RC24028 (C0608-007) August 16, 2006
Computer Science

IBM Research Report

A Graph-search Based Approach to BPEL4WS Test Generation

Yuan Yuan, Zhongjie Li, Wei Sun
IBM Research Division
China Research Laboratory
HaoHai Building, No. 7, 5th Street
ShangDi, Beijing 100085
China

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g, payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home.

A Graph-search Based Approach to BPEL4WS Test Generation

Yuan Yuan, Zhongjie Li, Wei Sun
China Research Lab
IBM
Beijing, P.R.C.
{yyuan, lizhongj, weisun}@cn.ibm.com

***Abstract*—Business Process Execution Language for Web Services (BPEL4WS) is a kind of concurrent programming languages with several special features that raise special challenges for verification and testing. This paper proposes a graph-search based approach to BPEL test case generation, which effectively deals with BPEL concurrency semantics. This approach defines an extension of CFG (Control Flow Graph) - BPEL Flow Graph (BFG) - to represent a BPEL program in a graphical model. Then concurrent test paths can be generated by traversing the BFG model, and test data for each path can be generated using a constraint solving method. Finally test paths and data are combined into complete test cases.**

Keywords-BPEL; test generation; test path generation; graph-search

I. Introduction

SOA (Service Oriented Architecture) is being adopted in various industries. Business Process Execution Language for Web Services [1] (BPEL4WS, abbr. BPEL) is a key element of the SOA protocol stack, and offers a standards-based approach to build flexible business processes by choreographing multiple Web Services.

In today's software development lifecycle, testing remains largely a manual work whose effectiveness and efficiency mainly depend on human skills and experience. Among all the testing related activities, test planning and test case design are most time-consuming because they usually have little automation support. It's no exception in the BPEL case. To test a BPEL program thoroughly, we need to cover different execution scenarios. Human test case design is tedious and difficult, especially for large and complex BPEL processes. So it is highly desirable to introduce automation support in BPEL test case design. Automatic test generation is the right technology to fulfill this requirement by searching execution paths, generating required test data and combining them into test cases in a systematic and highly-automated way.

Test generation research for concurrent programs has a long history. BPEL has constructs that express concurrency and synchronization, thus can be regarded as a kind of concurrent program. In the area of testing concurrent programs, many existing research works are based on reachability analysis. A common method is to construct a reachability graph (RG) of the program under test [2]. This method is limited in practice due to state space explosion problem. To overcome this deficiency, our approach will not construct a RG and enumerate all the serialized paths. Instead, we only count concurrent path that represents a partial order list of BPEL activities.

Some test generation methods based on path analysis are also proposed for testing sequential and concurrent [3][4] programs. These methods firstly select local paths for individual tasks, then compose global paths with these local paths. They are applicable to programs consisting of communication processes or tasks, like those written in Ada or CSP, but inappropriate for BPEL, which has neither explicit separation of individual processes nor synchronization via rendezvous.

Furthermore, BPEL has unique features in both syntax (e.g. flow with activity synchronization, join condition) and semantics (e.g. dead-path-elimination) that need special treatment.

This paper proposes a graph-search based approach to BPEL test case generation, which effectively deals with BPEL special features. This approach defines an extension of CFG (Control Flow Graph) - BPEL Flow Graph (BFG) - to represent a BPEL program in a graphical model. Then concurrent test paths can be generated by traversing the BFG model, and test data for each path can be generated using a constraint solving method.

This paper is organized as follows. Some BPEL basics and features will be introduced in the next Section, then the proposed test generation method is elaborated in Section 3. Some complexity analysis is shown in Section 4. Section 5 follows with related works. Section 6 concludes the paper with future work predictions.

II. BPEL Language

A. BPEL Basics and Features

Like any programming language, BPEL has typical control structures including sequence, switch, while, etc. In addition, BPEL uses the flow construct to provide concurrency and synchronization. Synchronization and concurrency among activities are provided by means of links. Each link can have a source activity and a target activity. Furthermore, a transition

condition, which is a Boolean expression, is associated with each link and is evaluated when the source activity terminates. As long as the transition condition of a link has not been evaluated, the value of the link is undefined. Each activity of a flow has a join condition, which consists of incoming links of the activity combined by Boolean operators. Only when all the values of its incoming links are defined and its join condition evaluates to True an activity is enabled and can start. Otherwise if the join condition evaluates to False, the activity will not be executed and this effect will be propagated downstream to subsequent activities, so-called Dead Path Elimination (DPE).

Due to the DPE semantics, when a BPEL program is run, there can be three different states for an activity or link: executed, un-executed, and dead path, which are shown in Fig. 1.

