here, we define an extended CFG for a single service unit at first, and then propose the concept about ISCFG.

Definition 1 (*SCFG*)**:** The control flow of a BPEL program can be modelled via Service Control Flow Graph (SCFG), $SCFG = (N, E, n_e, n_x)$, where:

N is the set of nodes representing the executable statements in BPEL program, which can be divided into three categories, i.e., N_{normal}, N_{branch} and N_{con}. Here, N_{normal} is the normal node set as similar to the traditional program, and N_{branch} is the ordinary branch (or choose) node set. N_{con} is a new type of node set representing the concurrent control statement set in BPEL. Especially, n_e and n_x are the entry node and exit node of whole program body, respectively.

E is the edge set of BPEL program, and can be expressed as follows: $E = (n_s, n_t, L)$, where n_s is the source node of a control edge, and n_t is the target node. L is the label of the control edge, such as the condition of a branch edge.

The above-mentioned definition is an integrated form of BPMN and the existing CFGs. To facilitate slicing calculation, SCFG simplifies the graphical representations of the existing related definition, and also mainly adopts the partial denotation in BPMN. In SCFG, there are five kinds of nodes as shown in Figure 3. The first three, i.e., basic activity node, select-branch node and merge node, are similar to the nodes in traditional CFG. However, the last two (concurrent node and synchronised node) are introduced to express the parallel execution behaviours in BPEL process. They are also called parallel gateway and exclusive gateway in BPMN.

Figure 3 Five basic nodes in Service Control Flow Graph (SCFG)

For the control constructs representation of a single service unit, the paper does not discuss it here since it has been fully addressed in reference (Mao, 2009). Here, we mainly concern on the problem caused by inter-service invoking. The invoking actions in BPEL programs can be divided into two kinds: service unit invoking and exception handler invoking. For the former, it can be further divided into two types, i.e., synchronous invoking and asynchronous

invoking. How to model the above-mentioned three special control constructs will be addressed as follows.

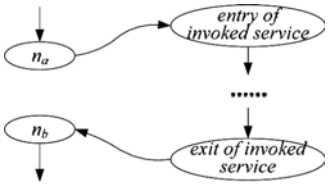
3.2 Modelling for synchronous invoking

In WCS, if the composed service needs to call another service unit, the calling sentence should be written in the following format in BPEL.

```
<invoke name="..." partnerLink="..." portType = "..." operation = "..." input-Variable = "inputVar" outputVariable = "outputVar" />
```

The salient feature of this invoking style lies in that the input variable and output variable are simultaneously assigned in the `<invoking>` statement. If control structure of the invoked service needs to be expanded (e.g., the invoked service is also a composite service or its code is available), the above `<invoking>` statement should be split into two sub-nodes. Suppose a service calling node is labelled as n, this calling site should be split into two parts: n_a and n_b, and the control construct of inter-service invoking can be modelled in Figure 4.

Figure 4 Construct model for inter-service synchronous invoking in BPEL



In the above-mentioned construct model, the function of node n_a is to pass the input parameters to the invoked service. Accordingly, node n_b is used to receive return results. While considering the example BPEL program in this paper, the statements labelled with '06' and '25' are the typical synchronous invocations. Therefore, the service invoking site at 06 (or 25) can be represented with the above-mentioned manner if the invoked service `creditRatingService` (or `idCheckService`) can be further expanded.

3.3 Modelling for asynchronous invoking

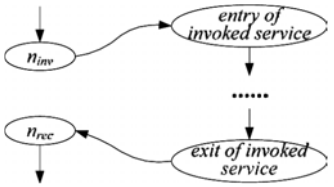
Another typical interaction between web service units is asynchronous invoking. In this invocation style, the request for a service and its response are separately treated as two statements. The general invoking grammar in BPEL can be demonstrated in the following program segment.

```
<invoke name="..." partnerLink="..." portType="..." operation="..." input-Variable= "inputVar"/>
<receive name="..." partnerLink="..." portType="..." operation="..." variable= "outputVar"/>
```

For the asynchronous invoking pattern, it is unnecessary to perform special treatment during modelling process. If the invoked service is a composite service or its internal structure is visible, then ISCFG can be constructed via the manner shown in Figure 5. It should be pointed out that the nodes n_{inv} and n_{rec} represent service invoking statement (i.e., `<invoke>` clause) and message return statement (i.e., `<receive>` clause), respectively.

For the LoanDemo system, both statement pairs 10-11 and 12-13 are asynchronous invocations. Accordingly, the control flow of such statement pair can be modelled in the above-mentioned way if the control construct of invoked service can be expanded.

Figure 5 Construct of inter-service asynchronous invoking



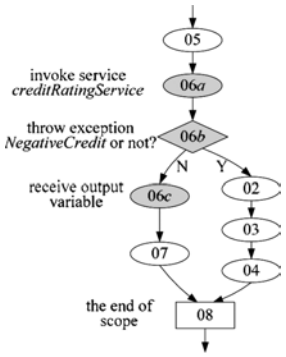
3.4 Modelling for fault exception handlers

Fault handling in WS-BPEL is designed to be treated as 'reverse work', aiming to undo the partial and unsuccessful work of a scope in which a fault has occurred (OASIS OASIS WSBPEL Technical Committee, 2007). The optional fault handlers attached to a scope provide a way to define a set of custom fault-handling activities, syntactically defined as **catch** activities. From the perspective of service invoking, fault handler is used to do some compensation work when the invoking cannot properly work. There are various sources of faults in WS-BPEL, such as `<invoke>` activity and `<throw>` activity. Take the LoanFlow.bpel for an example, the fault handler from '02' to '04' is used to monitor the execution status of service unit creditRatingService. If it throws a fault message with type of NegativeCredit, then the corresponding can be triggered, otherwise not. It is not hard to find that the execution flow of whole BPEL process will be changed by fault handler. In general, if the control flow is converted to the fault handler, then the control flow will point to merge node of the scope with fault handler after the `<catch>` body has been executed.

