Automatic Test Case Generation for BPEL Using Stream X-Machine

Chunyan Ma1,2, Junsheng Wu1, Tao Zhang1, Yunpeng Zhang1, Xiaobin Cai2
1College of Software and Microelectronics, Northwestern Polytechnical University, Xi'an 710072, China
2Institute 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 $Z=(\Sigma, \Gamma, Q, M, \Phi, F, q_0, m_0)$ [8], where

- $\Sigma$ and $\Gamma$ 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.
- $\Phi$ is a finite set of distinct processing functions; a process function is a non-empty (partial) function of type $M \times \Sigma \rightarrow \Gamma \times M$.
- $F$ 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

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 \( a_1, a_2, \ldots, a_m \) 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\_condition, 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\_condition, 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.

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 a₁, a₂, … aₘ denote all sub-activities of the flow activity, then the number of their alternant occurrence sequences is \( Pₘ \). We denote the activity sequences set of the number of \( Pₘ \) as S.

**Step3.** Compute all order pairs set C. For the activity pair \( aᵦaᵦ \), if a link exists, and \( aᵦ \) as source, \( aᵦ \) as its target, we call \( aᵦaᵦ \) an order pair. The order pair presents the synchronization dependency between activities.
Step4. \( \forall s \in S \), 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 \( S_1 \). Let \( S_2 = S - S_1 \).

Step5. \( \forall s \in S_2 \), Assume \( s = b_1 b_2 \cdots b_m \) and their SXM fragments are \( M_1, M_2, \ldots, M_m \) 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 \( b_kb_{k+1} (1 \leq k \leq m) \) 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_f1 and targetName_f2 respectively. The transition whose processing function targetName_f1 returns true leads to the starting state of \( M_{k+1} \). The transition whose processing function targetName_f2 returns false leads to the ending state of \( M_k \) (namely dead-path-elimination (DPE)).

Step6. Assume \( S_2 = \{ s_1, s_2, \ldots, s_j \} \), and \( s_1, s_2, \ldots, s_j \) corresponding SXM fragments built by the step 5 are \( M_{s_1}, M_{s_2}, \ldots, M_{s_j} \) respectively. We combine \( M_{s_1}, M_{s_2}, \ldots, M_{s_j} \) into one SXM fragment \( M \) with \( \varepsilon \). We introduce two new states \( q_1 \) and \( q_2 \) which are as the starting state and the ending state of \( M \) respectively. The transitions of \( q_1 \) lead to the starting state of \( M_{s_1}, M_{s_2}, \ldots, M_{s_j} \) respectively. Their ending states lead to \( q_2 \) respectively by \( \varepsilon \) 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 \( M = \{ \text{request} , \text{risk}, \text{approval} \} \). Since memory has to be filled with specific values, the initial memory value \( m_0 = \{ \text{null} , \text{null}, \text{null} \} \).
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 $\phi_1$ (receive_request), $\phi_3$ (invoke_check), $\phi_5$ (invoke_approve), and $\phi_9$ (reply-request). Receive request denotes the name of $\phi_1$. 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 $\phi_1$ is as follows:

$$\phi_1: (m, \text{CreditInformationMessageV} ) \rightarrow (\bot, m_1),$$

Where, $\text{CreditInformationMessageV}$ is one value of the $\text{creditInformationMessage}$ type which is as the input message; we assume that $m = \{\text{valueOne}, \text{valueTwo}, \text{valueThree}\}$, then $m_1 = \{\text{CreditInformationMessageV}, \text{valueTwo}, \text{valueThree}\}$. 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 $\phi_9$ is as follows:

$$\phi_9: (m, \bot) \rightarrow (\text{approvalMessageV}, m),$$

Where $\text{approvalMessageV}$ is one value of the $\text{approval-Message}$ 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 $\phi_3$ and $\phi_5$ are as follows:

$$\phi_3: (m, \text{creditInformationMessageV}) \rightarrow (\text{risk-AssessmentMessageV}, m_1),$$

Where, $\text{creditInformation-MessageV}$ is one value of the $\text{creditInformation-Message}$ type which is as the input message; $\text{risk-AssessmentMessageV}$ is one value of the $\text{risk-
AssessmentMessage type which is as the output message; we assume that \( m = \{ \text{CreditInformationMessageV, ValueOne, ValueTwo} \} \), then \( m_1 = \{ \text{CreditInformationMessageV, riskAssessmentMessageV, ValueTwo} \} \). \( \phi_5: (m, \text{creditInformationMessageV}) \rightarrow (\text{approvalMessageV}, m_1) \). Where, \( \text{creditInformationMessageV} \) is one value of the \( \text{creditInformationMessage} \) type which is as the input message; \( \text{approvalMessageV} \) is one value of the \( \text{approvalMessage} \) type which is as the output message; we assume that \( m = \{ \text{CreditInformationMessageV, ValueOne, ValueTwo} \} \), then \( m_1 = \{ \text{CreditInformationMessageV, valueOne, approvalMessageV} \} \).