With flow and link constructs, BPEL can easily support a workflow pattern named "multiple-choice" [5]. This pattern specifies that from an activity, there can be multiple outgoing flows enabled simultaneously, which represents a concurrent behavior. Which and how many outgoing flows are enabled depend on the transition conditions of the outgoing links. So if an activity has n outgoing links, there are at most 2^n possible execution scenarios.

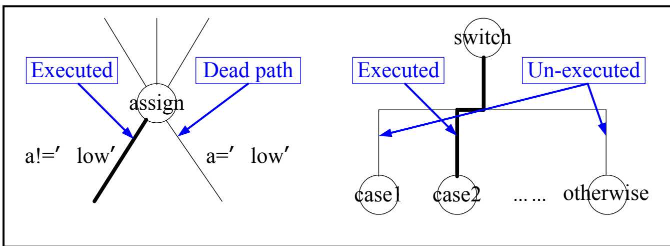

Figure 1. Three states of BPEL control logic

B. BPEL Example

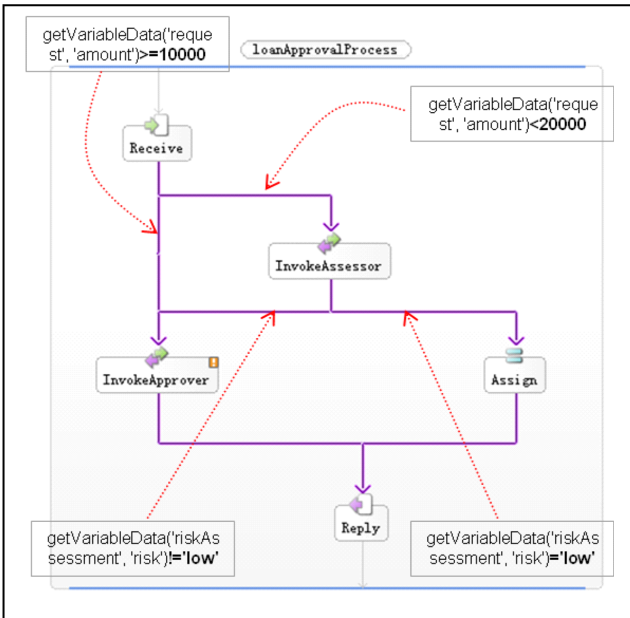

Figure 2. Loan approval process

Fig. 2 is an example of BPEL program describing a loan approval process. This process begins by receiving a loan request. For low amounts (less than $20,000) and low-risk individuals, approval is automatic. For high amounts or high-risk individuals, each credit request needs to be studied in more detail. The use of risk assessment and loan approval services is represented by invoke elements, which are contained within a flow, and their (potentially concurrent) behavior is staged according to the dependencies expressed by corresponding link elements. Note that the transition conditions attached to the links determine which links get activated, all the join conditions use default setting. Finally the process responds with either a "loan approved" message or a "loan rejected" message. Note that this example is adapted from the original loan approval process in the BPEL specification [1]. A transition condition is modified from getVariableData('request', 'amount')<10000 to getVariableData('request', 'amount')<20000 to demonstrate current test path generation method in dealing with multiple-choice workflow pattern.

III. BPEL TEST GENERATION

In this Section, we'll elaborate the proposed BPEL test generation method. This method contains four steps as illustrated in Fig. 3.

Step1: Transform the BPEL program to an intermediary model - BFG.

This step is critical because BPEL is a highly compact and expressive language that is hard to analyze. BFG can be seen as a "partially executed" representation of BPEL and functions in the same way as CFG does in traditional sequential program testing. Compared to the BPEL program, BFG has the following differences to facilitate test path searching. Firstly, it unravels the folded structures of BPEL (e.g. while loop, dead path elimination) into unfolded structures that are directly traversable in graph searching. Secondly, it turns implicit, disjoint control flows (e.g. exception handling) into explicit, connected control flows. Thirdly, it reduces the quantity of control structure types - some control structures are represented uniformly according to their similar semantics (e.g. switch and pick both express branching control flow, thus will share the same notation in BFG). BPEL can be viewed as a graph composed of nodes (activities) and edges (implicit sequence concatenations, links), and BFG is also defined as a graph. In the transformation, each activity or link in BPEL is transformed to a corresponding node in BFG or a sub-graph composed of a set of normal nodes, control nodes and edges.

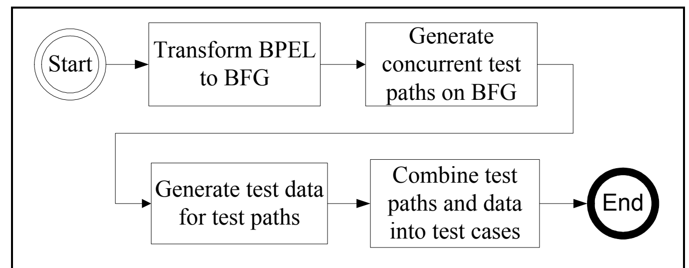

Figure 3. Process of the proposed test generation method

Step2: Traverse the BFG to generate test paths.

