Automatic Test Case Generation for BPEL Using Stream X-Machine

Chunyan Ma[1,2], Junsheng Wu[1], Tao Zhang[1], Yunpeng Zhang[1], Xiaobin Cai[2]

[1]College of Software and Microelectronics, Northwestern Polytechnical University, Xi'an 710072,   China
[2]Institute of Computer Science, Northwestern Polytechnical University, Xi'an 710072,   China
machunyan@nwpu.edu.cn

Abstract. To generate test cases for the unit testing of business process written in BPEL, developers have to prepare input data for the BPEL process under test (PUT) and verification conditions for output data from the PUT. It could be a tedious task due to the complexity of the PUT which describes the web flow of a distributed collaboration of individual service providers executing concurrently. This paper shows how Stream X-machine (SXM) based testing techniques could be applied to automatically generate test cases for BPEL process. The SXM describes a system as a finite set of states, an internal memory and a number of transitions between the states. One of the strengths of using a SXM to specify a system is that, under certain well defined conditions, it is possible to produce a test suite that is guaranteed to determine the correctness of the implementation under test.

Keywords: BPEL; test cases; Stream X-machine;

1   Introduction

Web Services Business Process Execution Language (WS-BPEL or BPEL) [1] is designed to compose Web services in realizing Service-Oriented Architecture (SOA). As more and more workflows modeled using BPEL, the quality of BPEL program is of crucial importance to the enterprise, since the malfunction of the process may have significantly negative financial impact on it. Thus, unit testing of Web Service compositions becomes increasingly important for ensuring good-quality BPEL code.

Unit testing for BPEL has been addressed by several works. Mayer and Lübke [2] have proposed a framework for performing white-box unit testing. Nevertheless, non-systematic way for defining test cases is presented. In the work of Yan et al. [3] and Yuan et al. [4], a BPEL process is first analyzed and translated into extended control flow graphs from which test paths are extracted. However, for the current activity example in section 11.6.4 [1], test paths obtained is incomplete, and test data generation for operation sets is not considered. Y. Zheng [5] presented an automatic test generation framework for BPEL.

For large and complex BPEL, it is tedious and difficult to manually generate test cases, so a way to generate test cases automatically is imminently needed. In this paper, we model the specification of BPEL process with stream X-machines. Using stream X-machines we can: (a) present the sequence of interactions of all Web services of BPEL process; (b) present control flow information of all activities and enough data flow information of BPEL process (c) generate all the sequential test paths, test data and verification conditions for output data automatically. The testing process can therefore be performed automatically. We could check whether it is identical between the output sequences produced by the BPEL process implementation and the ones expected from the BPEL process model.

This paper is structured as follows. Section 2 presents a background on relevant concepts and definitions. Section 3 presents a method for transforming BPEL process into its corresponding stream X-machine model, and demonstrates the transformation by using the example of loan approval process. Automatic test cases generation is shown in Section 4. Remarks and outlook for future research are presented in Section 5.

2   Background

2.1   BPEL

BPEL employs a distributed concurrent computation model with variables. It can express a causal relationship between multiple invocations by means of control and data flow links. BPEL provides three kinds of activities to exchange information with the outside Web service providers: *invoke*, *receive*, and *reply*. In addition to the communication primitive activities, BPEL provides an *assign* activity for accessing variables. It also include other activities concerning to implement control flows such as *sequence*, *if*, *while*, *repeatUntil*, *pick*, *flow*, *forEach* and *scope*. Loan approval process is an example to use a *flow* activity and links in the BPEL specification [1]. This process will be used as an example in this paper.

2.2   Stream X-machines

A stream X-machines (SXM) is a tuple $Z=(\sum, \Gamma, Q, M, \Phi, F, q_0, m_0)$ [8], where
- $\sum$ and Γ are finite sets called the input alphabet and output alphabet respectively.
- Q is a finite set of states.
- M is a (possibly) infinite set called memory.