To represent the control flow with fault-handling artefacts, the service invoking node, which perhaps throws an exception, should be split into three sub-nodes: service invoking sub-node, exception judgement sub-node and output receiving sub-node. Meanwhile, the exit statement of the corresponding scope should also be modelled as a merge node to receive the control flows from the normal execution and exception treatment. Here, we adopt the exception handling constructs in scope 'GetCreditRating' to demonstrate the modelling method for exception control flow.

In the below-mentioned figure, the service invoking node '06' with potential exception in LoanFlow.bpel is divided into three sub-nodes, i.e., 06*a*, 06*b* and 06*c*. The first sub-node is used to invoke the service creditRatingService and pass input variable to it. The second sub-node is an artificial node, responsible for monitoring the appearance of some specific exception, such as NegativeCredit in the example. The last sub-node 06*c* is in charge of collecting return results from the invoked service. Finally, the scope exit statement is used for joint node to merge the normal and exceptional control flows.

Figure 6 The strategy for modelling fault handling structure



3.5 Inter-service Control Flow Graph

On the basis of the above-mentioned analysis for three type inter-service invocations, ISCFG can be represented via the following definition.

Definition 2 (*ISCFG*)**:** An ISCFG can be expressed as a tri-tuple $ISCFG = (S_{SCFG}, E_s, E_e)$, where S_{SCFG} is the set of SCFGs, and each SCFG stands for the control construct of a single service unit or an exception handler. E_s represents the set of edges connecting main service and the invoked service, and E_e represents the set of invoking edges between service and exception handler.

According to the above-mentioned definition, it is not hard to find that ISCFG can be constructed in the following four steps:

Step 1: Construct an SCFG for each service (i.e., BPEL file). The control flow structure for exception handler is ignored at current step.

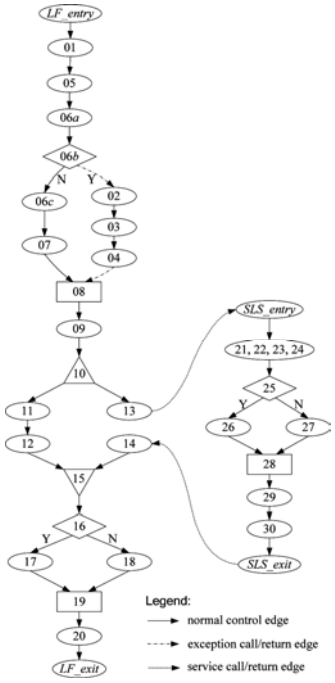
Step 2: For each exception handler, we model its control flow as a traditional method. Since its modelling method is not particular, the detailed process is omitted in this paper.

Step 3: Add the inter-service connecting edge to attach the SCFG of the invoked service to the call site in main BPEL process. Obviously, this type of edge belongs to the set E_s.

Step 4: Add the exception connecting edge to attach the CFG of exception handler to the service invoking point, which perhaps will throw out an exception. As mentioned in the second definition, the set of such edges is E_e.

While considering the example web service system, several services are used, such as creditRatingService, UnitedLoanService and StarLoanService. For the sake of clear representation, here we suppose the service StarLoanService is a composite service, and others are atomic services. The process of composite service StarLoanService is shown in the BPEL file in code listing 2. According to the above-mentioned ISCFG constructing steps, the ISCFG can be built as Figure 7. In the figure, the main control process refers to the LoanFlow service, and the flow between *SLS_entry* and *SLS_exit* represents the business process in StarLoanService.

Figure 7 ISCFG for the aforementioned example system LoanFlow



4 Dependence analysis

4.1 Control dependence analysis

Control dependence and data dependence are two common forms of dependence relations in program, as well as the important foundation for computing slices in general. For WSS, its execution process and information flow are expressed via BPEL program. On the basis of this fact, the following control and data dependence analysis for WSS are mainly intended for the corresponding BPEL programs.

Some elements in BPEL program are not greatly different from the traditional procedural program, so the traditional control dependence relationship also exists in such program. Here, we review the concept about control dependence so as to facilitate the dependence analysis for BPEL program. In general, control dependence is

usually defined in terms of dominance and post-dominance (Ferrante et al., 1987; Tip, 1995).

Definition 3 (*Dominance*/*Post-dominance*)**:** Node n_i dominates n_j if and only if all paths from the start node to n_j intersects n_i. A dual notion is the concept of post-dominance: n_i post-dominates n_j if and only if all paths from n_j to the exit node intersects n_i.

In general, a (post-) dominator tree is used to summarise the dominance/post-dominance relationship between statement nodes in program. Besides basic dominance relation, parallel execution is a prominent character of WSS. Here, we define this feature in the following formal way.

Definition 4 (*Parallel Execution Relation*)**:** Suppose seq_1 and seq_2 are two sequences immediately following a concurrent node in SCFG/ISCFG; these sequences will simultaneously execute when the BPEL program runs on process engine, denoted as $seq_1 \Updownarrow seq_2$. Similarly, the nodes (e.g., n_1 and n_2) in parallel sequences also have such relation in CFG, i.e., $n_1 \Updownarrow n_2$.

For the LoanFlow.bpel file in demo system, the sequence 11→12 has the parallel execution relation with 13→14. Furthermore, the nodes in set {11, 12} also have parallel relation with those in {13, 14}.