It is easy to find a set of test paths on BFG to satisfy a specific coverage goal, such as branch coverage and some user-specified coverage goals. A BPEL test path is a partially-ordered list of basic activities that are executed during a specific test run. BPEL activities in a test path are classified into three types: one is "input-type" (receive, the receiving direction of 2-way invoke, etc); another is "output-type" (the sending direction of 2-way invoke, reply, 1-way invoke, etc); the third is "dataHandling-type" (assignment, etc).

Step3: Filter infeasible test paths, and generate test data for feasible test paths.

After test paths are found, test data should be generated to make the test path executable. Not all the test paths found in Step 2 are feasible, however. A test path is feasible if there exists at least one set of test data that satisfies all the conditions (described as Boolean expressions) in this path. We can use constraint solving (CS) tools [7] to solve the formulas of inequalities and equalities derived from each test path. If no solution is found, the path is infeasible and filtered; otherwise the path is feasible, and a solution is also at hand. Typically the data constraints and the solution only cover part of the required data. The rest can be derived using random data generation tools automatically or by human manually.

Step4: Generate abstract test cases by combining test data and paths.

To generate a test case, the "dataHandling-type" will be removed, as it is useless in test case execution, while both "input-type" and "output-type" logic will remain to be populated with real test data. Part of the input-type data can be automatically generated based on the constraint set. However, output-type logic should be manually prepared in a form of either exact data values or invariable assertions, which are commonly called "test oracles".

Note that the resulted test cases are abstract, in a sense that they are not directly executable. They must be transformed to a program written in a specific programming language to be able to execute in some testing environment, where the behavior of the program under test is verified, and unexpected behaviors are reported to the user. Therefore, our abstract test case is a kind of platform independent model of the tests.

A. BFG Definition

BFG is proposed as an intermediary model to facilitate the test generation work. It is an extension of Control Flow Graph and adds some concurrency related syntax. It contains not only structural information, which specifies all control flow information of a BPEL program and enough data flow information for test data generation, but also semantic information such as dead paths.

The structural definition of BFG is as follows:

BFG=<N, E, s, F>, where

N is a set of nodes,

E is a set of edges,

s is the start node, and

F is a set of final nodes.

$N=\{n_i\}$, $1<=i<=p$, p is the number of BFG nodes, where

$n_1=s$, $n_i\in\{NN, DN, MN, FN, JN\}$ where the meaning of NN, DN, MN, FN, JN is shown in Table 1.

$E=\{e_j\}$, $1<=j<=q$, q is the number of BFG edges, where

$e_j=<a, b>$, $a, b\in N$, $e_j\in\{TE, FE\}$ where the meaning of TE, FE is shown in the Table 1.

The extension of BFG to CFG is both on syntax (e.g. FN, JN) and semantics (e.g. FE/DP) to express concurrency and dead path elimination, which is explained as follows.

If the condition associated with an edge is evaluated False, which means a dead path, the False value should be propagated downstream till a join node. The condition of the outgoing edges of the join node will be evaluated when all the status of incoming edges have been determined.

B. Transform Multiple-choice to Exclusive-choice

1) Transform multiple-choice to all conditions of exclusive-choice.

To facilitate test path searching, we transform the multiple-choice pattern in BPEL to exclusive-choice structure in BFG. Suppose an activity has n outgoing links, the following transformation is performed one by one.

① We add one decision node, 2^n fork nodes, and n merge nodes between the considered activity and the target activities of these links.

② An edge is added between the decision node and each of the fork nodes, representing all the possible execution scenarios.

③ Each merge node will be connected to a target activity, representing that multiple incoming edges merge here and then connect to the target activity.

④ Between the fork nodes and the merge nodes, edges are added to reflect the 2^n execution scenarios. For each scenario, we may add two types of edges. If a link is enabled in this scenario, we add a True edge between the fork node and the merge node leading to the target activity of the enabled link. If a link is disabled in this scenario, we add a False edge between the fork node and the merge node leading to the target activity of the disabled link. Associated with each scenario there will be a condition, which is the conjunction of the following two expressions: transition conditions of the enabled links, the negation of the transition conditions of the disabled links.

Fig. 5 gives a pictorial representation of this transformation for the multiple-choice pattern in the loan approval process example.