- Φ is a finite set of distinct *processing functions*; a process function is a non-empty (partial) function of type $M\times\Sigma\leftrightarrow\Gamma\times M$.
- F is the (partial) next state function, $F:Q\times\Phi\rightarrow Q$, F is usually described by a state-transition diagram.

- $q_0 \in Q$ is the initial state.
- $m_0 \in M$ is the initial memory value.

SXM is a formal method, which is capable of modeling both the data and the control of a system. Transitions between states are performed through the application of functions. In contrast to finite state machines, SXM are capable of modeling non-trivial data structures by employing a memory, which is attached to the SXM. Functions receive input symbols and memory values, and produce output while modifying the memory values.

3   From BPEL to Stream X-Machines

To transform BPEL process to the corresponding SXM, we first determine the states and the transitions, then the memory structure, the input and output sets, and finally we define the processing functions.
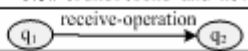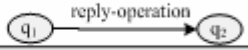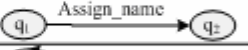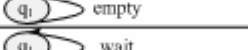
3.1   States and Transitions
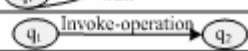
Fig. 1.   The SXM fragment of basic activity

A BPEL process is translated into a main SXM. States and transitions are obtained by examining the activity and its sub-activities in BPEL process. BPEL activities are divided into 2 classes: basic and structured. Each activity is translated into an appropriate SXM fragment which has a starting state and an ending state. Basic activities translations are mostly straight and shown in figure 1.

Structured activities prescribe the order in which a collection of activities is executed. There are *sequence*, *if*, *while*, *repeatUntil*, *pick*, *flow* and the *scope* in BPEL2.0 [1].

Assume that s is the *sequence* activity and activities a_1, a_2,…,a_m are included orderly in it. We translate s into a SXM fragment M. The translation process consists of two steps. First, each a_i ($1 \leq i \leq m$) is translated into an appropriate SXM fragment M_i. Second, the ending state of M_i is as the starting state of M_{i+1}, where $1 \leq i \leq m$. The starting state of M_1 is as the starting state of M. The ending state of M_m is as the ending state of M.

If provides a conditional behavior. The activity consists of an ordered list of one or more conditional branches defined by the *if* and the optional *elseif* elements, followed by an optional *else* element. Its SXM fragment is presented by figure 2. In order to have the unique starting and ending state of the SXM fragment, We introduced an empty processing function ε. It denoted that an empty transition between states. We introduce auxiliary predicate processing functions *if_condition, elseif1_condtion, elseif2_condition… else_condition* for conditional expressions on *if* activity.

Both *while* and *repeatUntil* provide for repeated execution of a contained activity. Their SXM fragments are shown in figure 2. We introduce auxiliary predicate processing functions *while_condition, ~while_conditon, until_condition* and *~until_condition* for conditional expression on *while* and *repeatUntil*.

The *pick* activity is comprised of a set of branches, each containing an event-activity pair. The *pick* activity completes when the selected activity completes. Its SXM fragment is shown in figure 2.
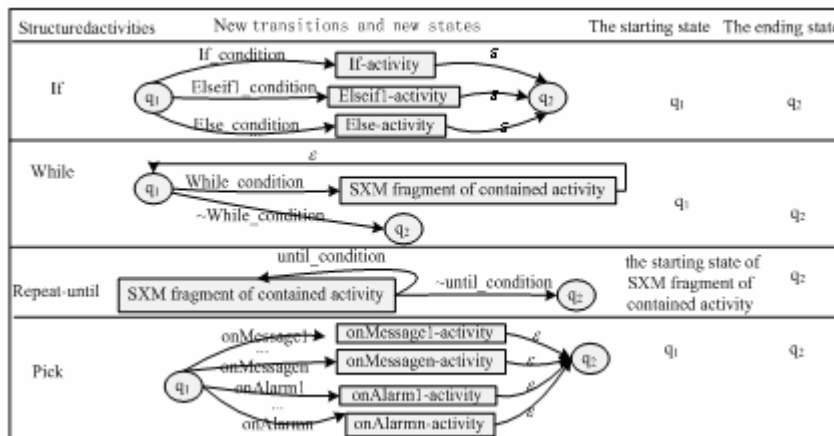