Definition 5 (*Parallel Execution Set*)**:** Suppose node $n_1, n_2, \ldots$, and n_s have parallel execution relation with node $r_1, r_2, \ldots$, and r_t, we call all nodes as parallel execution set, i.e., $S_P = \{n_1, n_2, \ldots, n_s\} \bigcup \{r_1, r_2, \ldots, r_t\}$.

Obviously, for the above-mentioned example, the parallel execution set can be yielded as $S_P = \{11, 12, 13, 14\}$. The parallel execution in LoanFlow.bpel file is a simple case for computing the set S_P, which can be directly achieved. For some complex cases, there will be several candidate parallel execution sets if branch statement exists in a parallel execution body.

Without losing generality, control dependence relation-ship in BPEL program can be defined in the similar way to traditional program (Ferrante et al., 1987; Tip, 1995).

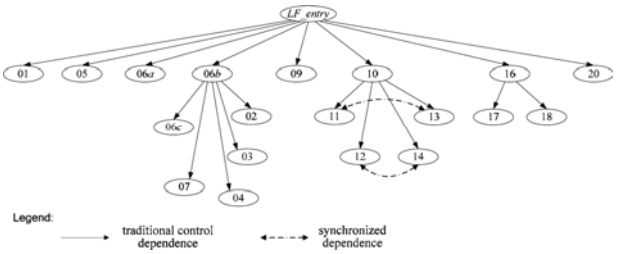
Definition 6 (*control dependence*): A node n_j is (direct) control dependent on a node n_i if:

- there exists a path P from n to n_j such that any $n_k \neq n_i$, n_j in P is post-dominated by n_j, and
- n_i is not post-dominated by n_j.

Determining the control dependences in programs with arbitrary control flow is studied in Ferrante et al. (1987). Here, we should pay much attention to the control dependences caused by the parallel execution relation. For such relation, a special denotation named *synchronised dependence edge* is introduced into the traditional control dependence sub-graph. In our model, it is a bidirectional edge, connecting the first node of one parallel sequence with the first node of the other. Similarly, it also appears between the last nodes of parallel sequences.

On the basis of the above-mentioned analysis, the Control Dependence Graph (CDG) of service LoanFlow can be constructed as shown in Figure 8. It is not hard to find that, in fact, it is a post-dominator tree. In the tree, edges include the traditional control dominance as well as synchronised execution relation. The bidirectional synchronised dependence edge between nodes 11 and 13 implies that if the entry (i.e., node 11) of one parallel sequence has been executed, then the entry (node 13) of the other parallel sequence must also have been executed, and vice versa. The synchronised dependence edge between nodes 11 and 13 implies the similar parallel execution rule for the exit nodes of parallel sequences.

Figure 8 Control dependence graph for the main BPEL program (i.e., LoanFlow.bpel)



Here, we only build the CDG of LoanFlow program to demonstrate the method for constructing dependence relation graph of BPEL program. The CDG of the invoked service StarLoanService can also be constructed in the similar way, but omitted for space reason.

4.2 Data dependence analysis

Data dependence is also an important dependence relation between program elements, which mainly manifests the reference relations about variable value between two statements. The common data dependence (Ottenstein, 1978) of variable definition and use also exists in BPEL programs. However, the variables in BPEL program have their special features such as data composition, so it is necessary to perform some extended analysis on them.

Definition 7 (*Data Definition/Use*): Let v be a variable in program, the assignment of a value to variable v in statement n_d is identified by a tuple $def(v, n_d)$. Meanwhile, the use of v is identified by a tuple $use(v, n_u)$, where the value of variable v is referenced without modifying statement n_u.

For BPEL programs, variable definition mainly occurs in the following cases:

- a receiving activity
- `outputVariable` clause in invoking activity

- in the element `<to .../>` of assign activity.

In general, the use of a variable can be divided into calculation use (*c-use*) and predicate use (*p-use*). In BPEL programs, c-use mainly appears in

- reply activity
- `inputVariable` clause in invoking activity
- in the element `<from .../>` of assignment activity.

The *p*-use of a variable usually appears in a predicate statement, and can be found in the switch condition of case or while statement, or in the `transitionCondition` of state transition.

Take the BPEL program LoanFlow for an example, the variable `input` is defined in the statement 01 and is used in statement 05. Then, this variable is redefined in 07 and is used in 09. While considering the invoked service StarLoanService, the variable `loanApp` is defined in the statement 21 and is used in 22 and 23. For the variable `apRate`, it is defined in 26 or 27, and is used in 29. Detailedly speaking, the variable definitions and uses in BPEL program LoanFlow and StarLoanService can be analysed and represented in Tables 1 and 2, respectively.

Table 1 Variable defs and uses in LoanFlow program

#Node	*Definition*	*Use*
01	Input	
02	crError	
03	input.creditRating	
05	crInput.SSN	input.SSN
06	crOutput	crInput
07	input.creditRating	crOutput.rating
09	loanApplication	input
11		loanApplication
12	loanOffer1	
13		loanApplication
14	loanOffer2	
16		loanOffer1.APR, loanOffer2.APR
17	selectedLoanOffer.APR	loanOffer2.APR
18	selectedLoanOffer.APR	loanOffer1.APR
20		selectedLoanOffer

Table 2 Variable defs and uses in StarLoanService program

#Node	*Definition*	*Use*
21	loanApp	
22	sNumber	loanApp.SSN
23	crdRate	loanApp.creditRating
24	idStatus	sNumber
25		idStatus, crdRate
26	apRate	
27	apRate	
29	loanOffer.APR	apRate
30		loanOffer

Definition 8 (*Def-Use Pair*)**:** A definition-use pair is an ordered pair (n_d, n_u), where a statement called n_d contains a definition of a variable v, which is used in a statement n_u in a program.

