2012 12th International Conference on Quality Software

Scenario-Oriented Testing for Web Service Compositions using BPEL

Chang-ai Sun[1,2*], Yan Shang[1], Yan Zhao[1], Tsong Yueh Chen[3]
[1]School of Computer and Communication Engineering, University of Science and Technology Beijing, 100083 China
[2]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Science, Beijing 100190, China
[3]Faculty of Information and Communication Technologies, Swinburne University of Technology, Melbourne Australia
e-mails: casun@ustb.edu.cn, shangyan@live.com, 952645709@qq.com,tychen@swin.edu.cn

***Abstract*—Applications are increasingly constructed by orchestrating Web services. Ensuring the reliability of such loosely coupled service compositions is a challenging task. In this paper, we propose an automatic scenario-oriented testing approach for service compositions specified by Business Process Execution Language (BPEL). In our approach, an abstract test model (BGM) is first defined to represent the control flows of BPEL specifications, then test scenarios are derived from the BGM with respect to the given coverage criteria, finally test data are generated to drive the execution of resulting test scenarios. A case study is conducted to demonstrate how our approach can be applied to complex real-life service compositions and the experimental results validate the effectiveness of our approach.**

Keywords- Service Compositions; BPEL; Scenario-Oriented Testing

I. INTRODUCTION

In recent years, Service Oriented Architecture (SOA) has been increasingly adopted to develop various applications [6]. In order to execute complex business processes, it is necessary to orchestrate a bundle of Web services. Such orchestrations can be described by Business Process Execution Language (BPEL) [4], which is a widely recognized process-oriented executable service composition language. SOA provides a perfect solution to address the challenging issues in a distributed and heterogeneous environment, such as data exchange and application interoperability. In the meanwhile, ensuring the reliability of loosely coupled SOA software becomes difficult yet important.

Testing SOA involves testing individual Web services and testing service compositions [8]. Service composition testing is greatly different from the traditional integrated testing in two aspects. First, Web services are developed and tested independently, and it is hard to expect all possible scenarios at the side of service developers. Second, services within compositions are usually abstract ones and only bound to concrete Web services at run-time. This run-time binding delays the execution of testing and calls for the on-the-fly testing techniques.

In this paper, we propose an automatic scenario-oriented testing approach for BPEL service compositions. In our approach, an abstract test model is first defined to represent BPEL processes. Then, test scenarios are generated from the test model with respect to the given coverage criteria. Finally, test data are generated and selected to drive the execution of test scenarios. A case study is conducted to demonstrate our approach and validate its applicability and effectiveness.

The rest of the paper is organized as follows. Section II introduces underlying concepts related to BPEL and scenario-oriented testing. Section III proposes the scenario-oriented testing approach for BPEL service compositions. Section IV describes a case study where the proposed approach is used to test a real-life BPEL service composition. Section V describes related works and Section VI concludes the paper and proposes the future work.

II. BACKGROUND

A. Business Process Execution Language (BPEL)

A BPEL process often consists of partner link statements, variable statements, handler statements and interaction steps [4]. *Basic activities* execute an atomic execution step, including *assign*, *invoke*, *receive*, *reply*, *throw*, *wait*, *empty*, and so on. *Structural activities* are composites of basic activities and/or structural activities, including *sequence*, *switch*, *while*, *flow*, *pick*, and so on. In addition, BPEL provides *concurrency* among activities via flow activities and *synchronization* via link tags within flows. *Each link* has a *source activity* and a *target activity*. A transition can happen only when the associated conditions are satisfied.

B. Scenario-oriented Testing

A scenario usually represents an execution path of a software system. The functionalities of a system can be described as a set of scenarios in terms of workflows. To test a system, one needs to first derive a set of possible test scenarios and then execute these scenarios and observe their behaviors. If an execution of scenarios does not happen as expected, then a fault is detected. BPEL provides the concurrency mechanism for concurrent behaviors in workflows. To decide to what extent concurrent elements should be tested, we proposed three coverage criteria in our previous work, namely *weak*, *moderate,* and *strong concurrency coverage* [7].