Fig. 2.   The SXM fragments of *if*, *while*, *repeatUnti*l and *pick*

The translation of the *flow* activity is the most complex. It provides concurrency and synchronization for a set of activities, in order to apply SXM to obtain test cases for all possible execution paths, Its SXM fragment construction consists of the following steps:

Step1. Each sub-activity appearing directly in the *flow* is translated into a SXM fragment.

Step2. For all sub-activities, we denote their concurrency with their alternant occurrence. Let $a_1, a_2, … a_m$ denote all sub-activities of the *flow* activity, then the number of their alternant occurrence sequences is P_m^m. We denote the activity sequences set of the number of P_m^m as **S**.

Step3. Compute all **order pairs** set **C**. For the activity pair $a_i a_j$, if a link exists, and a_i as source, a_j as its target, we call $a_i a_j$ an **order pair**. The **order pair** presents the synchronization dependency between activities.

Step4. $\forall s \in \mathbf{S}$. if the order of the activities occurrence in s disobeys the order of any **order pair** in C, then s is an illegal activity sequence which disobeys synchronization dependency. Obtain all illegal activity sequences set **S1**. Let **S2**=**S-S1**.

Step5. $\forall s \in \mathbf{S2}$. Assume $s = b_1 b_2 \ldots b_m$, and their SXM fragments are M_1, M_2, ..., M_m respectively. We translate s into a SXM fragment M. (1). The starting state of M_1 is as the starting state of M. (2). The ending state of M_m is as the ending state of M. (3). For the consecutive activities $b_k b_{k+1} (1 \le k \le m)$ in s, let t denote the number of b_{k+1} target links. If t equals 0 or the join condition of the activity b_{k+1} must be true, then the ending state of M_k and the starting state of M_{k+1} is combined into one state. Otherwise, if the join condition of the activity b_{k+1} may be true or false, we introduce two new transitions from the ending state of M_k. Their processing functions which encapsulate join condition of the activity b_{k+1} are named targetName_f_1 and targetName_f_2 respectively. The transition whose processing function targetName_f_1 returns *true* leads to the starting state of M_{k+1}. The transition whose processing function targetName_f_2 returns *false* leads to the ending state of M_{k+1} (namely dead-path-elimination (DPE)).

Step6. Assume **S2**={ s_1, s_2,..., s_j }, and s_1, s_2,..., s_j corresponding SXM fragments built by the step 5 are Ms_1, Ms_2, ..., Ms_j respectively. We combine Ms_1, Ms_2, ..., Ms_j into one SXM fragment M with $\boldsymbol{\varepsilon}$. We introduce two new states q_1 and q_2 which are as the starting state and the ending state of M respectively. The $\boldsymbol{\varepsilon}$ transitions of q_1 lead to the starting state of Ms_1, Ms_2, ..., and Ms_j respectively. Their ending states lead to q_2 respectively by $\boldsymbol{\varepsilon}$ transition. Finally M is the SXM fragment of *flow* activity.

The SXM fragment of the *scope* is the SXM fragment of the enclosed top sub-activity.

The translation of a BPEL process consists of two steps. First, each top sub-activity is translated into an appropriate SXM fragment which has a starting state and an ending state. Second, their asynchronous product in the same order appearing in BPEL composes a main SXM. The starting state of the first activity is as the initial state. The ending state of SXM fragment of the previous activity is as the starting state of SXM fragment of the current activity. So the obtained SXM for BPEL process is nondeterministic. Standard finite-state automata minimization approach [7] can be used here to turn a nondeterministic SXM into a behaviorally-equivalent deterministic SXM (namely DSXM which is defined in [8]). The deterministic state transition diagram of loan approval process is depicted in Figure 3.

