2009 Ninth International Conference on Quality Software

Generating Effective Test Sequences for BPEL Testing

Shan-Shan Hou[1,2], Lu Zhang[1,2], Qian Lan[1,2], Hong Mei[1,2], Jia-Su Sun[1,2]
[1]Key Laboratory of High Confidence Software Technologies, Ministry of Education
[2]School of Electronics Engineering and Computer Science, Peking University, Beijing, 100871, China
{houss,zhanglu,lanqian09,meih,sjs}@sei.pku.edu.cn

Abstract

With the popularity of Web Services and Service-Oriented Architecture (SOA), quality assurance of SOA applications, such as testing, has become a research focus. Programs implemented by Business Process Execution Language for Web Services (BPEL), which can compose partner Web Services into activities, are one popular kind of SOA applications. The unique features of BPEL programs bring new challenges into testing. Without explicit user interfaces, a BPEL test case is a sequence of messages that can be received by the BPEL program under test. Although the research on message-sequence generation and instance-routing issues are very popular in testing of object-oriented programs, previous research has not studied the message sequence generation issues induced by unique features of BPEL as a new language.

In this paper, we propose an approach to generating effective message sequences for testing BPEL. In particular, we model the BPEL program under test as a message sequence graph (MSG), and generate message sequences based on MSG. We performed an experimental study on our approach with six BPEL programs. The results show that the BPEL message sequences generated using our approach can effectively expose faults.

1 Introduction

With the emergence of service-oriented architecture (SOA), software engineers can construct service-oriented applications by integrating Web Services. Web Services Business Process Execution Language for Web Services (abbreviated as BPEL in this paper) [23] is one of the most popular Web Services composition languages. BPEL can specify business processes through integrating functionality implemented by Web Services. A business process defined by a BPEL program can also be published as a composite Web Service and imported by other services or processes.

In recent years, BPEL is becoming an industrial standard of Web Services composition, and great efforts have been investigated on related research and implementation. The language has been refined with a new published version [23], and many BPEL execution engines have been developed, such as ActiveBPEL [1], BPWS4J [3], and Oracle BPEL Process Manager [5]. With the popularity of BPEL, the testing of BPEL programs has become a research focus in quality assurance of SOA applications [10, 19, 20, 29].

Among the issues in BPEL testing, test-case generation is one of the toughest issues. Although BPEL programs are published as composite Web Services, the test-case generation of BPEL programs can be very different from that of Web Services [7, 15, 28]. Actually, as BEPL defines a business process by importing partner Web Services' interfaces and composing them in BPEL activities, a BPEL test case is usually a sequence of SOAP messages.

Test-case generation in the form of message sequences is very popular in the testing of object-oriented programs (abbreviated as OOP) [8, 11–13, 16, 27], , and several test-sequence generation tools for OOP have been developed and widely used, such as Randoop [6], JCrasher [4], and Jtest [25]. In OOP, a message that is sent to an object invokes a method of the object. In order to set an object of a class under test into an expected or correct state effectively, researchers have focused on detecting equivalence of object states and proposed techniques to generate test sequences with few ineffective test sequences [11, 27]. As message receptions of BPEL programs can create stateful BPEL instances, it is necessary to study how to set BPEL instances into expected states of executing specific business processes during test. Previous research on BPEL testing [18–21, 29] has not focused on message sequence generation for testing BPEL, which is a new language compared with OOP languages.

Different from the test-sequence generation for OOP testing, there are two specific issues that should be considered in generating test sequences for BPEL programs:

1. The sequence of invocation of methods in classes of OOP is specified explicitly while the sequence of message receptions of a BPEL instance is defined in structures of BPEL programs, such as *sequence*, *flow* and

1550-6002/09 $26.00 © 2009 IEEE
DOI 10.1109/QSIC.2009.50

pick [23]. Therefore, the message sequences sent to a BPEL program should strictly comply with the orders defined in message receptions of the BPEL program.

2. For OOP, the construction of an object is implemented by the class constructor, and a message sent to a specific object of a class can be specified using object references. However, an instance a BPEL program is created by a BPEL activity whose instance creation property is "yes". Furthermore, and the instance routing in BPEL is implicitly implemented using message correlations. That is to say, we can only implicitly claim the target instance that should receive a message with particular parameter values.

In practice, complicated BPEL structures and message correlations for instance routing are widely used in BPEL programs. In order to generate effective message sequences for BPEL testing, in this paper, we consider generating BPEL test sequences based on the preceding unique features of BPEL. Our basic idea is to model structures and message correlations of BPEL programs using message sequence graph (abbreviated as MSG), which can depict the order relations among message receptions and correlations among different messages. Based on MSG, we design a message-sequence generation technique for testing BPEL.

The rest of the paper is organized as follows. Section 2 shows an example. Section 3 presents details of our approach. Section 4 reports an experimental study. Section 5 discusses related work. Section 6 concludes this paper.

2 BPEL Basics and Motivating Example

2.1 BPEL Basics

A BPEL program defines business processes by compositing activities [23]. The basic activity of BPEL includes *invoke*, *receive*, *reply*, and *assign*. The *invoke* activity invokes the functionality of a partner Web Service. The *receive* activity receives a message from outside. The *reply* activity replies a message to the environment. The *assign* activity implements the value assignment operation.