TABLE I. DEFINITION AND DENOTATION OF BFG

Name	Node and edge description		
	Abb.	*Definition*	*Legend*
Normal Node	NN	The node with only one incoming and only one outgoing edge.	○
Decision Node	DN	The node with one incoming edge and multiple exclusive outgoing edges.	◇
Merge Node	MN	The node with multiple incoming edges and one outgoing edge. Any incoming edge will enable the outgoing edge.	◆
Fork Node	FN	The node with one incoming edge and multiple concurrent outgoing edges.	▭
Join Node	JN	The node with multiple concurrent incoming edges and one outgoing edge. Only when all the incoming edges have been arrived, can the outgoing edge starts.	▬
True Edge	TE	Edge with True condition value.	⟶
False Edge/Dead Path	FE/DP	Edge with False condition value. Also called dead path (DP).	– – →

2) Remove invalid scenarios.

In most cases, there are not so many scenarios because most of the scenario conditions can not hold, for example, getVariableData('request', 'amount')<10000 && getVariableData('request', 'amount')>=20000. If we can decide in BFG construction time whether the condition is satisfy-able or not, the generated scenarios can be reduced greatly.

We can use some simple approaches, e.g. interval analysis, to identify the feasible scenarios. For the example in Fig. 2, the value domain of "amount" can be divided into three intervals: (0, 10000), [10000, 20000), [20000, INFINITE). Thus there are only three feasible scenarios, which are shown in BFG denotation in Fig. 4. For more complex conditions, we can use the constraint solving method.

C. Transform BPEL to BFG

There are two kinds of control flow representations in BPEL: One is explicit control flow defined by links and the other is implicit one defined in sequence. To facilitate the unified handling of these two control flows, we first replace all the "sequence" structures with "flow" structures wherein the links have TRUE transition conditions. Then we map BPEL to BFG structures, which include basic activities (receive, reply, invoke, assign, throw, terminate, wait, empty) and structural activities (sequence, flow, switch, while, pick, etc). The following mapping rules are referenced in the transformation.

Rule 1. BPEL basic activities are mapped to BFG nodes.

Rule 2. BPEL structural activities are mapped to BFG nodes and edges. Depending on activity types, several sub rules are needed.

Rule 2.1. Switch and pick. Map the <switch>-</switch> pair to a <DN>-<MN> pair (the definition of <DN><MN> can be referred to Table 1). The target links of the switch activity are mapped to edges that connect to the decision node, and edges are also added to connect the decision node and the nodes mapped from "case" branches of the switch activity. Similar mapping applies to the pick activity.

Rule 2.2. While. For loop control flow that may repeat many times, it is common practice to assume a 0-1 criterion in test case generation, where only two samples are used. One is zero repetition, which corresponds to no-execution of the contained activity, and the other is one repetition, which corresponds to one execution of the contained activity. Similarly, the <while>-</while> pair is mapped to a <DN>-<MN> pair. One outgoing edge of the decision node connects to the merge node directly, and the other connects to the node mapped from the contained activity.