According to the above-mentioned example analysis, for the variable `input` in LoanFlow program, the corresponding def-use pairs include (01, 05) and (07, 09) (cf. Table 3). Similarly, def-use pair (21, 22) and (21, 23) for variable `loanApp`, (26, 29) and (27, 29) for variable `apRate` (cf. Table 4).

Definition 9 (*Data Dependence*)**:** Suppose (n_d, n_u) is a def-use pair w.r.t. variable v, node n_u is data dependent on node n_d if there is no variable redefinition of v within the path from n_d to n_u. Accordingly, the path from n_d to n_u is called *clear def-use path* w.r.t. variable v.

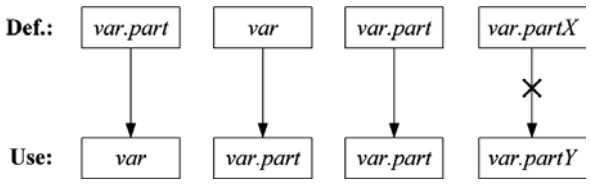
In BPEL programs, variables (especially service's parameters) are defined in service description file, i.e., WSDL file. However, these variables are in the composite form in general, i.e., a variable contains several parts, and each part is an element with basic data type. To represent the data dependence relations caused by such composite variables, the concept of partial data dependence is proposed as follows.

Definition 10 (*Partial Data Dependence*)**:** Suppose variable v is a composite variable, either of the following two conditions have been satisfied, the corresponding def-use pair is called partial data dependence: If the path from n_d to n_u is a clear def-use path, and

- the part of variable v is defined in n_d, and v is used in n_u; or
- variable v is defined in n_d, and the part of v is used in n_u.

For a composite variable, we can list four possible def-use combinations as shown in Figure 9. In the below illustration figure, *var.part* refers to one part element of variable *var*, and *partX* and *partY* stand for two different parts of this variable. In four def-use combinations, (a) and (b) are the typical partial data dependence, and (c) is the general data dependence in the traditional programs. But, case (d) is an illegal def-use pair, so it cannot be viewed as data dependence at all.

Figure 9 Four possible def-use combinations for a composite variable



Whether partial data dependence or common data dependence, they all show that the data dependence relation about a variable exists between the definition site and use site. For the sake of simpleness, here we express the data dependence relations in the form of whole variable. If there is a need for a precise data dependence or precise slicing, the partial data dependence is a preferable choice.

Table 3 Data dependences of variables in LoanFlow program

Variable	*Data dependences*
input	(01, 05) (03, 09) (07, 09)
crError	
crInput	(05, 06*a*)
crOutput	(06*c*, 07)
loanApplication	(09, 11) (09, 13)
loanOffer1	(12, 16) (12, 18)
loanOffer2	(14, 16) (14, 17)
selectedLoanOffer	(17, 20) (18,20)

Table 4 Data dependences of variables in StarLoanService program

Variable	*Data dependences*
loanApp	(21, 22) (21, 23)
sNumber	(22, 24)
crdRate	(23, 25)
idStatus	(24, 25)
apRate	(26, 29) (27, 29)
loanOffer	(29, 30)

For the service invoking node 06, it has been split into three sub-nodes. We can find that 06*a* is a parameter input node, and 06*c* is a parameter output node in fact. A data dependence edge must be added from parameter input node (i.e., 06*a*) to parameter output node (i.e., 06*c*). At the current stage, we analyse the data dependence relations of the main service and invoked service separately. In fact, these two services have data dependence through parameter passing. The information flow caused by service invoking will be discussed in details in the next sub-section.

4.3 Dependence graph construction

On the basis of the above-mentioned control and data dependence analysis, BPDG can be built through extending the traditional PDG construction method. As the traditional PDG, BPDG must include the control and data dependence edges. Moreover, it also contains the synchronised dependence edges and some parameter mapping edges. Therefore, we can define BPDG as follows.

Definition 11 (*BPDG*)**:** Suppose P is a BPEL program with service invocation, its program dependence graph can be denoted as $BPDG(P) = (V, E)$, where V represents the set of statement/parameter nodes in BPEL program, E is the dependence edge set and can be divided into $E = (E_c, E_d, E_{syn}, E_{pcm})$, where, E_c and E_d are the common control and data dependence respectively, E_{syn} stands for the sub-set of synchronised dependence edges, E_{pcm} is the sub-set of parameter/call mapping edges.

In the above-mentioned definition, mapping edge E_{pcm} can be further divided into three categories, i.e., $E_{pcm} = E_{p\text{-}in} \bigcup E_{p\text{-}out} \bigcup E_{call}$, where $E_{p\text{-}in}$ refers to the parameter-in edge set, $E_{p\text{-}out}$ refers to the parameter-out edge set, and E_{call} is the service call mapping edge set. In our model, parameter-in edge is from actual input parameter node to formal input node, and parameter-out edge has the reverse direction. Moreover, call mapping edge is from the call site to the entry of invoked service.