III. SCENARIO-ORIENTED TESTING FOR BPEL SPECIFICATIONS

A. Overview of Our Approach

To ensure the reliability of service compositions specified using BPEL, we propose a scenario oriented testing approach due to its workflow nature. The proposed approach is sketched in Figure 1. In the context of BPEL specifications, a test scenario corresponds to a set of activities and transitions. The number of test scenarios may be very huge when service

*Corresponding author

1550-6002/12 $26.00 © 2012 IEEE
DOI 10.1109/QSIC.2012.47

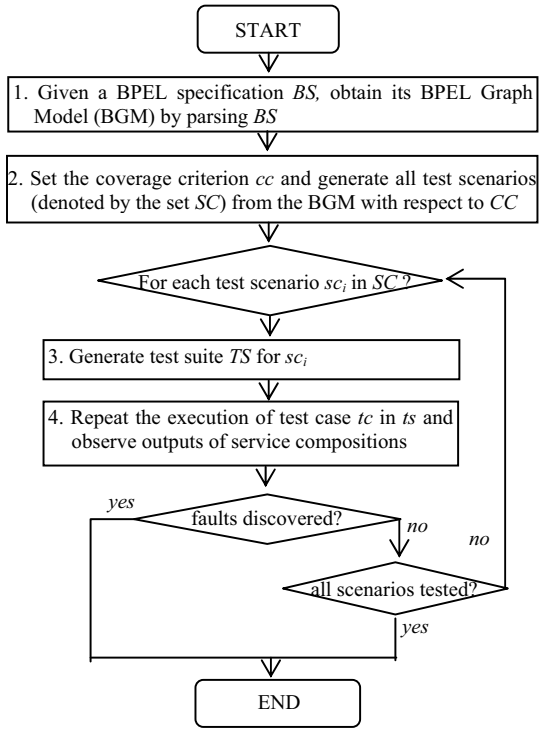

Figure 1. Scenario Oriented Testing of BPEL Specifications

compositions are complex. How to generate a set of test scenarios according to some given coverage criteria is a key issue. We next elaborate the main steps of our approach when applying the approach to test BPEL specifications.

B. BPEL Graph Model(BGM)

To make the testing task simple and effective, we first define an abstract test model called BPEL Graph Model (*BGM*). The BGM of a BPEL specification is an extended graph *BGM*=<*Nodes*, *Edges*>. A node in *Nodes* corresponds to a BPEL activity and is represented as an entry <*id*, *responseid*, *outing*, *type*, *name*>, where

- *id* is the unique identification of an activity. It starts from zero and each activity inside *optional* or *parallel* activities are labeled in a depth-first way.
- *outing* refers to the number of subsequent activities of the activity.
- *type* refers to the type of the activity.
- *name* refers to the name of the activity.
- *responseid* is used to identify the hierarchy of BPEL activities. The definitions of different node type's *responseid* are further explained as follows. Node types in BGM are summarized in Table I.
  - *responseid* of an *action* or *initial* node denotes the next *non-normal* node;
  - *responseid* of a *branch node* denotes its matching *merge* node;
  - *responseid* of a *fork* node denotes its matching *join* node;
  - *responseid* of a *merge* node or *join* node denotes its next node after the *merge* or *join* node, respectively;
  - *responseid* of an *end* node denotes itself;
  - *responseid* of a *cycle* node denotes itself.

An edge in *Edges* corresponds to a transition in BPEL and is represented as an entry <*iID*, *oID*>, where

- *iID* refers to the *ID* of the incoming activity of the transition;
- *oID* refers to the *ID* of the outgoing activity of the transition.

TABLE I. THE DEFINITION OF NODES' TYPE IN BGM

Node Type	Description
initial	The *beginning* of interactions
end	The *end* of interactions
action	A *normal* activity
branch	The beginning of an *optional* activity
merge	The end of an *optional* activity
fork	The beginning of a *parallel* activity
join	The end of a *parallel* activity
cycle	A *loop* activity

C. Conversion of BPEL Specifications into BGMs

We define a set of mapping rules with respect to each type of BPEL activities. For the *normal* activities, they are directly mapped to *action* nodes. For the other four types of activities, we next discuss their mapping rules individually.