3.2   Memory

WS-BPEL variables provide the means for holding messages that constitute a part of the state of a business process. The messages held are often those that have been received from partners or are to be sent to partners. Variables can also hold data that are needed for holding state related to the process and never exchanged with partners. So the memory structure can be directly derived from BPEL variables. In this paper M is a finite set of BPEL variables. For the loan approval process, let M={*request*, *risk*, *approval*}. Since memory has to be filled with specific values, the initial memory value m_0={ *null*, *null*, *null*}.
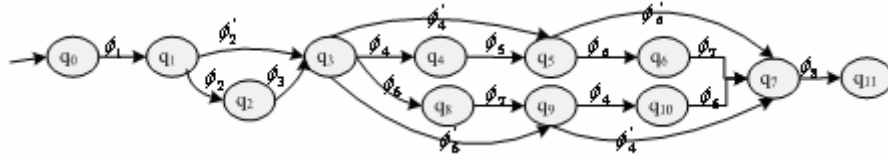
Fig. 3.   The state transition diagram of loan approval process

3.3   Processing Functions

In our case, we define three kinds of processing functions. For each processing function we have to define the input and the memory state that trigger the function, the output that the function produces and the updated memory.

   (1) Operation processing functions

   Only basic activities *receive*, *reply* , and *invoke* define communications with external service providers. In our case, the operations defined by *receive*, *reply*, and *invoke* are **operation processing functions**. The input and output of the **operation processing function** are the input message and the output message of the operation defined in the corresponding activity. For the loan approval, operation processing functions have ϕ_1 (*receive_request*), ϕ_3 (*invoke_check*), ϕ_5 (*invoke_approve*), and ϕ_9 (*reply-request*). *Receive_request* denotes the name of ϕ_1. The *receive* activity specifies variables using the *variable* attribute or *fromPart* elements to be used to receive the message data. So the memory variables are updated. For the loan approval, the complete form of process function ϕ_1 is as follows: ϕ_1: *(m, CreditInformationMessageV)* $\rightarrow$ *($\perp$, m_1)*, Where, *CreditInformationMessageV* is one value of the *creditInformationMessage* type which is as the input message; we assume that *m* ={ *valueOne*, *valueTwo*, *valueThree*}, then m_1={*CreditInformationMessageV, valueTwo, valueThree*}. *OnMessage* event in *pick* activity is similar to a *receive* activity.

   The *reply* activity specifies variables using the variable attribute or *toPart* elements to be used to indicate the response message. So the memory variables are not updated. For the loan approval, the complete form of process function ϕ_9 is as follows: ϕ_8: *(m, $\perp$)* $\rightarrow$ *(approvalMessageV, m)*, Where *approvalMessageV* is one value of the *approval-Message* type which is as the output message.

   For the corresponding process function with *Invoke*, there are one-way invocation and request-response invocation. One-way invocation requires only the inputVariable (or its equivalent *toPart* elements) since a response is not expected as part of the operation. So its memory variables of the One-way invocation are not updated. Its input is the input message of the operation defined in the activity. Request-response invocation requires both an inputVariable (or its equivalent *toPart* elements) and an outputVariable (or its equivalent *fromPart* elements). So its memory variables may be updated. For the loan approval, the complete form of process function ϕ_3 and ϕ_5 are as follows: ϕ_3: *(m, creditInformationMessageV)* $\rightarrow$ *(risk-AssessmentMessageV, m_1)*, Where, *creditInformation-MessageV* is one value of the *creditInformation-Message* type which is as the input message; *risk-AssessmentMessageV* is one value of the *risk-*

AssessmentMessage type which is as the output message; we assume that m = {*CreditInformationMessageV, ValueOne, ValueTwo*}, then m_1 = {*CreditInformation-MessageV,riskAssessmentMessageV,ValueTwo*}. ϕ_5:   *(m,   creditInformation-MessageV)$\rightarrow$ (approval-MessageV, m_1)* , Where, *creditInformationMessageV* is one value of the *creditInformationMessage* type which is as the input message; *approvalMessageV* is one value of the *approvalMessage* type which is as the output message; we assume that *m* = {*CreditInformationMessageV, ValueOne, ValueTwo*}, then m_1 = {*CreditInformationMessageV, valueOne, approvalMessageV*}.