BPEL defines several structures, including *sequence*, *flow*, *pick*, *switch*, *while*, and *If-Then*. *Sequence* specifies a sequential structure containing activities that must be performed in a specific order. A process can exit a *sequence* only if all sequential activities has been executed. *Flow* describes a structure containing several concurrent activities that can execute in any order. A process can exit a *flow* only if all concurrent activities has been executed. In the body of *flow*, *link* defines the synchronization between concurrent activities with a transition condition. *Pick* is used to wait for the occurrence of one of a group of events and only the first responding event can execute. *Switch*, *If-Then*, and *while* are branch structures and the loop structure similar to those of other programming languages.

```
<message name="lockerInfoMessage">
  <part name="lockerNumber" type="xsd:integer"/>
  <part name="data" type="xsd:string"/>
</message>
<message name="statusMessage">
  <part name="information" type="xsd:string"/>
</message>
<bpws:property name="lockerID" type="xsd:integer"/>
<bpws:propertyAlias propertyName="tns:lockerID"
                    messageType="tns:lockerInfoMessage"
                    part="lockerNumber"
                    query="/"/>
```

Figure 1. WSDL Segment of *GymLocker*

2.2 Motivating Example

2.2.1 *GymLocker* Description

GymLocker, which is a sample program of BPWS4J [3], defines a process of the management behavior of a lock, including locking up a lock and unlocking a lock. As shown in Figure 2, the process will lock up a lock after receiving a *lockLockerInfo* message in the first message reception of activity *receive* . Similarly, the process will unlock a lock after receiving an *unlockLockerInfo* message in the second message reception of activity *receive*. The message type of both *lockLockerInfo* and *unlockLockerInfo* is *lockerInfoMessage*, which is defined in Figure 1.

In Figure 2, correlation set *lockerUsage* correlates *lockLockerInfo* and *unlockLockerInfo* by sharing their first part *lockerNumber*. That is to say, if a lock is locked up after receiving a *lockLockerInfo* in a BPEL instance, the lock can only be unlocked after receiving an *unlockLockerInfo* whose *lockerNumber* value is the same with that of the preceding *lockLockerInfo*.

2.2.2 Issues in Generating Test Cases for *GymLocker*

When testing *GymLocker*, ineffective message sequences may be generated for the following issues:

- The message receptions of *lockLockerInfo* and *unlockLockerInfo* are defined in a sequence structure. As the *createInstance* property is *yes*, the reception activity of *lockLockerInfo* creates a new instance. The reception of *unlockLockerInfo* cannot create a new instance as its *createInstance* property is *no*. Therefore, when generating message sequences for *GymLocker*, each *unlockLockerInfo* should have a precursive *lockLockerInfo*. Otherwise, once a *lockLockerInfo* has no subsequential *unlockLockerInfo*, the instance created by the message reception of the *lockLockerInfo* will wait for an *unlockLockerInfo* in the sequence structure. As a result, the created instance can never execute the following unlock activity. In this situation, we cannot test the whole process of *GymLocker*, and the message sequence can be viewed as an ineffective test case.

```
<variables>
  <variable name="lockLockerInfo"
    messageType="tns:lockerInfoMessage"/>
  <variable name="unlockLockerInfo"
    messageType="tns:lockerInfoMessage"/>
  <variable name="status"
    messageType="tns:statusMessage"/>
</variables>
.....
<correlationSets>
  <correlationSet name="lockerUsage"
    properties="tns:lockerID"/>
</correlationSets>

<sequence>
  <receive partner="user" portType="tns:gymLockerP"
    operation="lock" variable="lockLockerInfo"
    createInstance="yes">
    <correlations>
      <correlation set="lockerUsage" initiate="yes"/>
    </correlations>
  </receive>
  .....
  <! Lock a lock >
  .....
  <receive partner="user" portType="tns:gymLockerPT"
    operation="unlock" variable ="unlockLockerInfo"
    createInstance="no">
    <correlations>
      <correlation set="lockerUsage" initiate="no"/>
    </correlations>
  </receive>
  .....
  <! Unlock a lock >
  .....
</sequence>
```

Figure 2. BPEL Segment of *GymLocker*

For example, using message sequence (*lockLockerInfo*, *lockLockerInfo*, *unlockLockerInfo*) to test *GymLocker*, at least the second *lockLockerInfo* will create an idle instance that cannot execute its whole business process for awaiting *lockLockerInfo* in its sequence structure. To reducing the generation of idle instances, our approach aims to generate effective message sequences in which each *lockLockerInfo* is followed by a *unlockLockerInfo*, such as (*lockLockerInfo*, *unlockLockerInfo*, *lockLockerInfo*, *unlockLockerInfo*).

- The part *lockerNumber* of *lockLockerInfo* and that of *unlockLockerInfo* are correlated by correlation set *lockerUsage*. That is to say, if an instance locks up a lock by receiving a *lockLockerInfo*, the lock can only be unlocked after receiving an *unlockLockerInfo* whose value of *lockerNumber* is the same with that of the *lockLockerInfo*. As a result, in a message sequence, if the *lockerNumber* of an *unlockLockerInfo* has never appeared in preceding *lockLockerInfo* messages, the *unlockLockerInfo* will be ignored and the unlock activities of instances created by receiving *lockLockerInfo* will not be executed and tested. This message sequence can be regarded as an ineffective message because it creates idle instances that cannot test BPEL program well.

  For example, (*lockLockerInfo*(1,"first"), *unlockLockerInfo*(2,"second"), *lockLockerInfo*(3,"third"), *unlockLockerInfo*(4,"forth")) is an ineffective test case because none of the instances created by its *lockLockerInfo* messages will execute its unlock process. To reducing the generation of idle instances that keep waiting for *unlockLockerInfo* in the *sequence* structure, our approach aims at generating parameters for correlated messages in a way that a *unlockLockerInfo* has a great chance to have the same *lockerNumber* value with its preceding *lockLockerInfo*, such as (*lockLockerInfo*(1,"first"), *lockLockerInfo*(2,"second"), *unlockLockerInfo*(2,"second"), *unlockLockerInfo*(1,"first")).