According to the above-mentioned definition, the BPDG of BPEL program P can be constructed in the following three steps: At the first stage, the traditional control dependence relations between statement nodes should be analysed to construct a post-dominator tree, and then the synchronised dependence edges should be added between entry node-pair of each parallel execution body, similar to the exit node-pair. At the second stage, the data dependence shown in Section 4.2 can be analysed, and then a data dependence edge for each dependence pair is added into the above-mentioned CDG. At the last stage, it mainly treats the parameter/call mapping problem. For each service invoking site and entry node of the corresponding service, each parameter should be split into several nodes according to the number of its parts. Then, connect the actual-in node to the formal-in node for each part of input parameter. For the value-return node, the parameter mapping edges can be added in the similar way. In addition, the call site and the called service should also be connected via a mapping edge.

To facilitate program maintainers or testers to build BPDG, the whole construction algorithm can be represented via the following pseudo-code form.

On the basis of the above-mentioned algorithm, the BPEL program dependence graph of the example WSS can be constructed as shown in Figure 10. In the figure, the program dependence graphs of the main service and invoked service are connected together through introducing parameter/call mapping edges. For the invoking clause (i.e., node 13) at service invoking point, two actual input parameters are attached to it. They represent the value passing of part SSN and creditRating in variable loanApplication. Accordingly, two formal input parameter nodes are also attached to the entry of the invoked service.

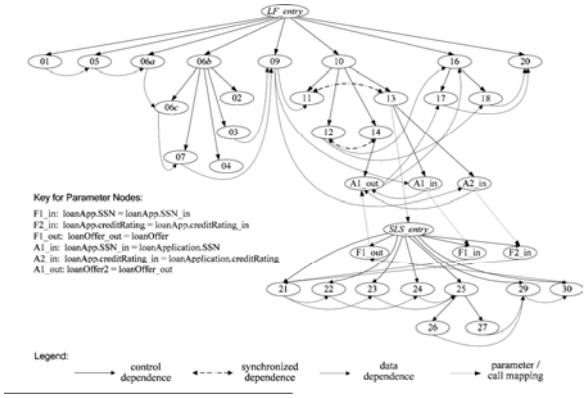
To indicate their map relations, two parameter mapping edges from actual node to formal node are introduced into BPDG. On the other hand, the actual and formal output nodes are also added to BPDG. The difference lies in that the parameter mapping edge is from formal output node to actual output node, or called from the invoked service to the main service. Furthermore, in the sub-BPDG of invoked service, we add some data dependence edges from formal input nodes to the first receive clause of this service. And, a data dependence edge from the last call-back invoke clause (i.e., node 30) to the formal output node is also added. Therefore, the final BPDG contains four types of dependence edges, i.e., control dependence, data dependence, synchronised dependence and parameter/call mapping dependence.

Algorithm. BPDG Construction

Input: A BPEL program P.
Output: BPDG of the program P, i.e. $BPDG(P)$.

```
01 Begin
   Stage 1: Build the control dependence sub-graph
02 for each service unit si in program P do
03  construct the post-dominator tree pdti for si;
04  for each entry node-pair (ep1, ep2) of parallel execution
      body p1 and p2 do
05      add a bidirectional synchronized dependence edge
        between ep1 and ep2 into pdti;
06    endfor
07  for each exit node-pair (xp1, xp2) of parallel execution
      body p1 and p2 do
08      add a bidirectional synchronized dependence edge
        between xp1 and xp2 into pdti;
09    endfor
10 endfor
   Stage 2: Add data dependence edges
11 for each data dependence pair (nd, nu) do
12   add a data dependence edge from nd to nu; // if nd or
     // nu is a service calling node, the corresponding data
     // dependence is transferred to its actual parameter nodes.
13 endfor
   Stage 3: Add parameter mapping edges
14 for each calling site in the main service do
15   add a actual input node for each part of parameter for
     the invoke clause node;
16   add a actual output node for each part of parameter for
     the receive clause node;
17   add a data dependence edge from each parameter input
     node to the parameter output node;
18 endfor
19 for each entry node in the invoked service do
20   add a formal input node for each part of parameter, and
     add a formal output node for return result parameter;
21 endfor
22 add a parameter mapping edge from each actual input
   node to the corresponding formal node;
23 add a parameter mapping edge from each formal output
   node to the corresponding actual node;
24 add a call mapping edge from the call site to the entry of
   invoked service.
25 return the final dependence graph, i.e. BPDG(P);
26 End
```

Figure 10 BPEL program dependence graph for the example web service-based system



5 Static slicing algorithm

Program slicing is a useful program comprehension method proposed by Weiser, which is greatly beneficial to software maintenance activities, such as debugging, testing, measurement and change impact analysis. A *program slice* consists of the parts of a program that potentially affect the values computed at some point of interest (Weiser, 1984), referred to as a *slicing criterion*. Typically, a slicing criterion is denoted with a 2-tuple $C = (node\_number, variable)$. The parts of a program, which have a direct or indirect effect on the values computed at a slicing criterion, C are called the program slice w.r.t. criterion C (Tip, 1995). The slices produced on the basis of the above-mentioned criterion are called *static slices*, and the corresponding process is *static slicing*. The other typical one is *dynamic slicing*, whose slicing criterion is extended as the form of a triple $C_d = (node\_number, variable, value)$. In our current work, we mainly discuss the former case. But the dynamic slices can be easily yielded through properly adapting static slicing algorithm.

Given a slicing criterion, the statements and control predicates, which will affect the variable in it, can be computed by means of a backward traversal of the program from the *node_number* in criterion. Naturally, these slices are referred to as *backward slices*. On the contrary, the other form is to trace dependences in the forward direction, and the corresponding slices are called *forward slices*. In the paper, we mainly focus on how to generate static slices for BPEL programs in WSCs. In fact, algorithms for forward slicing and backward slicing have essential difference. The backward slicing is to produce the relevant statements from the given statement node to program entry. While forward slicing produces the relevant statements from the given node in forward order. This section addresses the backward slicing algorithm in details, and the forward slicing algorithm can run in the similar way.

