Toward Effectively Locating Integration-Level Faults in BPEL Programs

Chang-ai Sun[1,2*], Yimeng Zhai[1], Yan Shang[1], Zhenyu Zhang[2]
[1]School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing 100083, China
[2]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Science, Beijing 100190, China
Emails: casun@ustb.edu.cn, 570282867@qq.com, shangyan@live.com, zhangzy@ios.ac.cn

***Abstract*—Business Process Execution Language (BPEL) is a widely recognized executable service composition language. Since BPEL integrates services of desired functionality to compose business processes, it is significantly different from typical programming languages. How to effectively locate the integration-level faults in BPEL programs is an open issue. In this paper, we propose the *BPEL fault localization guidelines* based on the characteristics of BPEL programs, and adapt Tarantula, a traditional fault localization technique, to locate the integration-level faults in BPEL programs. We also conducted an empirical study to demonstrate the feasibility of our methodology.**

Keywords- Service Compositions; BPEL; Fault Localization

I. INTRODUCTION

Service Oriented Architecture (SOA) has been increasingly adopted to develop various applications [7][9]. In the context of SOA software, Web services are basic units which provide their functionalities by exposing a set of interfaces, and are coordinated in some way to execute complex business processes. The Business Process Execution Language (BPEL) [5] is a process-oriented executable service composition language which can be used to construct loosely coupled systems by orchestrating a bundle of Web services. Such service compositions exhibit some specific features. For instance, Web services under composition can be implemented in any programming languages and from different application domains. Such features make BPEL programs different from traditional module integrations.

Basically, BPEL integrates different services of desired functionalities to compose a business process. There is no guarantee that such composition is bug-free. However, there is no dedicated study on locating the representative integration-level faults in BPEL.

In this paper, we investigate how to effectively locate the integration-level faults within BPEL programs. We investigate the characteristics of BPEL programs and propose to adapt Tarantula [4], which is one of the most popular fault localization techniques, to effectively locate faults in BPEL programs. An empirical study is conducted to validate the feasibility of our methodology, and report the effectiveness of fault localization.

The contributions of this paper are three-fold. Firstly, we propose the BPEL fault localization guideline based on the characteristics of the representative BPEL integration-level faults. Secondly, we adapt a traditional fault localization technique to locate the integration-level faults in BPEL programs. Thirdly, we empirically evaluate the feasibility of our methodology the report the effectiveness of fault localization.

The rest of the paper is organized as follows. Section II introduces the underlying concepts related to BPEL. Section III elaborates on how to adapt Tarantula to locate integration-level faults in BPEL programs. Section IV describes the empirical study which is used to validate the feasibility of the proposed methodology. Section V concludes the paper and discusses the future work.

II. BACKGROUND

BPEL [11] is an executable service composition language which executes complex business processes by orchestrating Web services. BPEL programs are significantly different from the traditional programs. Firstly, BPEL provides an explicit integration mechanism (architectural glues) to compose Web services into large-scale systems, while such integrations in traditional programs are implicit. Secondly, Web services composed by BPEL programs may be implemented in different programming languages, while modules in the traditional programs are usually implemented in the same programming languages. Thirdly, BPEL programs are represented as XML files, and the statements are not the same as the one in the traditional programs. Finally, BPEL provides *concurrency* among activities via flow activities and *synchronization* via link tags within flows, which is not common in the traditional programs. Testing such programs meets new challenges [2][9].

Usually, a BPEL program consists of four sections, namely *partner link section*, *variable section*, *handler section* and *interaction section* [5]. The *partner link section* describes the relationship among the BPEL process and invoked Web services. The *variable section* defines input and output messages. The *handler section* declares the handlers when an exception or specific event occurs. The *interaction section* describes the process how external Web Services are coordinated to execute a business process. Activities are the basic interaction units of BPEL processes, and are further divided into basic activities and structural activities. The former executes an atomic execution step. The latter are composites of basic activities and/or structural activities, including *sequence*, *switch*, *while*, *flow*, and so on.