3 MSG-based Message-Sequence Generation

3.1 Approach Overview

Following the notion of the motivating example in Section 2, our basic idea is to generate message sequences for BPEL considering both the BPEL structures containing message receptions and instance routing constraints imposed by message correlations. Specifically, our approach employs two main steps.

Step 1: MSG Model Construction. To capture the factors that may affect the quality of generated test cases, we first model the BPEL program under test as a message sequence graph (abbreviated as MSG), which describes both order relations among message receptions in BPEL program's structures and constraints among correlated messages defined by BPEL program's message correlations. We will introduce the definition and construction of the MSG model in detail in Section 3.2.

Step 2: MSG-based test-case generation. To generate message sequences for the BPEL program under test, we design an algorithm to create BPEL message sequences with message parameters according to our MSG model constructed in **Step 1**. We will introduce the details of the algorithm and the strategy in Section 3.3.

3.2 Message Sequence Graph

As mentioned earlier, in order to generate effective message sequences for BPEL testing, we should consider two factors: (1) order relations among message receptions defined by BPEL program's structures; and (2) constraints on correlated messages' values imposed by BPEL programs's correlation sets. Therefore, our MSG model should con-

tain both order relations and correlation constraints of different message receptions. The definition for MSG is given in Definition 1.

Definition 1 (Message Sequence Graph (MSG)) *A message sequence graph for a BPEL program is a 4-tuple* $\langle N_{start}, N_{end}, N, E\rangle$*:*

1. N_{start} *is the beginning node of MSG;*
2. N_{end} *is the ending node of MSG;*
3. N *is a group of nodes* N_i $(1 \leqslant i \leqslant m)$*. Node* N_i *is a tuple* (M_i, C_i, I_i) *that presents a message reception activity.* M *is the message type.* C *is a list of correlation sets that are involved in the reception of* M*.* I *is a boolean flag that indicates if the reception activity of* M *is permitted to create a new BPEL instance.*
4. E *is a group of edges* (N_i,N_j) $(1 \leqslant i \leqslant m,\ 1 \leqslant j \leqslant m,$ *and* $i \neq j$*). Edge* (N_i,N_j) *indicates an edge from* N_i *to* N_j*. If there is an edge* (N_i,N_j) *in a BPEL's MSG, it means that there is a valid BPEL message sequence in which the message reception denoted by* N_j *follows the message reception denoted by* N_i*.*

In this section, we will introduce the details of order relations among messages receptions defined in BPEL's structural activities (Section 3.2.1), correlation constraints among messages (Section 3.2.2), and the construction of MSG (Section 3.2.3).

3.2.1 Order Relations of Message Receptions in BPEL

Message receptions defined in a BPEL program's structures have order relations. According to the BPEL specification [23], we summarize order relations of the message receptions in this section. Supposing that a BPEL program instance B can receive messages $m_1, m_2, ..., m_n$, we analyze the order relations of the receptions of $m_1, m_2, ..., m_n$ $(n \geqslant 2)$ when they are defined in different BPEL structures.

(1) Sequential relation.

The receptions of messages $m_1, m_2, ..., m_n$ have a sequential relation if B must receive the messages in a specific order $(m_1, m_2, ..., m_n)$. That is to say, B must receive m_i before the message reception of m_{i+1} $(1 \leqslant i \leqslant n-1)$. The following two BPEL structures induce sequential relations among message receptions.

- *sequence*. The relation among message receptions defined in a *sequence* is a sequential relation. For example, for the *GymLocker* in Section 2.2.1, the *lockLockerInfo* and *unlockLockerInfo* are received by two *receive* activities in a *sequence*. Therefore, an instance of *GymLocker* can only receive *unlockLockerInfo* after the reception of *lockLockerInfo*.
- *link* of *flow*. The relation among two message receptions, which are respectively the *source* and the *target* of a *link*, is an sequential relation. Structure *link* implies that the activity in *source* must execute before the activity in *target*. For example, for a *link* structure of B, if the activity of its *source* receives m_1 and the activity of its *target* receives m_2, B must receive m_1 before receiving of m_2.

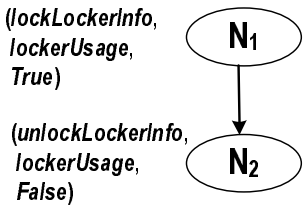

Figure 3. MSG Example: Sequential Relation

Supposing that the receptions of messages $m_1, m_2, ..., m_n$ have sequential relations, we model them in MSG in the following way: First, for the reception of each message m_i $(1 \leqslant i \leqslant m)$, there is a corresponding node N_i=(m_i, C_i, I_i) in MSG. Second, there are $m-1$ edges (N_1,N_2), (N_2,N_3),...(N_{m-1},N_m) in MSG.

For the example BPEL *GymLocker* in Section 2.2.1, relationship among message receptions in the *sequence* is modeled in MSG as shown in Figure 3.

(2) Paratactic relation.

The receptions of messages $m_1, m_2, ..., m_n$ have a paratactic relation if B can receive $m_1, m_2, ..., m_n$ in any order. That is to say, B receives any one of the $n!$ arrangements of $m_1, m_2, ..., m_n$. For message receptions defined in a *flow* structure, if they are not included in the *source* and *target* of a *link*, the relation among them is a paratactic relation. For example, for the BPEL program in the sub-figure (a) of Figure 4, the relationship between the reception of *A* and that of *B* is a paratactic relation. Therefore, an instance of the BPEL program receives both (*A*,*B*) and (*B*,*A*).