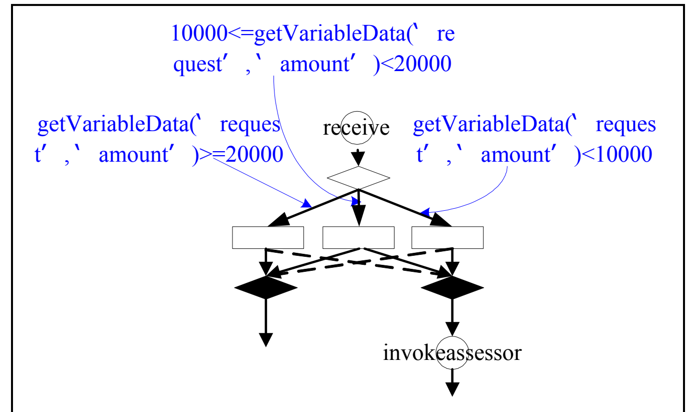

Figure 4. Exclusive-choice example in BFG

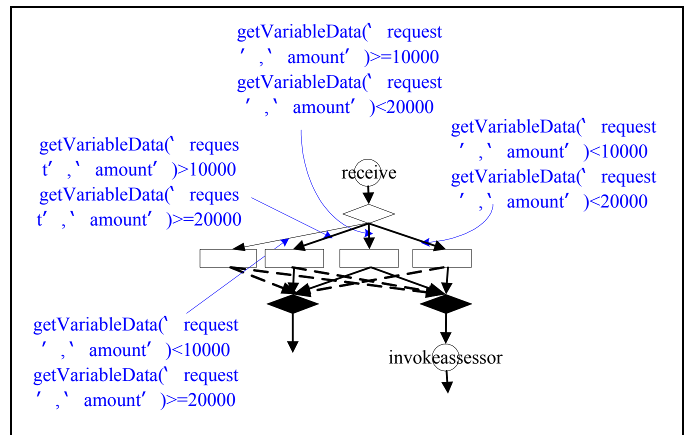

Figure 5. A general transformation of BPEL multiple-choice in BFG

Rule 3. Flow. The handling of "flow" is the most complex. The <flow>-</flow> pair is mapped to a <FN>-<JN> pair. Besides this mapping, there are also similar mapping for activities inside a "flow". For any activity, let m denote the number of its target links, and n denote the number of its source links. If m>1, insert a join node between the edges mapped from the target links and the node mapped from the considered activity. If n>1, insert a fork node between the node mapped from the considered activity and the edges mapped from the source links. Then, we have to process the outgoing edges of every fork node to transform multiple-choice structure to exclusive-choice structure, as has been described in Section 3.2. Finally, for an edge with a transition condition that may evaluate to either True or False, it should be further transformed to two edges. One is with transition condition of True, the other is with transition condition of False. Fig. 6 shows the BFG transformed from the loan approval process example.

D. *BFG Traversing Method for Test Path Searching*

The path searching method on BFG can be based on Depth First Search (DFS) algorithm. It scans the BFG and processes each node and edge. There are four kinds of processing for different node types to find paths based on branch coverage: (1) for normal node or merge node, it adds the node to the current path; (2) for decision node, it duplicates the current path to n paths (n is the number of outgoing edges from the decision node), and adds the following n nodes to the n paths respectively; (3) for fork node, it searches n sub-paths (n is the number of outgoing edges from the fork node) and adds all the sub-paths to the current path; (4) for join node, it awaits all the incoming edges to be processed, and then evaluates the join node based on the join condition.

The path searching algorithm can be implemented in a matrix transformation method, which contains five steps:

Step1: Mark each node in BFG with a unique number. The labeled BFG is shown in Fig. 6.

Step2: Record the preconditions and post-conditions of each node, and the relation among these conditions. For example, table 2 shows some of the preconditions and post-conditions and their relation for Fig. 6.

Step3: Determine the start node and the final nodes. The node whose precondition is null is the start node, while whose post-condition is null are the final nodes. To find a path on BFG is to find a path from start node to any of the final nodes.

Step4: Find paths by matrix transformation. The row and column index of the matrix represents BFG node number, and the element in matrix[i][j] represents the BFG edge $<N_i, N_j>$. An element takes its value from the space of {-1, 0, 1}, where 0 denotes that the edge is not been visited currently, 1 denotes that a normal edge is visited, and -1 denotes that a dead edge is visited or that the edge is on a dead path. All elements in the matrix have an initial value 0. The path searching begins with processing the outgoing edges of the start node, i.e. the post-conditions of the start node. The searching process continues by finding elements with value 1 or -1, for example

TABLE II. BFG CONDITION AND RELATION INFORMATION

Node Id	Conditions and relation			
	Preconditions	***Relation among preconditions***	***Post-conditions***	***Relation among post-conditions***
1	NULL		E1-2	
2	E1-2		E2-3, E2-4, E2-5	or
3	E2-3		E3-6, $E3-7	and
4	E6-14, E12-14	and	E14-15	