Debugging is a very challenging and inevitable task during software development. Debugging starts after testers detect a fault. To debug a program, one first needs to know the possible location that the fault may happen to, and then attempt to revise the relevant codes. In this context, locating the suspicious statements is crucial, and in recent years

* Corresponding author.

1550-6002/12 $26.00 © 2012 IEEE
DOI 10.1109/QSIC.2012.48

IEEE computer society

various fault localization techniques have been proposed to improve the performance of debugging activities [4][8][11][12]. Their effectiveness has been validated through empirical studies on typical programs such as C or Java [8][11][12]. For example, The intuition behind the Tarantula technique is that the entities in a program covered by "failed" test cases are more likely to be faulty than those that are covered by "passed" test cases. Following this intuition, the suspiciousness score of an entity *e* can be calculated using the following equation.

$$suspiciousness(e) = \frac{\frac{failed(e)}{totalfailed}}{\frac{passed(e)}{totalpassed} + \frac{failed(e)}{totalfailed}} \quad (1)$$

where *passed(e)* is the number of "passed" test cases that executed the entity *e* at least once; *failed(e)* is the number of "failed" test cases that executed the entity *e* at least once; *totalpassed* and *totalfailed* are the sum of "passed" test cases and "failed" test cases, respectively.

Traditional fault localization techniques work out a ranked list of suspicious program elements and suggest the programmer to search for faults along the resulting ranked list. However, when these techniques are employed to debug BPEL service compositions, are they still effective with respect to the specific features of BPEL programs? In this paper, we will propose how to employ traditional fault localization techniques to locate the integration-level faults in BPEL based on the characteristics of BPEL programs.

III. FAULT LOCALIZATION FOR BPEL PROGRAMS

In this section, we first present substantial concerns when debugging BPEL programs, and then employ a fault localization technique to demonstrate how to adapt it to locate faults in BPEL programs.

A. Debugging Concerns of BPEL Programs

Based on the BPEL programming model discussed in Section II, the most important concerns are as follows when debugging BPEL programs.

(1) Integration level debugging

As mentioned before, BPEL is a kind of architectural glues which is used to assemble Web services to build an executable process. BPEL programs only focus on the integration of Web services, and do not at all touch the implementation of Web services. This means that service integrations and service implementations are completely separated. In this context, we only focus on the integration faults when debugging BPEL programs.

```
<invoke inputVariable="request" name="invokeapprover"
   operation="approve" outputVariable="approvalInfo"
   partnerLink="approver" portType="apns:loanApprovalPT">
      <target linkName="receive-to-approval" />
      <target linkName="assess-to-approval" />
      <source linkName="approval-to-reply" />
</invoke>
```

Figure 1. An illustration of BPEL statement blocks

(2) Interaction debugging

Among four sections of BPEL program as illustrated in Section II, only interaction statements represent the execution steps of a business process and direct interactions with services under composition. These statements are crucial to the correctness of BPEL programs, while statements in other sections are not executable. One should give the highest priority to the interaction section when she debugs BPEL programs. In this paper, we will focus on how to effectively locate faults related to interactions. Locating faults in other sections beyond the scope of this study.

B. Dividing BPEL Programs into Statement Blocks

Based on the BPEL programming model and debugging concerns discussed above, we assume that possible faults only happen in statement blocks in the interaction section. Here, BPEL programs are represented as a set of hierarchical statement blocks. A *statement block* corresponds to a set of elements enclosed by the matched XML tags. Figure 1 illustrates an example of the *invoke* statement block, which describes the interaction through specifying the activity type, operation name, input variable, output variable, partner link, port type, target link name and source link name. Each activity can be treated as a statement block, and the whole BPEL program is composed of statement blocks in a hierarchical way.