The static slicing algorithm for BPEL programs is implemented by traversing the BPDG in two phases.

Suppose that slicing starts at node v. The first phase determines all nodes from which v can be reached without descending into service calls. In the second phase, the algorithm descends into all previously stepped service calls and determines the remaining nodes in the slice set. The whole algorithm can be represented via the following pseudo-code.

Algorithm. Static Slicing

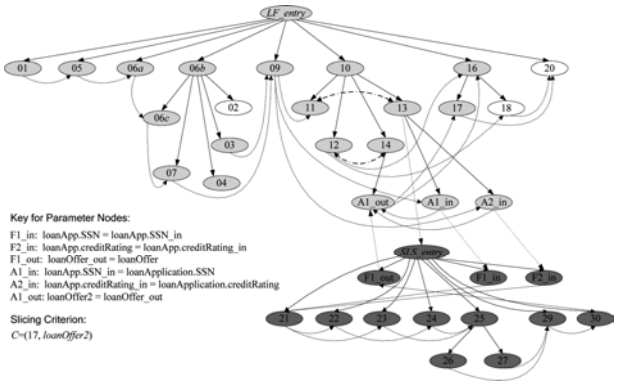
Input: BPDG of the program P, and slicing criterion $C = (lno, var)$.

Output: Static slice set $SS(P, (n, var))$.

```
01 Begin
02  marked_list1 = {lno};
03  work_list = {lno};
    Phase 1: Slice without descending into called services
04  while work_list ≠ ∅ do
05    select a node u from work_list;
06    work_list = work_list − {u};
07    for each v ∈ V(G) and (v, u) ∈ E(G) − E_p-out(G) do
08      if v ∉ marked_list1 then
09        work_list = work_list ∪ {v};
10        marked_list1 = marked_list1 ∪ {v};
11      endif
12    endfor
13  endwhile
14  work_list = marked_list1 ∪ {lno};
15  marked_list2 = ∅;
    Phase 2: Slice called services without ascending to call
    sites
16  while work_list ≠ ∅ do
17    select a node w from work_list;
18    work_list = work_list − {w};
19    for each z ∈ V(G) and (z, w) ∈ E(G) − E_p-in(G) −
      E_call(G) do
20      if z ∉ marked_list2 then
21        work_list = work_list ∪ {z};
22        marked_list2 = marked_list2 ∪ {z};
23      endif
24    endfor
25  endwhile
26  SS(P, (n, var)) = marked_list1 ∪ marked_list2;
27  return SS(P, (n, var));
28 End
```

According to the above-mentioned algorithm, the static slice set can be generated as shown in Figure 11. The nodes in the slice set with respect to slicing criterion $C = (17, loanOffer2)$ are shown shaded. Light shading indicates the nodes identified in the first phase of the static slicing algorithm, and dark shading indicates the nodes identified in the second phase.

Figure 11 Static slicing result w.r.t. the slicing criterion $C = (17, loanOffer2)$



6 Related work and discussion

6.1 Related work

In this section, we review various existing methods for constructs and dependence analysis of web services and discuss their limitation.

Control dependence analysis. Different from the traditional programs, BPEL programs import quite a few new elements such as pick and flow activities. Therefore, how to represent the control flow of the program with these new features is a great challenge. As mentioned in Section 3.1, the most three popular forms for modelling BPEL programs are UML activity diagram, Petri Net (Hamadi and Benatallah, 2003) and BPMN (White, 2004). Besides, Li et al. (2008) developed an RDF (Beckett, 2004) entailment mechanism and applied it to RDF graphs to determine the control flow relation between operations. To model the concurrency and synchronisation characters of WS-BPEL, Liu et al. (2008) and Hou et al. (2008) adapted the BPMN notation to depict the possible process execution flow. Following a similar line of thought, Wang et al. (2008) proposed a control flow representation model XBFG. Although our work also adopts the similar way to represent control flow of web services, it is mainly used for dependence analysis instead of generating test cases as shown in Wang et al. (2008), Liu et al. (2008) and Hou et al. (2008).

The automaton is also an important tool to describe the behaviours of web services (Fu et al., 2004). Keum et al. (2006) suggested a procedure to derive an EFSM model from WSDL description of a service. In EFSM, each node represents a state. While in the Extended Control Flow Graph (XCFG) model, proposed by Yan et al. (2006) to represent a BPEL program, a state of an execution may correspond to more than one node.

Data dependence analysis. Message passing is an important interaction between web service units, so data flow and its dependence analysis is critical for understanding WSCs. Studies in Hou et al. (2008) and Bartolini et al. (2008) essentially adapt data flow coverage criteria conceived for CFG or its extension to generate test cases to validate WSCs. Choy et al. (2008) presented a data-dependency-based approach to alleviate the effort needed for generating test cases. In this approach, data dependencies are defined by using XPath expression, and type definitions in WSDL documents are then leveraged to generate independent data automatically. Since XML is fundamental to many service-oriented workflow applications, Mei et al. (2008) also studied XPath at a concept level and had developed an algorithm to construct XPath Rewriting Graph (XRG) and a novel family of data flow testing criteria to test WS-BPEL applications. Recently, Liu and Chen (2009) identified and discussed various usages of data in WS-BPEL. The contribution of their work is that a method is proposed to abstract the data flow information of various WS-BPEL constructs with the considerations of the concurrency and synchronisation semantics.