Supposing that messages $m_1, m_2, ..., m_n$ have paratactic relations, we model them as a subgraph sub-MSG $\langle N'_{start}, N'_{end}, N', E'\rangle$ in MSG in the following way:

- N'_{start} is the starting node and N'_{end} is the ending node;
- The message receptions of each arrangement of $m_1, m_2, ..., m_n$ have a sequential relation. We view a preceding sequential relation as a composite node of sub-MSG. As there are $n!$ arrangements of $m_1, m_2, ..., m_n$, we model them as $n!$ composite nodes in sub-MSG.
- For each composite node N'_i, there are two edges (N'_{start}, N'_i) and (N'_i, N'_{end}).

For the BPEL program in the sub-figure (a) of Figure 4, relation among message receptions defined in the *flow* is a paratactic relation, and it is modeled in MSG shown in the sub-figure (b) of Figure 4. The two arrangements of the reception of *A* and *B* are presented as two composite nodes containing sequential message receptions.

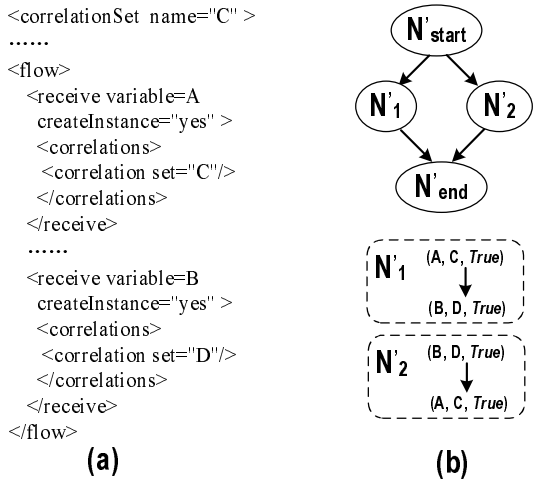

Figure 4. MSG Example: Paratactic Relation

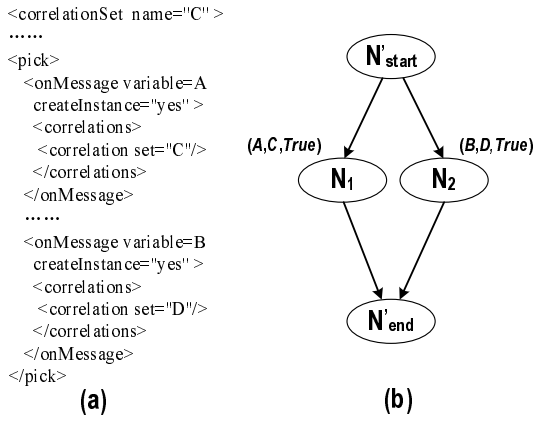

Figure 5. MSG Example: Exclusive Relation

(3) Exclusive relation.
The reception of messages $m_1, m_2, ..., m_n$ have an exclusive relation if an instance of B can only receive one of $m_1, m_2, ..., m_n$. That is to say, once B receives m_i ($1 \leqslant i \leqslant n$), B cannot receive any other message in $(\{m_1, m_2, ..., m_n\} - \{m_i\})$. The following BPEL structures induce exclusive relations.

- *pick*. The relation among message receptions of *onMessage* activities defined in a *pick* structure is an exclusive relation. That is to say, if messages $m_1, m_2, ..., m_n$ are respectively received by *onMessage* activities in a *pick* structure, B awaits the arrival of $m_1, m_2, ..., m_n$ and can only receive message m_i ($1 \leqslant i \leqslant n$) that arrives first.
- *switch*, *if*. The relation among message receptions defined in different *case* branches is exclusive relation. That is to say, if messages $m_1, m_2, ..., m_n$ are received by activities in different *case* branches of a *switch* structure respectively, B can only receive m_i ($1 \leqslant i \leqslant n$) whose reception satisfies the branch condition. Structure *if* can be regarded as a particular *switch* with two branches.
- *while*, *repeatUntil*, *ForEach*. We view a loop structure as one branch entering the loop body and another branch skipping over the loop body. Therefore, the relation among message receptions defined in the two branches is an exclusive relation.

Supposing that messages $m_1, m_2, ..., m_n$ have exclusive relations, we model them as a subgraph sub-MSG $\langle N'_{start}, N'_{end}, N', E' \rangle$ in MSG in the following way:

- N'_{start} is the starting node and N'_{end} is the ending node;
- For the reception of each message m_i ($1 \leqslant i \leqslant m$), there is a corresponding node N'_i=(m_i, C_i, I_i) in sub-MSG;
- For each node N'_i, there are two edges (N'_{start}, N'_i) and (N'_i, N'_{end}).

For example, the sub-figure (a) of Figure 5 defines a *pick* structure awaiting messages *A* or *B*. The relationship between the reception of *A* and that of *B* is modeled in MSG as shown in the sub-figure (b) of Figure 5.

3.2.2 Correlated Relations in BPEL

In order to deliver messages to specific instances of BPEL processes, *message correlation* is defined to specify correlated activities within an instance, including message receptions. A correlation set is a set of properties shared by all messages in the correlated activities. For example, for *GymLocker* in Section 2, correlation set *lockerUsage* shares property *tns:lockerID*, which correlates both the part *lockerNumber* of *lockLockerInfo* and that of *unlockLockerInfo* according to the property alias defined in the WSDL of *GymLocker* in Figure 1.