- For the *sequential* activities, a sequence of nodes is created and each node corresponds to a child activity; in the meanwhile, edges between the two nodes are added.
- For the *optional* activities, the <*switch*>-<*/switch*> pairs are mapped to *branch-merge* node pairs, and each branch is mapped to a child node.
- For the *loop* activities, their child activities repeat until some conditions are satisfied. For the *loop* activities, their child activities are mapped to a sequence of nodes, and a *cycle* node are added in the end whose outgoing edges are towards the first node in the loop and the first node after the loop.
- For the *parallel* activities, they are used to support concurrency with flows in BPEL. For those activities having no the source or target activities, they are executed in parallel. In this context, the <*flow*>-<*/flow*> pairs are mapped to *fork-join* node pairs, and each parallel branch is mapped to a child node of *fork* and the father node of *join*. For those child activities having source and target elements within the flows, transitions are enabled depending on whether the link's conditions are satisfied. In this situation, their mapping rules have been discussed as above.

Based on these mapping rules, we propose a recursive conversion algorithm as illustrated in Figure 2. In the treatment, we ignore the creation of edges for simplicity and they can be supplemented through analyzing the order of nodes. The algorithm is able to convert complex and nested structural activities, and finally terminate when all basic activities are converted. Its time complexity is proportional to the number of basic activities, namely O(*n*) where *n* denotes the number of basic activities.

D. Generating Test Scenarios via BGM

Through the conversion, we get an abstract test model for the BPEL specification. It provides a formal basis from which we can define algorithms to automatically generate test scenarios with respect to the given coverage criterion. We propose an algorithm to generate test scenarios from BGM with respect to the *weak concurrency coverage*, as illustrated in Figure 3. The algorithm generates test scenarios from a *start* node to an *end* node recursively. In the first round, the *start* node corresponds to the *initial* node and the *end* node corresponds to the *end* node of a BGM. After that, a *start* node and an *end* node form a segment of BGM, and partial test scenarios are generated according to the types of nodes. The algorithm traverses all nodes once, so its time complexity is proportional to the number of nodes, i.e. O (*n*) where *n* denotes the number of nodes in BGM.

E. Generating Test Suite for Test Scenarios

BPEL processes are often implemented as composite Web services. In order to generate a test suite, we first decide the input format by analyzing the interfaces of the composite Web service. Then, a large number of test data is randomly generated. Finally, we select a test suite from these test data to satisfy the constraints along the paths with respect to each test scenario.

F. Executing Tests

Before the execution, BPEL processes are often deployed in a service container. At the runtime, BPEL specifications are interpreted by a BPEL engine. To automatically test BPEL specifications, a test proxy is needed.

```
INPUT: B is a BPEL specification
OUTPUT: G is a BPEL graph model (BGM)
PROCEDURE Convert (B, G)
1.    n ← getTopActivityNumber (B);
2.    FOR i = 1 TO n DO
3.      currentActivity ← getActivity(B, i);
4.      IF (currentActivity is a normal activity) THEN
5.        create an action node a;
6.        add a to G;
7.      ELSE IF
8.        IF (currentActivity is a sequential activity) THEN
9.          BS ← getBPELSegment(currentActivity);
10.         Convert (BS, G);
11.       ENDIF
12.       IF (currentActivity is a branch activity) THEN
13.         Create a branch node b and add b to G;
14.         BS ← getBPELSegment(currentActivity);
15.         Convert(BS, G);
16.         Create a merge node m and add m to G;
17.       ENDIF
18.       IF (currentActivity is a loop activity) THEN
19.         BS ←getBPELSegment(currentActivity);
20.         Convert(BS, G);
21.         Create a cycle node c and add c to G;
22.       ENDIF
23.       IF (currentActivity is a parallel activity) THEN
24.         Create a fork node f and add f to G;
25.         BS ← getBPELSegment(currentActivity);
26.         Convert(BS, G);
27.         Create a join node j and add j to G;
28.       ENDIF
29.   ENDIF
END PROCEDURE
```

Figure 2. The conversion algorithm for BPEL specifications

IV. Case Study

A. Subject Program

SmartShelf [5] receives an input message with *name*, *amount* and *status*. The process returns an output message with *quantity*, *location*, and *status*. It behaves as a typical concurrent program and hence is very representative. The BPEL service composition for SmartShelf involves the interactions among 14 Web services.