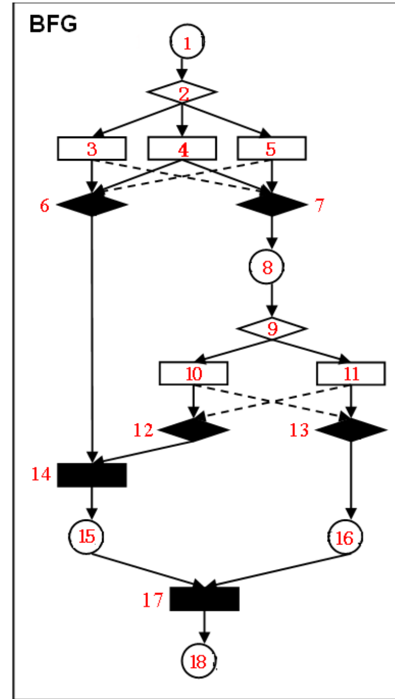

Figure 6. BFG node labeling

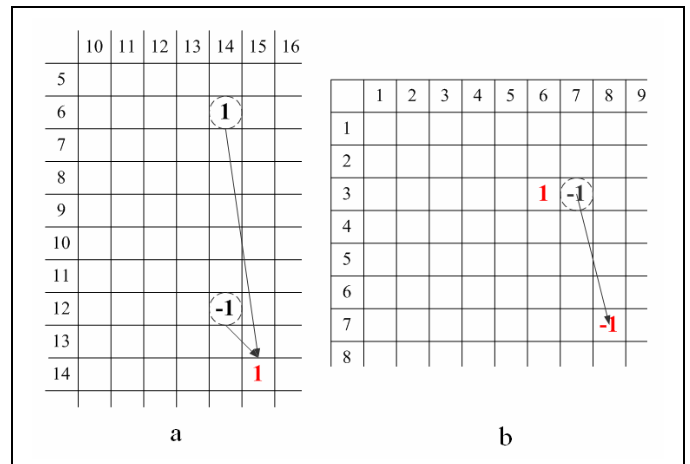

Figure 7. Handling of join node and fork node

	1	2	3	4	5	6
1		1				
2			1			
3						

	1	2	3	4	5	6
1		1				
2				1		
3						

	1	2	3	4	5	6
1		1				
2					1	
3						

Figure 8. Handling of decision node

matrix[2, 3], and recording the second dimension, for example 3, as the node number for next round processing after the said elements are reset to value 0. In this procedure some rules are applied when handling different types of nodes. The searching process stops when meeting the nodes whose post-condition is null, i.e. meeting the final nodes. The searching process should follow three rules:

Note we leave the element in matrix empty to represent value 0.

① When a node to be processed next, we need to examine its “relation among preconditions”. If the relation is an expression, the node cannot be selected to process until all the precondition edges are marked as 1 or -1 in the matrix. Otherwise the node will be selected for next processing, and all the precondition edges will be reset to value 0, as shown in Fig. 7a.

② When a node is being processed, we need to examine its “relation among post-conditions”. If the relation is “and”, all the post-condition edges will be marked in the matrix. If the relation is “or”, the current matrix will be duplicated n times where n is the number of post-condition edges, and each post-condition edge is marked in a duplicate matrix respectively, as shown in Fig. 8. Then the searching process is also forked into n processes.

③ If a post-condition edge of the current processing node is a dead edge, the corresponding element is set to value -1 in the matrix. In the following transformation, all the edge value on the dead path will be -1, as shown in Fig. 7b.

Step5: Collect all the test paths. A complete test path on the BFG is composed by the elements that are ever marked with value 1 or -1 in the searching process. Every generated test path can be marked in the BFG, for example a test path shown in Fig. 9 with bold lines.

E. Test Data Preparation

After the test paths are found, the next step is to attach test data to each path, which can be done manually or (semi-)automatically. A test path may consist of two types of path segments: those consisting of True edges, and those denoting dead paths. In test data generation, we only generate data for the first type of path segments because they contain the activities that will be really executed.

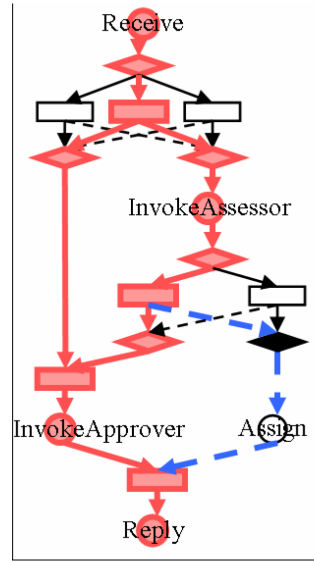

Figure 9. A test path denoted on BFG

In terms of automatic test data generation, there are many existing methods. For example, a common method is to collect all the constraints on a test path into a constraint system, which is then solved to get a solution. Test data generated cover input messages only, or also cover output messages. In the former case, the output messages will be provided manually. For the latter case, the output messages will usually be reviewed and modified by human. In both cases, there are parts of messages that are not used in the constraint system. These parts can be seen as free variables, and thus can be generated using random distribution methods.

For the test path in Fig. 9, we only need to generate test data for the path segments marked with bold lines. The constraint system is:

10000<=getVariableData('request', 'amount')<20000

getVariableData('riskAssessment', 'risk')!='low'

A solution is:

getVariableData('request', 'amount')=10001

getVariableData('riskAssessment', 'risk')='high'

F. Test Case Generation

When test paths and test data are ready, they can be combined into test cases automatically. The test cases contain the test control logic and data that are sufficient for test execution. Compared to test paths, test cases remove some irrelevant information, including “dataHandling-type” (assignment, etc) nodes, assistant nodes (decision, merge, fork, join, etc), and dead path segments. The key of this simplification is to get only the information sufficient for test execution. Here, we give a representation of a test case, expressed as an xml schema in Fig. 10.

In the schema each test path is described in a TestPath element, wherein test data are grouped in an attribute. All the activities on a test path are recorded as sub-elements of

TestPath. Each activity includes an operation and a set of links. Each operation can have up to two values: input and output, which should be populated in a test case.

Fig. 11 shows an example of test case expressed in the exemplary model.

The resulted test cases are abstract - in a sense that they are not directly executable - and must be transformed to a program written in a specific programming language to execute in some testing environment. To get executable test cases, it only need to add one more step - generate executable test cases to current approach. We have implemented a module to generate executable test cases automatically from abstract ones in Java code and integrated the function to a testing environment.