In our model, statement blocks are further classified into *atomic statement block* and *non-atomic statement block*. The former refers to an atomic execution step, including *assign*, *invoke*, *receive*, *reply*, *throw*, *wait* and *empty.* The latter is composites of atomic statement blocks or non-atomic statement blocks, including *sequence*, *switch*, *while*, *flow* and *pick*. We further abstract those statement blocks with similar semantics as the same type. In this context, non-atomic statement blocks can be classified into the following four types, namely *sequential*, *optional*, *parallel*, and *loop*.

- *Sequential statement blocks* refer to those whose child statement blocks are executed in a sequential order, such as *sequence* activity.
- *Optional statement blocks* refer to those among whose child statement blocks, only one can be executed, such as *switch*, *if/else/elseif*, and *pick* activity.
- *Parallel statement blocks* refer to those whose child statement blocks are executed simultaneously, such as *flow* activity.
- *Loop statement blocks* refer to those whose child statement blocks are executed all the time until some conditions are satisfied, such as *while*, *unitlWhile* and *forEach* activity.

C. BPEL Fault Localization Guidelines

According to the characteristics of the integration-level faults in BPEL programs, we design the following fault localization guidelines to facilitate accurate fault localization in BPEL programs.

(1) Usually, fault localization techniques can locate faults accurately in the traditional programs. This may not be true

because statement blocks in BPEL programs are composed in a hierarchical way, and when a fault occurs at the higher level statement blocks, it may propagate downstream to the lower-level statement blocks. Therefore, for statement blocks in *optional*, *loop* or *parallel* having the highest suspiciousness, one should debug in the condition part.

(2) If a statement block in the sequential statement block has the highest suspiciousness, one should debug faults from near to far at the most suspicious statement blocks. It is rather natural to blame those statement blocks close to the most suspicious statement blocks if the latter are not the fault.

D. Locating Integration-Level Faults in BPEL Programs

We next demonstrate how we use Tarantula to locate the integration-level faults in BPEL in four steps.

Step 1: When a failure *f* is reported during testing BPEL program *bp*, we first get the test suite *ts* which was used to reveal *f*.

Step 2: For each test case *t* in *ts*, we run *bp* to decide which statement blocks are covered, and the current test passes or fails. Correspondingly, *t* is identified as a "passed" test case or a "failed" test case.

Step 3: Repeat **Step 2** until all test cases in *ts* are executed, and then apply *Tarantula* to figure out the most suspicious statement block *mssb* with the available testing history.

Step 4: According to the type of *mssb*, the possible position set *pps* is recommended to check using the *BPEL fault localization guideline*.

IV. An Empirical Study

A. Choosing Subject Programs

Two programs *SupplyChain* and *SmartShelf* are chosen as subjects in this empirical study. They demonstrate most of major features of BPEL.

SupplyChain [1] is widely used to demonstrate common features of BPEL. The implementation involves two Web services and consists of 11 BPEL statement blocks. The working process receives an order which is represented by an input message consisting of *name* and *amount* of goods and returns an output message to indicate whether the warehouse can accept the order.

SmartShelf [6] is complex and demonstrates some other features of BPEL, for instance supporting the concurrency behavior. BPEL program for SmartShelf involves 14 Web services' interactions and consists of 48 statement blocks. SmartShelf receives an input message called commodity, which is composed of three fields, namely name, amount and status and returns an output message which is composed of quantity, localization and status.

B. Mimicking Faults of BPEL Programs

We first seed some faults into two BPEL programs. Among the 26 mutation operators proposed in [3], only six of them are selected to generate the mimicking faults. This is because not all these mutation operators are applicable to our subject programs. In addition, we have manually seeded faults into BPEL programs because up to now, there is yet not an automatic and practical mutation system for this task.

For the BPEL program of the SupplyChain and SmartShelf, 13 and 20 faults are generated, respectively.

C. Generating Test Suites

To apply the above fault localization techniques to BPEL programs, test suites (including "passed" test cases and "failed" test cases) are required. For this task, the scenario-oriented testing approach proposed in our previous work [10] is employed.