B. Scenario-Oriented Testing of Smartshelf

Firstly, we convert the service composition of SmartShelf into its BGM. Secondly, we generate test scenarios from the BGM. When the weak coverage criterion is used, we generate 12 scenarios which are represented as sequences of action nodes. Thirdly, we randomly generate a large amount of input data by analyzing the WSDL file of SmartShelf BPEL. As a consequence, a test suite of 120 test cases are created to satisfy weak concurrency coverage criterion. Finally, we execute tests using the resulting test suite. During the tests, we compare the actual outputs with the expected outputs for each test.

C. Results and discussions

Mutation analysis is widely used to assess the adequacy of a test suite and the effectiveness of testing techniques [1]. Hence, it is used to evaluate the effectiveness of our

```
INPUT: G is a BPEL Graph Model (BGM).
OUTPUT: TestPaths is a set of test scenarios.
PROCEDURE WeakCoverage(Node start, Node end)
1.    IF (start != end) THEN
2.      IF (start.type = "action") THEN
3.        tmpPaths←WeakCoverage(start.afterNodes.get(0), end);
4.        add start node to each path in tmpPaths;
5.        RETURN tmpPaths;
6.      ENDIF
7.      IF(start.type = "branch" || start.type = "fork") THEN
8.        node←getResponseNode(start.responseid);
9.        FOR i = 1 TO start.afterNodes.size() DO
10.         tmp[i] ←WeakCoverage(start.afterNodes.get(i), node);
11.       ENDFOR
12.       tmpPaths←WeakCoverage(node.afterNodes.get(0), end);
13.       merge tmp[] and tmpPaths into resultPaths;
14.       RETURN resultPaths;
15.     ENDIF
16.     IF(start.type = "cycle") THEN
17.       startNode ← the first node in the loop;
18.       endNode ← the last node in the loop;
19.       tmpPaths1 ←WeakCoverage(startNode, endNode);
20.       tmpNode ←the first node after the loop;
21.       tmpPaths2 ← WeakCoverage(tmpNode, end);
22.       merge tmpPaths1 and tmpPaths2 into resultPaths;
23.       RETURN resultPaths;
24.     ENDIF
25.   ELSE
26.     add start node to resultPaths;
27.     RETURN resultPaths;
28.   ENDIF
END PROCEDURE
```

Figure 3. The Algorithm of Generating Test Scenarios from BGM with respect to *weak concurrency coverage*

approach. We seed faults into the BPEL code of SmartShelf using mutation operators described in [2]. As a result, we have totally 25 non-equivalent mutants. Table II summarizes the mutation score of scenario-oriented testing using the *weak concurrency coverage*.

TABLE II. A SUMMARY OF MUTATION SCORE USING WEAK CONCURRENCY COVERAGE

Number of Killed Mutants	Number of Total Mutants	Mutation Score (MS)
23	25	92%

The result is very encouraging, and 92% mutants are killed. This means that when executing scenario oriented testing on BPEL specifications, *weak concurrency coverage* may be the most cost-effective choice because it can detect most of faults with fewer test cases. With this case study, we have validated the feasibility of our approach by applying it to a realistic and complex BPEL service composition.

V. RELATED WORK

Fanjul et al. [3] proposed a model based approach to testing BPEL processes. In their approach, SPIN is employed as a model to generate the test suite for BPEL specifications. A transition coverage criterion is employed to select test cases. Since the approach needs to model all possible states of BPEL specifications, it is subject to the state space explosion problem when service compositions are complex and thus has limitations in practice.

Yuan et al. [9] proposed a graph-search based test case generation method for BPEL specifications. An extended control flow graph (BFG) is defined to represent BPEL specifications, and then traverse the BFG model to generate concurrent test paths. No experiments are reported on the effectiveness of their approach. Similarly, Yan et al. [10] proposed an extended control flow graph (XCFG) to represent BPEL specifications, and sequential test paths are generated from XCFG. Experimental results in terms of fault detection capability are missing. Our approach is similar since our approach also employs a graph model to abstract the BPEL specification. On the other hand, our approach is able to generate on demand executable test cases, and its fault detection capability is reported using a case study.