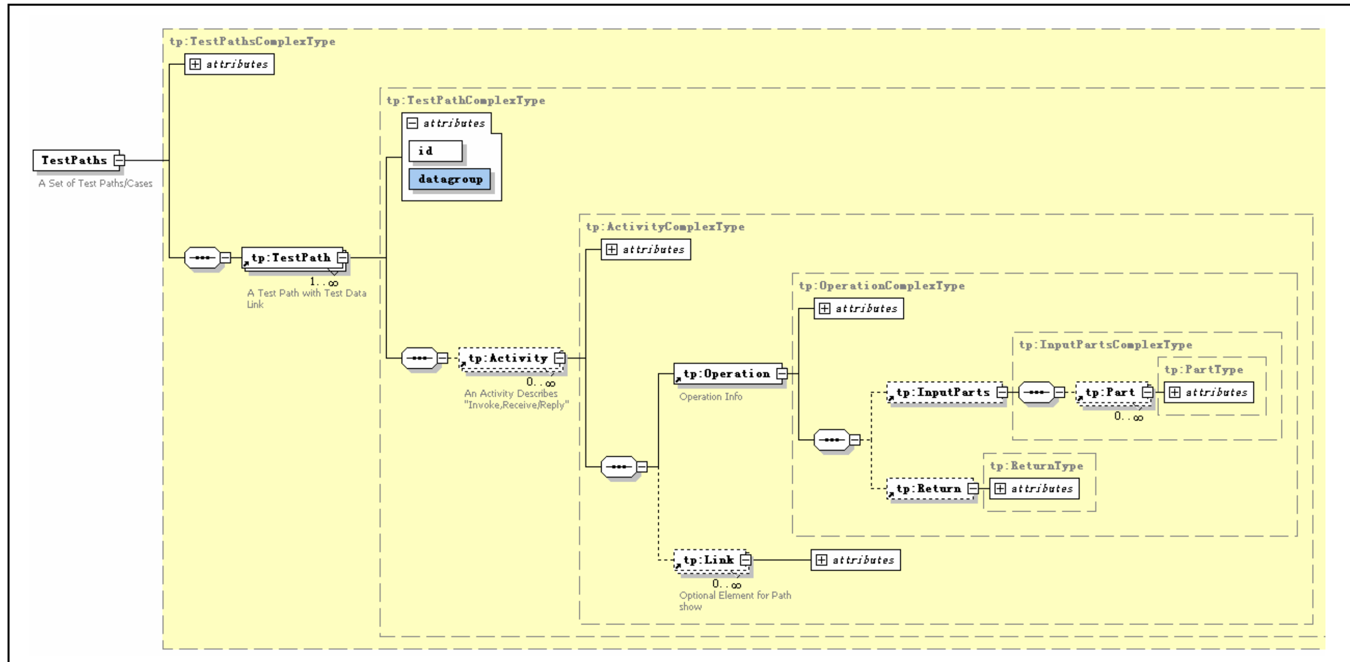

Figure 10. A schema of the test case model

```
<testPath datagroup="datagroup0, datagroup1, …" id="0">
   <activity name="SendPurchaseOrder" times="1">
      <operation opername="sendPurchaseOrder" porttype="wsdl:ProcessPortType">
         <inputParts>
            <part datatype="datatypes/PurchaseOrder" value="sendPurchaseOrder_input.xml"/>
         </inputParts>
      </operation>
   </activity>
   <activity name="PrepareRejection" times="1">
      <operation/>
   </activity>
   <activity name="PrepareInvoice" times="1">
      <operation opername="sendPurchaseOrder" porttype="wsdl:ProcessPortType">
         <return datatype=" XMLSchema:string" value=" sendPurchaseOrder_output.xml"/>
      </operation>
   </activity>
</testPath>
```

Figure 11. An example test case of the loan approval process

IV. Complexity Analysis

A. Evaluation of Node Explosion on BFG

The BPEL to BFG transformation step unravels the folded structures of BPEL (e.g. loop of while) and introduces many additional nodes and edges. Taking the loan approval process as an example, the BPEL graph has 5 nodes and 6 edges while the transformed result BFG has 18 nodes and 25 edges.

For this transformation method, we can estimate the worst-case complexity of node explosion in BFG, which depends on the activities that have more than one outgoing link.

Assume in a given BPEL file, there are m activities that have more than one outgoing link and each activity has A_i outgoing links, then the worst-case complexity of nodes number explosion is:

$O(2^{A1+A2+\ldots+Am})$

But in most situations, the worst case will not hold and the transformed result BFG will not have so many nodes for two reasons:

① In the worst case, each BPEL activity is assumed to be transformed to 2^{Ai} execution scenarios. But in most cases, the 2^{Ai} scenarios can be reduced greatly by removing those invalid ones.