Based on the message correlations in BPEL, we can initiate the C_i of node N_i in MSG to model the shared parts of different messages in a message sequence.

3.2.3 MSG Construction

In this section, we introduce the details of constructing MSG from the BPEL program under test and its related WSDL files. Basically, we present individual BPEL structures containing message receptions using MSG in the way introduced in Section 3.2.1. In practice, a BPEL program's structures may be very complicated. That is to say, the message receptions in a BPEL program are usually involved in nested structures. In order to distill the relationships and correlations among message receptions from BPEL programs and WSDL files, we design a group of algorithms to construct a MSG model for the BPEL program under test. In particular, the MSG construction includes three main steps.

Step (1): We use *MessageParser* in Figure 6 to parse related WSDL files to acquire the type definition of messages that may be received as variables by the BPEL program under test. *MessageParser* parses the message type

Algorithm *MessageParser*
Input W_B: the WSDL file of the BPEL under test
W: the set of WSDL files imported by BPEL's WSDL
Output M: the set of messages
Begin
For each node E in W_B **Do**
Switch E:
Case message node:
record E's name;
If E's type is defined by $W_i \in W$;
Then Parse W_i recursively;
End If record E's name;
Case property node:
record E's name;
Case propertyAlias node:
record E's name, message type, part name;
End Switch
End Do
For each W_i in W **Do**
If W_i defines property or propertyAlias
Then Parse W_i to record property and propertyAlias;
End If
End Do
End

Figure 6. Algorithm: *MessageParser*

M_i and the property of instance creation I_i to prepare information for a MSG node (M_i, C_i, I_i) in Definition 1. Using *MessageParser* to parse the WSDL file of the example *GymLocker* in Section 2, we obtain a message *lockerInfoMessage*, the property *lockerID*, and its propertyAlias *tns:lockerID*.

Step (2): We construct the order relations among message receptions by parsing the BPEL program under test using *MSG-Construction* in Figure 7. As the relations among message receptions may be defined in nested structures, for each structure containing message receptions, we analyze the structure itself and the nested activities defined in the structure, recursively. For each activity receiving messages, *MSG-Construction* models the message receptions into sequential relations, paratactic relations, or exclusive relations respectively, and transform them as corresponding elements of MSG according to the definition and representation of order relations in Section 3.2.1. Therefore, we create an initial MSG with edges and nodes without correlation information. At the same time, the transition conditions in the BPEL program under test are distilled for generating message parameters.

Using *MSG-Construction* to parse the BPEL program of *GymLocker* in Section 2, we obtain an initial MSG in sub-figure (a) of Figure 8, whose correlation information of nodes is absent.

Step (3): We parse the BPEL program to initiate the correlations of message reception nodes (excluding the beginning and ending nodes of MSG and its sub-graphs) for the

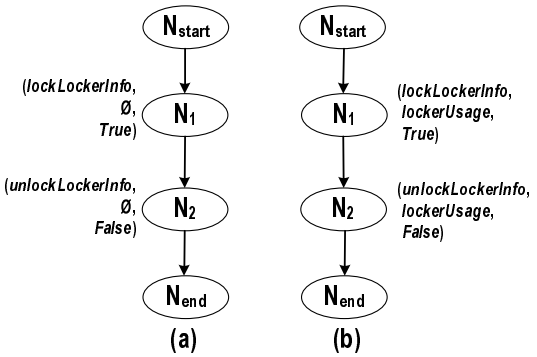

Figure 8. MSG for *GymLocker*

initial MSG. Once a correlation defined in a correlation set is used in a message reception, we locate nodes representing the message receptions in the initial MSG and set their correlation fields. Therefore, we construct a complete MSG model for the BPEL program under test. For *GymLocker* in Section 2, we obtain the MSG containing two nodes with correlation items *lockerUsage* as shown in sub-figure (b) of Figure 8.

3.3 Test-Case Generation Based on MSG

Based on the MSG model constructed in Section 3.2, our approach generates message sequences for testing BPEL. The basic rationale of our approach is that a message sequence, which is generated along a path from the beginning node to the ending node of our MSG and satisfies correlation constraints, is a potential message sequence that can effectively execute the BPEL program under test. Therefore, we consider generating message sequences by exploring potential paths in our MSG (Section 3.3.1) and generating message parameters that satisfy correlation constraints for each node in the paths (Section 3.3.2).

3.3.1 Sequence Generation

Formally, we present the algorithm *MS-Generation* for generating message sequences in Figure 9. In *MS-Generation*, we first traverse the MSG to label "leaf nodes" whose succeeding nodes are the ending node of the MSG. Then we visit nodes in the MSG again to generate message sequences by depth-firstly exploring nodes in MSG. In particular, if the node N is a message node but not a leaf node and all the succeeding nodes of N have been visited, we mark N as "visited"; if the node N is a non-message node N' which has no un-visited succeeding nodes, we mark N as "visited"; if the node N is a leaf node that has not been visited, we mark N as "visited", and the current path P from the beginning node to the leaf node is a message sequence. We continue the search in MSG by back tracking along P to the first non-message node N' that has not been visited.