Zhang et al [11] proposed a model-based approach to generating test cases for BPEL specifications. The approach transforms Web service flows into UML activity diagrams and then generates test cases from the activity diagram. This approach generates abstract test cases from a semi-formal representation. However, our approach generates executable test cases and supports concurrency coverage criteria on demand when generating test scenarios.

VI. CONCLUSIONS AND FUTURE WORK

We have presented a scenario-oriented testing approach to address the challenges related to ensuring the quality of BPEL service compositions. This approach first converts BPEL specifications into an abstract test model, and test scenarios are then automatically generated from the model with respect to a given coverage criterion. Test suites can be obtained semi-automatically for the derived test scenarios. A case study was conducted to demonstrate how the approach can be applied to realistic and complex service compositions. Experimental results validate the feasibility and effectiveness of the approach.

In our future work, we plan to explore fully automatic test suite generation via constraint solver techniques and automatic verification of testing results.

ACKNOWLEDGMENT

Authors thank to Tieheng Xue and Ke Wang from University of Science and Technology Beijing for BPEL implementations of Smartshelf. This research is supported by the National Natural Science Foundation of China (Grant No. 60903003), the Beijing Natural Science Foundation of China (Grant No. 4112037), the Fundamental Research Funds for the Central Universities (Grant No. FRF-SD-12-015A), the Open Funds of the State Key Laboratory of Computer Science of Chinese Academy of Science (Grant No. SYSKF1105), and a discovery grant of the Australian Research Council (Grant No.DP0771733).

REFERENCES

[1] R. A. DeMillo, R. J. Lipton, F. G. Sayward, "Hints on test data selection: Help for the practicing programmer", *IEEE Computer*, 1978, 1(4): 31-41.

[2] A. Estero-Botaro, F. Palomo-Lozano, and I. Medina-Bulo, "Quantitative Evaluation of Mutation Operators for WS-BPEL Compositions", *Proceedings of ICST 2010*, 2010, pp.142-150.

[3] J. García-Fanjul, J. Tuya and C. de la Riva, "Generating test cases specifications for BPEL compositions of web services using SPIN", *Proceedings of International Workshop on Web Services-Modeling and Testing,* 2006, pp.83-94.

[4] OASIS, "Web Services Business Process Execution Language Version 2.0", OASIS Standard, 2007.

[5] J. Park, M. Moon, and K. Yeom, "The BCD view model: Business analysis view, service Composition view and service Design view for service oriented software design and development", *Proceedings of 12th IEEE International Workshop on Future Trends of Distributed Computing System*, 2008, pp. 37-43.

[6] M. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-Oriented Computaing: A Research Roadmap", *International Journal on Cooperative Information Systems, 2008*, 17(2): 223-255.

[7] C. Sun, "A Transformation-based Approach to Generating Scenario-Oriented Test Case from UML Activity Diagrams for Concurrent Applications", *Proceedings of COMPSAC 2008*, IEEE Computer Society, 2008, pp.160-167.

[8] C. Sun. "On Open Issues on SOA-based Software Development", *China Sciencepaper Online*, http://www.paper.edu.cn/index.php/ default/releasepaper/ content/201107-461. 2011.

[9] Y. Yuan, Z. Li, and W. Sun, "A Graph-search Based Approach to BPEL4WS Test Generation", *Proceedings of ICSEA'06*, 2006. pp.14-14.

[10] J. Yan, Z. Li, Y. Yuan, W. Sun, and J. Zhang, "BPEL4WS Unit Testing: Test Case Generation Using a Concurrent Path Analysis Approach", *Proceedings of ISSRE '06*, 2006, pp.75-84.

[11] G. Zhang, M. Rong, and J. Zhang, "A Business Process of Web Services Testing Method Based on UML 2.0 Activity Diagram", *Proceedings of Intelligent Information Technology Application Workshop*, 2007, pp.59-65.