(2) Assign processing functions

The *assign* activity can be used to copy data from one variable to another, as well as to construct and insert new data using expressions. The memory variables are to be updated. In our case, the *assign* activity is as **assign processing function**. The variable in the *from-spec* of the *assign* activity is as its input. The variable in the *to-spec* of the *assign* activity is as its output. The memory updates are obtained from the *copy element*. For the loan approval, assign processing function has ϕ_7(*assign_approval*): *(m, copyValue)$\rightarrow$(approval, m_1)*, where *copyValue*={*value*} We assume that *m* ={*ValueOne, ValueTwo, ValueThree*}, then m_1={*ValueOne, ValueTwo, yes*}.

(3) Predicate processing functions

In the paper, we introduce auxiliary predicate processing functions for conditional expression in *while* activity, in *repeatUntil* activity, join Condition in *targets* element of some activities in the *flow* activity and in *onAlarm* event in *pick* activity. For the predicate processing function, its input is a variable which encapsulates the corresponding conditional expression and its output is *true* or *false*. The memory variables are not updated. Below we provide the definitions for the predicate process functions of the loan approval.

- ϕ_2(*invoke_check_jct*): *(m, invoke_check_jc)$\rightarrow$(true, m)*,
- ϕ_2' (*invoke_check_jcf*): *(m, invoke_check_jc)$\rightarrow$(false, m)*,
- ϕ_4(*invoke_approve_jct*): *(m, invoke_approve_jc)$\rightarrow$(true, m)*,
- ϕ_4' (*invoke_approve_jcf*): *(m, invoke_approve_jc)$\rightarrow$(false, m)*,
- ϕ_6(*assign_approval_jct*): *(m, assign_approval_jc)$\rightarrow$(true, m)*,
- ϕ_6' (*assign_approval_jcf*) : *(m, assign_approval_jc)$\rightarrow$(false, m)*,

For the loan approval process, the input set Σ={*creditInformationMessageType, copyValue, invoke_check_jc, invoke_approve_jc, assign_approval_jc*, $\perp$}, and the output set Γ={ *approvalMessage , riskAssessmentMessage*}.

4   Automatic Generation of Test cases

For the DSXM for BPEL process which is obtained in section 3, we can automatically generate test cases for BPEL with the testing strategy based on DSXM [8]. It is proved to find all faults in the implementation. In addition, the method requires that the DSXM models satisfy the design for test conditions, i.e. they are completely defined and output-distinguishable.

A DSXM may be transformed into one that is completely-defined by assuming that "refused" inputs produce a designated error output, which is not in the output alphabet of Z; this behavior can be represented as self-looping transitions or transitions to an extra (error) state. Let $\sigma1=creditInformationMessageType$, $\sigma2= invoke\_check\_jc$, $\sigma3= invoke\_approve\_jc$, $\sigma4= assign\_approval\_jc$, $\sigma5= copyValue$, $\sigma6= \perp$. In the loan approval process DSXM model, the erroneous behavior can be represented by two additional processing functions, $error_{\sigma1}$, $error_{\sigma2}$, $error_{\sigma3}$, $error_{\sigma4}$, $error_{\sigma5}$ and $error_{\sigma6}$ that take inputs from $creditInformationMessageType$, $\{yes\}$, $invoke\_check\_jc$, $invoke\_approve\_jc$, $assign\_approval\_jc$ and $\{\perp\}$ respectively, which will label appropriate self-looping transitions. That is, the state-transition diagram of the completely-defined DSXM will contain the following (extra) self-looping transitions:

- $error_{\sigma2}$, $error_{\sigma3}$, $error_{\sigma4}$, $error_{\sigma5}$, $error_{\sigma6}$ in state q_0,
- $error_{\sigma1}$, $error_{\sigma3}$, $error_{\sigma4}$, $error_{\sigma5}$, $error_{\sigma6}$ in state q_1,
- $error_{\sigma2}$, $error_{\sigma3}$, $error_{\sigma4}$, $error_{\sigma5}$, $error_{\sigma6}$ in state q_2,
- $error_{\sigma1}$, $error_{\sigma2}$, $error_{\sigma5}$, $error_{\sigma6}$ in state q_3,
- $error_{\sigma2}$, $error_{\sigma3}$, $error_{\sigma4}$, $error_{\sigma5}$, $error_{\sigma6}$ in state q_4,
- $error_{\sigma1}$, $error_{\sigma2}$, $error_{\sigma3}$, $error_{\sigma6}$ in state q_5,
- $error_{\sigma1}$, $error_{\sigma2}$, $error_{\sigma3}$, $error_{\sigma4}$, $error_{\sigma6}$ in state q_6,
- $error_{\sigma1}$, $error_{\sigma2}$, $error_{\sigma3}$, $error_{\sigma4}$, $error_{\sigma5}$ in state q_7,
- $error_{\sigma1}$, $error_{\sigma2}$, $error_{\sigma3}$, $error_{\sigma4}$, $error_{\sigma6}$ in state q_8,
- $error_{\sigma1}$, $error_{\sigma2}$, $error_{\sigma4}$, $error_{\sigma5}$, $error_{\sigma6}$ in state q_9,
- $error_{\sigma2}$, $error_{\sigma3}$, $error_{\sigma4}$, $error_{\sigma5}$, $error_{\sigma6}$ in state q_{10},
- $error_{\sigma1}$, $error_{\sigma2}$, $error_{\sigma3}$, $error_{\sigma4}$, $error_{\sigma5}$, $error_{\sigma6}$ in state q_{11}.

In the loan approval process DSXM model, Φ is output-distinguishable. The method in [8] may be employed to produce the test path set $U = \bigcup_{q \in Q_r} \{p_q\} pref(V(q)) W_s$. Then all we need to do is to translate each such sequence path into an appropriate input sequence to check whether it has been implemented correctly or not. The testing process can therefore be performed automatically by checking whether the output sequence produced by implementation is identical with the ones expected from the SXM model.

The execution of BPEL process is semiautomatic. When the BPEL process runs, test data generally need to be provided for the *receive* and *pick* activity, to trigger the execution of the succeeding activity. However, some *receive* and *pick* activities need no test data, and the *assign* activities will provide the input data for them. In order to automatically obtain the test data for test paths, we present the following solution.

For the *receive* or *pick* activity in BPEL process, if it is the succeeding activity of the *assign* activity, its test data need not to be provided. Otherwise, the test data of test the corresponding operations for an atomic web service will be used as test data of triggering the *receive* or *pick* activity. In order to judge whether the *receive* or *pick* activity is the succeeding activity of the *assign* activity or not, we could build a symbol table for all *assign* activities in BPEL process which store their succeeding activities. For instance, $p = \{\phi_1, \phi_2, \phi_3, \phi_6, \phi_7, \phi_4', \phi_8\}$ is a test path in the loan approval SXM. For p, we only need to provide test data for ϕ_1. Input data invoking other processing function will be obtained by the current memory.