3.3.2 Parameter Generation

For each message sequence represented by a path acquired using *MS-Generation*, we generate parameters for its mes-

```
Algorithm MSG-Construction
Input B: the BPEL under test; S: the set of messages
Output MSG_I: the initial MSG; T: the set of transition conditions
Begin
  For each node E in B Do
    Switch E
      Case sequence node:          /*Process Sequence Structure*/
        For each children node C of E Do
          If C is receive node  Then Construct sequential relation in MSG;
          Else Construct MSG using C recursively;
          End If
        End Do
      Case Pick node:          /*Process Pick Structure*/
        For each children node C in E Do
          If C is a onMessage node;  Then Construct exclusive relation in MSG;
          Else Construct MSG using C recursively;
          End If
        End Do
      Case Switch, if, or loop node:          /*Process branch and loop Structures*/
        For each branch node C in E Do
          add transition condition into T;
          If C is receive node;  ThenConstruct exclusive relation in MSG;
          Else Construct MSG using C recursively;
          End If
        End Do
      Case flow node:          /*Process flow and link Structures*/
        For each children node C of E Do
          If C has no link;
          Then If C is receive node  Then Construct paratactic relation in MSG;
               Else Construct MSG using C recursively;
               End If
          ElseIf C has link  Then Construct sequential relation for the source and target of link in MSG;
                                  Construct MSG using C recursively;
          End If
        End Do
    End Switch
  End Do
End
```

Figure 7. Algorithm: *MSG-Construction*

sages along the sequence. When generating a parameter for primitive data D for a message M, we consider the following rules:

(1) Message correlations. If the part D of M is correlated to the part D' of a message M' that have been assigned a value in preceding generation process, we assign the same value of D' to D.

(2) Test-data pools. For other situations, we randomly assigns a value to D based on the test-data pools. Some recent research [9] has shown that test-data pools containing boundary values are helpful for generating high-quality test data for random testing. Ciupa et al. [9] randomly selected a value from input domain with the probability *p*. For the remaining probability *1-p*, Ciupa et al. [9] randomly selected a value from the test-data pool. We consider to generate parameters for messages in a similar way. For each type of primitive data, we generate values for test-data pools using equivalence-class partitioning and boundary-value analysis based on the transition conditions in T, which we obtained in Section 7. In particular, for each expression in the form of $E\Theta V$, where E is a variable, Θ is an relational operator, and V is a value, there are two situations. If E is a numeric variable, we put values V, $V-1$, and $V+1$ into our test-data pools. If E is a string or character variable, we put values V into our test-data pools.

4 Experimental Study

We conducted an experimental study to evaluate the effectiveness of our approach. We applied our MSG-based approach on subject BPEL programs (Section 4.1), and re-

```
Algorithm MS-Generation
Input MSG: the MSG
Output MS: a group of message sequences {MS_i} (i ⩾ 1)
Begin
    /*Label leaf nodes */
    For each each node N in MSG Do
        If N has no succeeding node
        Then mark N as a leaf node;
        Else Process the succeeding node of E recursively;
        End IF
    End Do
    /*Generate message sequences using depth-first search*/
    For each node N in MSG Do
      Switch N:
        Case message node but not leaf node
            If All succeeding nodes N have been visited
            Then Mark N as "visited";
            End If
        Case non-message node
            If All succeeding nodes N have been visited
            Then Mark N as "visited";
            End If
        Case leaf node
            If N has not been visited
            Then Mark N as "visited";
                 Output current path P as a message sequence;
                 Back tracking along P to the first un-visited
                 non-message node;
            End If
      End Switch
    End Do
End
```

Figure 9. Algorithm: *MS-Generation*

port our experimental results (Section 4.2). We further discuss threats to validity (Section 4.3).

4.1 Experimental Setup

4.1.1 Subjects

In this experimental study, we used six BPEL programs of some popular BPEL engines (i.e. IBM BPWS4J [3], ActiveBPEL [1] and Apache ODE [2]). In the following section, we refer to these six subjects with their labels shown in the first column of Table 1. Columns Subject, Source, and Loc depict the name of the subject BPEL programs, the source engine publishing the subjects, and the lines of code of the BPEL programs, respectively. Columns STR and COR demonstrate whether structural activities and correlation sets have been used in the subject, respectively.

The two *LoanApproval* programs are different BPEL programs, which have different business processes and are collected from sample libraries of different BPEL engines ActiveBPEL [1] and BPWS4J [3]. We refer to them as *A* and *E* in our study, respectively.

In our study, we ran the BPEL programs of subjects *A*, *B*, and *F* on ActiveBPEL [1], and ran the other BPEL programs on BPWS4J [3]. All the partner Web Services of the proceeding BPEL programs were deployed on web server Tomcat with SOAP engine Axis 1.4.

Table 1. Subject programs

Ref.	Subjects	Source	Loc	STR	COR
A	LoanApproval-1	ActiveBPEL	125	Yes	No
B	ATM	ActiveBPEL	180	Yes	Yes
C	MarketPlace	BPWS4J	68	Yes	Yes
D	GymLocker	BPWS4J	52	Yes	Yes
E	LoanApproval-2	BPWS4J	102	Yes	No
F	BPEL1-5	Apache ODE	50	Yes	Yes

Table 2. Distributions of Seeded Faults

Ref.	A	B	C	D	E	F	Sum
BPEL	3	1	4	2	0	3	13
WSDL	6	0	1	1	2	1	11
XPath	1	3	2	1	3	0	10
$Total$	10	4	7	4	5	4	34

4.1.2 Seeded Faults and Parameter

In order to evaluate the effectiveness of our approach, we created 34 faulty versions for each BPEL program in a similar way with Mei et al. [20]. Faults are seeded to the WSDL files, the XPath query statements, and the BPEL programs themselves by some SOA developers who are not authors of this paper. Table 2 illustrates the distribution of the seeded faults. Columns A to F depict the respective references of the subjects in Table 1. The rows of BPEL, WSDL, and XPath show the numbers of the faults seeded in the BPEL programs, the WSDL files, and the XPath queries, respectively. The last row Total summarizes the total number of seeded faults for each subject. The last column All summarizes the total number of seeded faults of each kind.