② In the worst case, the nodes number is calculated by multiplication. But in most cases, the number is calculated by addition.

B. Evaluation of Test Paths

When BPEL is transformed to BFG, we can estimate the number of test paths, which depends on the number of decision nodes in BFG.

Assume in a BFG there are m decision nodes and each decision node has N_i outgoing edges, then the worst case complexity of the test path number is:

$O(N_1*N_2*\ldots*N_m)$

Actually, the test path number will be greatly reduced by two means:

① The calculation by multiplying outgoing flows of different decision nodes can be replaced by addition operations.

② If a decision node is in a dead path, the decision node will not contribute to test paths, because all the behavior on a dead path will be removed in test data and case generation.

V. Related Works

The goal of software verification is to validate specific properties of a program (e.g. there is no loop dependency), whereas software testing checks the correctness of a program with respect to functional requirements. Current BPEL verification works [6] are limited in control part analysis, and thus are not suitable for test data preparation, which is indispensable for test case generation. Furthermore verification is mostly based on model checking to explore the whole state space of a program, thus it is prone to state explosion and not scalable to BPEL, especially when BPEL becomes more and more complex.

There are some commercial web service testing tools that declare support for web service test generation [8]. They can generate test cases from WSDL descriptions of the web service. This is different with BPEL test generation in that it is a kind of black-box testing that analyzes the service interface definition, whereas BPEL test generation is a kind of white-box testing that analyzes the internal process definition.

In the area of concurrent program testing, many existing research works are based on reach-ability analysis. A common method is to construct a reach-ability graph [2] of the program under test. The RG is essentially a serialized state transition graph of the global behavior of the concurrent processes or tasks contained in the program. It represents all the possible execution scenarios resulted from nondeterministic run of the concurrent program. This method is limited in practice due to state space explosion problem. Compared to RG based concurrent program testing or verification, our approach will not construct a RG and cover all the serialized paths; instead, we only cover "basic paths", which differ with each other by at least one activity. A basic path is concurrent and can be regarded as a static, compact representation of many serialized paths. In this way, the complexity of state explosion is greatly reduced.

There have been many works in test generation for traditional procedures or member methods of OO classes. They are mostly based on Control Flow Graph. These models are insufficient for BPEL test generation, because BPEL has unique syntax and semantics. In addition, traditional coverage goals for sequential programs are inapplicable to BPEL.

VI. Conclusion

In this paper, we introduce a BPEL test generation method. We define a graph structure to represent BPEL, then search concurrent test paths with a matrix-based algorithm.

Our method is well modularized to support different phases of BPEL testing. People can use the test path searching capability to enumerate test paths only. This is useful when one wants to provide special-purpose test data manually. People can use the test data generation capability to generate test data automatically for test paths that is either automatically found or manually designed for special purpose. People can also use the full capability to automatically generate the abstract test case.

In future, we plan to extend this approach to support exception handling and other BPEL advanced features.

Acknowledgment

We thank for Bin Du for discussion and criticism on a earlier draft of this work. We also thank Jianjun Lu for implementation and verification of part system prototype.

References

[1] Business Process Execution Language for Web Service(BPEL4WS). Available at

http://www.casisopen.org/committees/download.php/2046/BPEL%20V1-1%20Ma%y%205%202003%20Final.pdf

[2] R. Taylor, D. Levine, and C. Kelly, “Structural testing of concurrent programs”, IEEE Transactions on Software Engineering, March 1992, 18(3), pp. 206-215.

[3] R. D. Yang and C. G. Chung, “A path analysis approach to concurrent program testing”, Information and Software Technology, 1992, pp. 34(1): 43-56.

[4] T. Katayama, E. Itoh, and Z. Furukawa, “Test-case generation for concurrent programs with the testing criteria using interaction sequences”, Proceedings of the 6th Asian-Pacific Software Engineering Conference, December 1999, pp. 590-597.

[5] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros, “Workflow patterns”, Distributed and Parallel Databases, 2003, 14(1), pp. 5-51.

[6] Mariya Koshkina and Franck van Breugel, “Modeling and Verifying Web Service Orchestration by means of the Concurrency Workbench”, Proceedings of the Workshop on Testing, Analysis and Verification of Web Services (TAV-WEB), ACM SIGSOFT Software Engineering Notes, September 2004, 29(5).

[7] M. Berkelaar, “Lp_solve”, a public domain Mixed Integer Linear Program solver, available at http://groups.yahoo.com/group/lp_solve/

[8] WebServiceTester™, Optimyz, available at http://www.optimyz.com/servicetester.html