In our work, we mainly adopt the traditional concept of data flow and dependence. Its key merit lies in its simplicity, for example, it does not need so complex queries on XML documents as shown in Mei et al. (2008). More importantly, four possible dependences between the basic data item (i.e., part) and its complex composition (i.e., `message` or `portType`) are discussed in our data dependence analysis.

Static slicing for web services. To the best of our knowledge, there is not much work of slicing the BPEL programs in WSCs. In our previous work (Mao, 2009), a static slicing algorithm for a single BPEL program in WSS is proposed. The work in this paper is an in-depth study, and it mainly considers the interactions between web services, i.e., BPEL program invocation. To obtain the precise slices in the case of service calling, the control flow representation and system dependence model are all extended from existing forms in Mao (2009).

6.2 Discussion

As mentioned in Section 3, event handler is another important construct in BPEL program, but the control structure of this element is much more complex than others. In general, each scope, including the process scope, can have a set of event handlers. These event handlers can concurrently run and are invoked when the corresponding event occurs (OASIS WSBPEL Technical Committee, 2007). The basic construct of an event handler can be illustrated as follows:

```
<eventHandlers>
 <onEvent partnerLink="..." portType="..."
 operation="..."            variable="..."
 messageExchange="..." >
  <correlations>
   <correlation                  set="..."
    initiate="yes|join| no" />
   ......
 </correlations>
 <fromParts>
  <fromPart  part="..."  toVariable="..."
   />
  ......
 </fromParts>
 <scope ...> ... </scope>
</onEvent>
<onAlarm>
 (    <for    expressionLanguage="anyURI">
  duration-expr     </for>     |     <until
  expressionLanguage="anyURI">  deadline-
  expr </until> )
 <repeatEvery          expressionLanguage=
  "anyURI"> duration-expr </repeatEvery>
 <scope ...> ... </scope>
 </onAlarm>
</eventHandlers>
```

As shown in the above-illustrated code, there are two types of events. The first is the `<onEvent>` event, which can be inbound messages that correspond to a WSDL operation. The second is the `<onAlarm>` event, which go off after user-set times. Event handlers are considered as a part of the normal behaviour of the scope, unlike fault, compensation and termination handlers. Two types of event handlers have different execution semantics, whose control representations will be addressed as follows, respectively.

Control construct of onEvent. The `<onEvent>` element indicates that the specified event waits for a message to arrive. When the message constituting an event arrives, the `<scope>` activity specified in the corresponding event handler is executed. Because the lifespan of this type of event handler is the whole associated scope, i.e., the scope directly defined within `<onEvent>`, the body of this event handler can be concurrently executed with the normal activity of the scope, and can occur at arbitrary times and an arbitrary number of times while the corresponding scope is active. According to the feature of `<onEvent>` handler, we can model it as a `<receive>` activity. The traditional receive activity is merely an atomic activity, but the `<onEvent>` handler is an execution body, so it should be viewed as isolated procedure and can be invoked anywhere. The control flow between the tag `<onEvent>` and `</onEvent>` can be represented by the method in Section 3.1.

Control construct of onAlarm. The <onAlarm> element marks a time-driven event. The <onAlarm> element specifies a timer event that is fired when the specified duration is exceeded. The clock for the duration starts at the point in time when the parent scope starts. Therefore, we can model the control flow of <onAlarm> handler in the traditional manner. It can be viewed as a parallel execution body of the parent scope of this element. So the entry edge to the head of <onAlarm> control construct is attached to the scope head node, and the exit node of this element points to the merge node of the parent scope.

On the basis of the above-mentioned analysis, the control constructs of BPEL programs even containing event handlers can be represented via SCFG or ISCFG. Therefore, the aforementioned control and data dependence analysis can be performed on the programs with event handlers. Accordingly, the static slicing algorithm is also suited to them.

7 Concluding remarks

Even though web services are becoming more and more widespread as an emerging technology, it is hard to analyse and maintain WSS because it is distributed application with numerous features that are differing from typical applications. Interoperability (Cohen, 2002) is an important character of WSS, so to analyse and model the interactions between service units plays a vital role in system understanding and maintenance. In our previous work, we have analysed the control and data dependences in a single web service. In this paper, we extended the concept of SCFG to ISCFG to express the service invocation relationship. Then, some new control dependence relations such as synchronised dependence are introduced to represent the concurrent execution behaviours. According to the data structure of messages passed between web services, data dependence analysis method for WSS is also proposed here. On the basis of the above-mentioned control and data dependence representations, an algorithm for constructing program dependence graph of BPEL programs is addressed. Finally, the corresponding static slicing algorithm is also presented. A real-world web service application has been

used to validate the above-mentioned methods, and experiment results show that our methods are effective to model and slice BPEL programs with interaction relations. In the future, we plan to make in-depth analysis on the dynamic slicing and other precise slicing algorithms for WSS.

Acknowledgements

This work was supported in part by the National Natural Science Foundation of China (NSFC) under Grant No. 60803046 and 61063013, the Natural Science Foundation of Jiangxi Province under Grant No. 2010GZS0044, the Science Foundation of Jiangxi Educational Committee under Grant No. GJJ10433, the Open Foundation of State Key Laboratory of Software Engineering under Grant No. SKLSE2010-08-23 and the Youth Foundation of Jiangxi University of Finance and Economics.

References

Aoyama, M., Weerawarana, S., Maruyama, H., Szyperski, C., Sullivan, K. and Lea, D. (2002) 'Web services engineering: promises and challenges', *Proc. of ICSE'02*, ACM Press, New York, pp.647, 648.

Bartolini, C., Bertolino, A., Marchetti, E. and Parissis, I. (2008) 'Data flow-based validation of web services compositions: perspectives and examples', *Architecting Dependable Systems V*, LNCS, Vol. 5135, pp.298–325.