According to the experimental results reported by Ciupa et al. [9], the best value of the probability p for selecting new values out of test-data pools is 0.25. In our study, when generating message parameters using *MS-Generation* (Section 3.3), we also set the probability p for generating new values out of our test-data pools as 0.25.

4.2 Results and Analysis

4.2.1 Effective Test Cases

As mentioned in Section 2, message sequences that cannot be accepted by the BPEL program under test are ineffective BPEL test sequences. Also, message sequences that only create idle instances, which do not execute the business logic of the BPEL program under test because their certain structures are stucked for awaiting for messages, are also ineffective test sequences. In our study, for each subject, our approach generated five test suites (denoted as T_1 to T_5) containing 100, 200, 300, 400, and 500 message sequences, respectively. The ratios of ineffective test cases of the five

Table 3. Ratio of Ineffective Test Cases

Ref.	T_1	T_2	T_3	T_4	T_5	Avg
A	3.00	4.00	4.00	4.50	3.60	3.82
B	0.00	0.00	0.00	0.00	0.00	0.00
C	0.00	1.00	1.00	0.75	0.60	0.67
D	0.00	0.00	0.00	0.00	0.00	0.00
E	0.00	0.00	0.00	0.00	0.00	0.00
F	7.00	5.00	4.67	5.00	5.20	5.38

Table 4. Faults Exposure Ratio

Ref.	T_1	T_2	T_3	T_4	T_5	Avg
A	90.00	100.00	100.00	100.00	100.00	98.00
B	100.00	100.00	100.00	100.00	100.00	100.00
C	100.00	100.00	100.00	100.00	100.00	100.00
D	50.00	50.00	75.00	100.00	100.00	75.00
E	80.00	80.00	80.00	80.00	80.00	80.00
F	50.00	100.00	100.00	100.00	100.00	90.00

test suites for each subject are listed in Table 3. Rows A to F depict the respective references of the subjects in Table 1. The second to sixth columns show ratios of the ineffective test cases for T_1 to T_5 for each subject, respectively. The last column shows the average ratio of ineffective test cases of the five test suites. From the table, we can observe that the average ratio of ineffective test sequences of these six subjects is 1.98%, which indicates that our approach can generate test sequences for BPEL programs with very few ineffective ones.

4.2.2 Capabilities for Exposing Faults

We also evaluated the capabilities for exposing faults of the test sequences generated using our approach. Table 4 shows the faults exposure ratios of the five test suites for each subject. Rows A to F depict the respective references of the sample projects in Table 1. The second to sixth columns show faults exposure ratios for T_1 to T_5 for each subject, respectively. The last column shows the average fault exposure ratio of the five test suites.

From the table we can observe that test suites generated using our approach can reveal most faults seeded in the six subjects. The average fault exposure ratio of these six subjects are over 90%. For each subject, with the growing of the size of test suites, the fault exposure ratios are not descending. Furthermore, the faults seeded in both subjects B and C can be revealed by all their test suites. For subject A, D, and F, the capabilities of exposing faults of generated test sequences are increasing with the growing of the size of test suites. We suspect that the reason of this phenomenon lies in the random value generation for message parameters. That is to say, with the growing of the number of generated values, the chance to hit fault-exposure values is increasing.

4.3 Threats to Validity

In our experimental study, there are two main threats to external validity. First, we use only six subjects. The particular characteristics of the subjects may have effect on our experimental results, which may not be generalized to other BPEL programs. To reduce this threat, we plan to evaluate our approach using several other BPEL programs in future work. Second, the subjects and the used Web Services are implemented in two BPEL engines and Tomcat server. Our experimental results may not be generalized to other situations. To reduce this threat, we plan to do more experiments using large-size BPEL programs with partner Web Services running on different platforms.

5 Related Work

In recent years, testing BPEL becomes a research focus in software testing. Mei et al. [20] focused on the XPath and developed the data-flow approach for testing BPEL. They defined a group of test adequacy criteria based on data-flow associations for testing BPEL. Furthermore, they applied their criteria in test-case prioritization for BPEL regress testing [21]. Fu et al. [14] modeled BPEL as the guarded automata and used SPIN model checker to verify BPEL. Li et al. [18] and Mayer [19] proposed testing framework for BPEL. Dong et al. [10] proposed to translate BPEL to Petri-Net and use existing Petri-Net tools for BPEL verification. All the preceding research has not discussed test-case generation for BPEL testing.

Recently, Yan et al. [29] proposed to generate test cases for BPEL using concurrent path analysis. They modeled event handlers and concurrent activities such as *link* and *flow* in the extended control-flow graph, and generated test cases to execute paths by resolving paths' constraints. This approach is basically a path-wise approach, which aims at generating message parameters but not message sequences for BPEL testing. However, we focus on the message-sequence generation for BPEL, which has not been discussed in previous work of BPEL testing.

Graph models have been widely used to generate sequences for testing software, such as using state diagrams [17,24], activity diagrams [26], and collaboration diagrams [24] of UML models to generate OO test sequences and using the event-flow model [22] to generate GUI test sequences. In this paper, we investigate on generating test sequences for BPEL based on the graph model MSG.

6 Conclusion

BPEL, which is a popular flow language for Web Service composition, defines a business process by importing parter

Web Services' interfaces and composing them in BPEL activities. We need to generate BPEL test cases in the form of message sequences. The complicated BPEL structures and the instance routing mechanism of BPEL bring new issues to BPEL testing. Although message-sequence generation and instance routing are very popular in testing object-oriented programs, previous research has not studied the message-sequence generation issues induced by unique features of BPEL as a new language.