For n'=12, the S_r, Q_r and W_S in the loan approval process DSXM are:

$S_r = \{\varepsilon, \phi_1, \ \phi_1\phi_2, \ \phi_1\phi_2\phi_3, \ \phi_1 \phi_2' \ \phi_4, \ \phi_1 \ \phi_2' \ \phi_4\phi_5, \ \phi_1\phi_2\phi_3 \ \phi_4' \ \phi_6, \ \phi_1\phi_2\phi_3\phi_6, \ \phi_1\phi_2\phi_3 \ \phi_4' \ \phi_6$, ϕ_1

$\phi_2 \ \phi_3 \ \phi_6 \ \phi_7 , \ \phi_1 \ \phi_2 \ \phi_3 \ \ \phi_6' \ \phi_4 , \ \phi_1 \ \phi_2 \ \phi_3 \ \ \phi_6' \ \ \phi_4 \ \phi_5\phi_8\}$

$Q_r = \{ q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}, q_{11}\}$

$W_S = \{\phi_1, \ \phi_2, \ \phi_3, \ \phi_5, \phi_7, \ \phi_8, \ \phi_4, \phi_4', \ \phi_6, \phi_6', \ error_{\sigma1} , \ error_{\sigma2}, error_{\sigma3} , \ error_{\sigma4} , \ error_{\sigma5} ,$

$error_{\sigma6}\}$

The derived U and t(U) are the following by employing the method in [8].

$U = \{ <\sigma1>, <\sigma1, \sigma2>, \dots, <\phi_1, \phi_2, \phi_3, \phi_6, \phi_7, \ \phi_4', \phi_8>, <\phi_1, \phi_2, \phi_3, \ \phi_6', \phi_4, \phi_5, \phi_8>, \dots\}$

$t(U) = \{ <\sigma1>, <\sigma1, \sigma2>, \dots <\sigma1, \sigma2, \sigma1, \sigma4, \sigma5, \sigma3, \sigma6>, <\sigma1, \sigma2, \sigma1, \sigma4, \sigma3, \sigma1,$

$\sigma6>, \dots\}$

5   Conclusion

In this paper we have presented an approach to automatically generate test cases with a formal method SXM for BPEL process. It is new to apply such a technique in the domain of testing BPEL process. To formalize BPEL process using SXM, we present the transitions details from BPEL process to SXM. The automatization of the transformation process could be implemented. We are currently working on the automated tool to support our approach. The tool supports to translate BPEL code to DSXM, and automatically generates test suite based on the test strategy in [8].   In the future, we will improve our method through a large number of experiments.

References

1. Web Services Business Process Execution Language Version 2.0 OASIS Standard, 11 April 2007 , http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf.
2. P.Mayer and D.Lübke.: Towards a BPEL unit testing framework. In TAV-WEB '06: Proceedings of Workshop on Testing, analysis, and verification of web services and applications, pages 33–42. ACM Press, 2006.
3. Y. Yuan, Z. Li, and W. Sun.: A graph-search based approach to bpel4ws test generation. In Proceedings of the International Conference on Software Engineering Advances (ICSEA 2006), page 14. IEEE Computer Society, 2006.
4. Jun Yan, Zhongjie Li, Yuan Yuan, Wei Sun and Jian Zhang.: BPEL4WS Unit Testing: Test Case Generation Using a Concurrent Path Analysis Approach. 17th International Symposium on Software Reliability Engineering(ISSRE'06), pages 75-84, 2006.
5. Y. Zheng, J. Zhou, and P. Krause.: A model checking based test case generation framework for web services. In Proceedings of the International Conference on Information Technology (ITNG'07), pages 715–722. IEEE Computer Society, 2007.
6. M. Holcombe and F. Ipate.: Correct Systems: Building a Business Process Solution. Springer Verlag, Berlin, 1998.
7. T. S. Chow.: Testing software design modeled by finite state machines. IEEE Transactions on Software Engineering, 4:178－187, 1978.
8. Florentin Ipate, Mike Holcombe.: Testing data processing-oriented systems from stream X-machine models, Theoretical Computer Science403(2008) Pages176-191.