(2) Assign processing functions

The \( \text{assign} \) 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 \( \text{assign} \) activity is as \( \text{assign processing function} \). The variable in the \( \text{from-spec} \) of the \( \text{assign} \) activity is as its input. The variable in the \( \text{to-spec} \) of the \( \text{assign} \) activity is as its output. The memory updates are obtained from the \( \text{copy} \) element. For the loan approval, assign processing function has \( \phi_7(\text{assign approval}) : (m, \text{copyValue}) \rightarrow (\text{approval, m}_1) \), where \( \text{copyValue} = \{\text{value}\} \). We assume that \( m = \{\text{ValueOne, ValueTwo, ValueThree}\} \), then \( m_1 = \{\text{ValueOne, ValueTwo, valueOne, approvalMessageV}\} \).

(3) Predicate processing functions

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

- \( \phi_2(\text{invoke_check_jct}) : (m, \text{invoke_check_jc}) \rightarrow (\text{true, m}) \),
- \( \phi_2(\text{invoke_check_jcf}) : (m, \text{invoke_check_jc}) \rightarrow (\text{false, m}) \),
- \( \phi_4(\text{invoke_approve_jct}) : (m, \text{invoke_approve_jc}) \rightarrow (\text{true, m}) \),
- \( \phi_4(\text{invoke_approve_jcf}) : (m, \text{invoke_approve_jc}) \rightarrow (\text{false, m}) \),
- \( \phi_6(\text{assign Approval_jct}) : (m, \text{assign_approval_jc}) \rightarrow (\text{true, m}) \),
- \( \phi_6(\text{assign Approval_jcf}) : (m, \text{assign_approval_jc}) \rightarrow (\text{false, m}) \).

For the loan approval process, the input set \( \Sigma = \{ \text{creditInformationMessageType, copyValue, invoke_check_jc, invoke_approve_jc, assign Approval_jc, } \perp \} \), and the output set \( \Gamma = \{ \text{approvalMessage, riskAssessmentMessage} \} \).

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 σ1 = creditInformationMessageType, σ2 = invoke_check_jc, σ3 = invoke_approve_jc, σ4 = assign_approval_jc, σ5 = copyValue, σ6 = ⊥. In the loan
approval process DSXM model, the erroneous behavior can be represented by two
additional processing functions, errorσ1, errorσ2, errorσ3, errorσ4, errorσ5 and errorσ6
that take inputs from creditInformationMessageType, {yes}, invoke_check_jc,
invoke_approve_jc, assign_approval_jc and {⊥} 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:

- errorσ2, errorσ3, errorσ4, errorσ5, errorσ6 in state q8,
- errorσ1, errorσ2, errorσ3, errorσ4, errorσ5, errorσ6 in state q1,
- errorσ2, errorσ3, errorσ4, errorσ5, errorσ6 in state q2,
- errorσ1, errorσ2, errorσ3, errorσ5, errorσ6 in state q3,
- errorσ2, errorσ3, errorσ4, errorσ5, errorσ6 in state q4,
- errorσ1, errorσ2, errorσ3, errorσ6 in state q5,
- errorσ1, errorσ2, errorσ3, errorσ4, errorσ6 in state q6,
- errorσ1, errorσ2, errorσ3, errorσ4, errorσ5 in state q7,
- errorσ1, errorσ2, errorσ3, errorσ4, errorσ6 in state q8,
- errorσ2, errorσ3, errorσ4, errorσ5, errorσ6 in state q9,
- errorσ2, errorσ3, errorσ4, errorσ5, errorσ6 in state q10,
- errorσ2, errorσ3, errorσ4, errorσ5, errorσ6 in state q11.

In the loan approval process DSXM model, Φ is output-distinguishable. The
method in [8] may be employed to produce the test path set

\[ U = \bigcup_{q \in Q} \{ p \} \text{pref}(V(q))W_s \]

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, \( p = \{ \phi_1, \phi_2, \phi_3, \phi_4, \phi_5 \} \) is a test path in the loan approval
SXM. For \( p \), we only need to provide test data for \( \phi_i \). 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:

$$S_r = \{ \varepsilon, \phi_1, \phi_2, \phi_3, \phi_4, \phi_5, \phi_6, \phi_7, \phi_8, \phi_9, \phi_10, \phi_11, \phi_12 \}$$

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

$$W_S = \{ \phi_1, \phi_2, \phi_3, \phi_4, \phi_5, \phi_6, \phi_7, \phi_8 \}$$

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

$$U = \{ <\sigma_1>, <\sigma_1, \sigma_2>, \ldots, <\phi_1, \phi_2, \phi_3, \phi_4, \phi_5, \phi_6, \phi_7, \phi_8>, \ldots \}$$

$$t(U) = \{ <\sigma_1>, <\sigma_1, \sigma_2>, \ldots, <\sigma_1, \sigma_2, \sigma_1, \sigma_4, \sigma_5, \sigma_6>, <\sigma_1, \sigma_2, \sigma_1, \sigma_4, \sigma_3, \sigma_1, \sigma_6>, \ldots \}$$

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