In particular, we proposed a message-sequence generation approach based on the message sequence graph (MSG) for BPEL testing. The information modeled by MSG works as the guide to generate both message sequences and parameters. We construct MSG by analyzing order relations among message receptions in structures and constraints induced by message correlations of BPEL programs. Also, we proposed an algorithm and a strategy to generate message sequences and parameters based on our MSG. We performed an experimental study on our approach with several BPEL programs. The results show that our approach can generate effective BPEL test sequences with good capability of exposing faults.

Acknowledgments. This research is sponsored by the National Basic Research Program of China under Grant No. 2009CB320703, the High-Tech Research and Development Program of China under Grant No. 2007AA010301, the Science Fund for Creative Research Groups of China under Grant No. 60821003, and the National Science Foundation of China No. 90718016.

References

[1] ActiveBPEL. Available at http://www.activebpel.org.

[2] Apache ODE. Available at http://ode.apache.org/.

[3] BPWS4J: a Platform for Creating and Executing BPEL4WS Processes, Version 2.1. Available at http://www.alphaworks.ibm.com/tech/bpws4j.

[4] Jcrasher. Availabe at http://ranger.uta.edu/c̃sallner/jcrasher/.

[5] Oracle BPEL Process Manager. Available at http://www.oracle.com/technology/bpel/index.html.

[6] Randoop. Available at http://people.csail.mit.edu/cpacheco/randoop/.

[7] A. Bertolino, G. D. Angelis, and A. Polini. A QoS test-bed generator for web services. In *Proc. International Conference on Web Engineering*, pages 17–31, 2007.

[8] H. Y. Chen, T. H. Tse, F. T. Chan, and T. Y. Chen. In black and white: An integrated approach to class-level testing of object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 7(3):250–295, 1998.

[9] I. Ciupa, A.Leitner, M. Oriol, and B. Meyer. Experimental assessment of random testing for object-oriented software. In *Proc. International Symposium on Software Testing and Analysis*, pages 84–94, 2007.

[10] W. L. Dong, H. Yu, and Y. B. Zhang. Testing BPEL-based web service composition using High-Level Petri Nets. In *Proc. International Conference Enterprise Distributed Object Computing*, pages 441–444, 2006.

[11] R. Doong and P. Frankl. The astoot approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, 1994.

[12] S. Fiedler. Object-oriented unit testing. *Hewlett-Packard Journal*, 40(2):69–74, 1989.

[13] R. Fletcher and A. Sajeev. A framework for testing object oriented software using formal specifications. In *Proc. International Conference on Reliable Software Technologies*, pages 159–170, 1996.

[14] X. Fu, T. Bultan, and J. Su. Analysis of interacting bpel web services. In *Proc. International Conference on World Wide Web*, pages 621–630, 2004.

[15] S. Hanna and M. Munro. An approach for specification-based test case generation for web services. In *Proc. International Conference on Computer Systems and Applications*, pages 16–23, 2007.

[16] M. Harrold and J. Mcgregor. Incremental testing of object-oriented class structures. In *Proc. International Conference on Software Engineering*, pages 68–80, 1992.

[17] J. Hartmann, C. Imoberdorf, and M. Meisinger. UML-based integration testing. In *Proc. International Symposium on Software Testing and Analysis*, pages 60–70, 2000.

[18] Z. Li, W. Sun, Z. B. Jiang, and X. Zhang. BPEL4WS unit testing: Framework and implementation. In *Proc. International Conference on Web Services*, pages 103–110, 2005.

[19] P. Mayer and D. Lubke. Towards a BPEL Unit Testing Framework. In *Proc. International Workshop on Testing, Analysis, and Verification of Web Services and Applications*, pages 33–42, 2006.

[20] L. Mei, W. Chan, and T. Tse. Data flow testing of service-oriented workflow applications. In *Proc. International Conference on Software Engineering*, pages 371–380, 2008.

[21] L. Mei, W. Chan, and T. Tse. Test case prioritization in regression testing of service-oriented business applications, to appear. In *Proc. International Conference on World Wide Web*, pages 371–380, 2009.

[22] A. M. Memon. An event-flow model of gui-based applications for testing. *Software Testing, Verification and Reliability*, 17(3):137–157, 2007.

[23] OASIS Web Services Business Process Execution Language (WSBPEL) Technical Committee. *Web Services Business Process Execution Language Version 2.0*, 2007. http://docs.oasis-open.org/wsBPEL/2.0/wsBPEL-v2.0.html.

[24] J. Offutt and A. Abdurazik. Generating tests from uml specifications. In *Proc. International Conference on the Unified Modeling Language*, pages 416–429, 1999.

[25] Parasoft. Jtest manuals version 4.5. Online manual. 2002.

[26] L. Wang, J. Yuan, X. Yu, J. Hu, X. Li, and G. Zheng. Generating test cases from uml activity diagram based on gray-box method. In *Proc. Asia-Pacific Software Engineering Conference*, pages 284–291, 2004.

[27] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proc. International Conference on Automated Software Engineering*, pages 196–205, 2004.

[28] W. Xu, J. Offut, and J. Luo. Testing web services by xml perturbation. In *Proc. International Symposium on Software Reliability Engineering*, pages 257–266, 2005.

[29] J. Yan, Z. Li, Y. Yuan, W. Sun, and J. Zhang. BPEL4WS unit testing: Test case generation using a concurrent path analysis approach. In *Proc. International Symposium on Software Reliability Engineering*, pages 75–84, 2006.