Beckett, D. (2004). *RDF/XML Syntax Specification* (*Revised*), W3C Recommendation.

Business Process Management Initiative (BPMI) (2002) *Business Process Modeling Language* (*BPML*), Available at http://xml.coverpages.org/BPML-2002.pdf, pp.1–98.

Canfora, G. and Penta, M.D. (2009) 'Service-oriented architectures testing: a survey', *Proc. of ISSSE 2006-2008*, LNCS, Salerno, Italy, Vol. 5413, pp.78–105.

Choy, K.Y., Ishio, T., Matsushita, M., Inoue, K., Shinomi, H. and Yuura, K. (2008) *Data Dependency Based Test Case Generation for BPEL Unit Testing*, Technical Report 2008-SE-159, Osaka University, Japan, pp.163–170.

Cohen, F. (2002) *Understanding Web Service Interoperability*, IBM Technical Report, Available at http://www.ibm.com/developerworks/library/ws-inter.html

Ferrante, J., Ottenstein, K.J. and Warren, J.D. (1987) 'The program dependence graph and its use in optimization', *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 3, pp.319–349.

Fu, X., Bultan, T. & Su, J. (2004). Analysis of Interacting BPEL Web Services. *Proc. of the 13th Int'l Conf. on World Wide Web*, New York, USA, pp.621–630.

Hamadi, R. and Benatallah, B. (2003) 'A petri net-based model for web service composition', *Proc. of the 14th Australasian Database Conference* (*ADC'03*), Adelaide, Australia, pp.191–200.

Hou, J., Xu, B., Xu, L., Wang, D. and Xu, J. (2008) 'A testing method for web services composition based on data-flow', *Wuhan University Journal of Natural Sciences*, Vol. 13, No. 4, pp.455–460.

IBM (2000) *Web Services: Taking e-Business to the Next Level*. White Paper, available at http://www.ibm.com/developerworks/cn/websphere/download/pdf/e-businessj.pdf

Keum, C., Kang, S., Ko, I., Baik, J. and Choi, Y-I. (2006) 'Generating test cases for web services using extended finite state machine', *Proc. of TestCom 2006*, LNCS, New York, USA, LNCS, Vol. 3964, pp.103–117.

Leymann, F. (2001) *Web Services Flow Language* (*WSFL 1.0*), IBM Corporation, Available at http://xml.coverpages.org/WSFL-Guide-200110.pdf, pp.1–108.

Li, L., Chou, W. and Guo, W. (2008). 'Control flow analysis and coverage driven testing for web services', *Proc. of 2008 IEEE Int'l. Conf. on Web Services* (*ICWS'08*), Beijing, China, pp.473–480.

Liu, C-H. and Chen, S-L. (2009) 'Data flow analysis and testing for web service compositions based on WS-BPEL', *Proc. of the 21st Int'l Conf. on Software Engineering & Knowledge Engineering* (*SEKE'09*), Boston, USA, pp.306–311.

Liu, C-H., Chen, S-L. and Li, X-Y. (2008) 'A WS-BPEL based structural testing approach for web service compositions', *Proc. of the 4th IEEE International Symposium on Service-Oriented System Engineering*, Jhongli, Taiwan, pp.135–141.

Mao, C. (2009) 'Slicing web service-based software', *Proc. of IEEE International Conference on Service-Oriented Computing and Applications (SOCA'09)*, Taipei, Taiwan, pp.91–98.

Mei, L., Chan, W.K. and Tse, T.H. (2008) 'Data flow testing of service-oriented workflow applications', *Proc. of the 30th Int'l Conf. on Software Engineering* (*ICSE'08*), Leipzig, Germany, pp.371–380.

OASIS WSBPEL Technical Committee (2007) *Web Services Business Process Execution Language Version 2.0*, Available at http://docs.oasis-pen.org/wsbpel/2.0/wsbpelv2.0.pdf, pp.1–264.

Oracle Corp. (2009) *Oracle BPEL Process Manager*, Available at http://www.oracle.com/technology/products/ias/bpel/index.html, Accessed on September 2009.

Ottenstein, K.J. (1978) *Data-Flow Graphs as an Intermediate Program*, PhD Dissertation, Computer Sciences Department, Purdue University, Lafayette, IN.

Tip, F. (1995) 'A survey of program slicing techniques', *Programming Languages*, Vol. 3, No. 3, pp.121–189.

Wang, D., Li, B. and Cai, J. (2008) 'Regression testing of composite service: an XBFG-based approach', *Proc. of 2008 IEEE Congress on Services* (*Part II*), Hawaii, USA, pp.112–119.

Weiser, M. (1984) 'Program slicing', *IEEE Transactions on Software Engineering*, Vol. 10, No. 4, pp.352–357.

White, S.A. (2004) *Introduction to BPMN*, Available at http://www.bpmn.org/Documents/Introduction to BPMN.pdf

World Wide Web Consortium (W3C) (2001) *Web Services Description Language* (*WSDL*) *Version 1.1*, Available at http://www.w3.org/TR/wsdl

World Wide Web Consortium (W3C) (2002) *W3C Web Services Activity*, Available at http://www.w3.org/ 2002/ws/, Accessed on September 2009.

Yan, J., Li, Z., Yuan, Y., Sun, W. and Zhang, J. (2006) 'BPEL4WS unit testing: test case generation using a concurrent path analysis approach', *Proc. of the 17th Int'l Symp. on Soft. Reliability Eng.* (*ISSRE'06*), Raleigh, NC, USA, pp.75–84.