We first generate a set of test scenarios for the two BPEL programs with respect to a given coverage criteria. Each test scenario corresponds to a sequence of statement blocks. For a specific test scenario, we derive 20 test cases which can be used as inputs to drive the execution of the test scenario. As a result, we derive a test suite of 40 test cases for the SupplyChain. For the SmartShelf, we generate 10 test cases for each test scenario and finally derive a test suite of 120 test cases.

D. Adapting Tarantula to Locate Faults

We then execute tests. For each test case, we record its actual output of a mutant and compare it with the expected one, which corresponds to the output of the original BPEL program for the same test case.

If the actual output is the same as the expected one, this test case is said to be a "passed" test case; it is said to be a "failed" test case. For each fault, we apply the proposed method to figure out the most suspicious statement block. If it contains the actual fault, the fault localization is *successful*; otherwise, it *failed.*

Finally, we use the average fault localization success rate to measure the effectiveness of Tarantula.

E. Results and Discussions

The evaluation results of Tarantula are summarized in Table I and II, respectively. Note that *No* refers to fault number; *NE* refers to "Number of test cases whose output is the same as the expected"; *NNE* refers to "Number of test cases whose output is different from the expected"; *MSSB* refers to "Most Suspicious Statement Block"; *PPS* refers to "fault's Possible Position Set"; and *LF* refers to "whether Locate the Fault"

TABLE I. Evaluation Results of of Tarantula for SupplyChain

No	NE	NNE	MSSB	PPS	LF
1	20	20	5-8	{4, 5}	Y
2	30	10	5-8	{4, 5}	Y
3	30	10	5-8	{4, 5}	Y
4	30	10	5-8	{4, 5}	Y
5	30	10	5-8	{4, 5}	Y
6	39	1	9-10	{4, 9}	Y
7	21	19	5-8	{4, 5}	Y
8	20	20	5-8	{4, 5}	Y
9	29	11	5-8	{4, 5}	Y
10	31	9	9-10	{4, 9}	Y
11	20	20	9-10	{4, 9}	Y
12	30	10	5-8	{4, 5}	Y
13	30	10	9-10	{4, 9}	Y

("*Y*" means success, while "*N*" means failure).

From Tables I and II, we have the following observations.

- For the BPEL programs of SupplyChain, Tarantula can successfully locate all the 13 faults. The correctness percentage of fault localization is 100%. For the SmartShelf, it can successfully locate 16 of 20 faults. The correctness percentage of fault localization is 80%.
- The effectiveness of the Tarantula technique varies when used to locate faults in two BPEL programs. This is because the BPEL program of SupplyChain is simpler and faults are clustered at some locations, while the BPEL program of SmartShelf is rather complex and the faults are scattered in a widen code region.

F. Summary

Through this empirical study, on one hand we validated the feasibility of the proposed methodology and the BPEL fault localization guideline; on the other hand we adapt a popular traditional fault localization technique Tarantula to locate the integration-level faults in BPEL programs, and show that it is effective.

V. CONCLUSIONS AND FUTURE WORK

How to effectively locate the representative integration-level faults in BPEL programs is inadequately studied. In this paper, we have proposed a BPEL fault localization guidelines and demonstrate how to adapt a popular fault localization technique in this domain.

We also have conducted an empirical study, and successfully validate the feasibility of the proposed methodology and the effectiveness of the fault localization technique when adapted to debug BPEL programs.

In our future work, we plan to involve more BPEL subject programs and more types of faults by means of mutation operators to evaluate the effectiveness of more fault localization techniques. Further, we will integrate with our previous work [12] to locate faults in BPEL programs from user feedback or failure reports.

TABLE II. EVALUATION RESULTS OF TARANTULA FOR SMARTSHELF

No	NE	NNE	MSSB	PPS	LF
1	4	116	13-25	{12, 13}	Y
2	116	4	21-24	{16, 21}	N
3	4	116	13-25	{12, 13}	Y
4	36	84	13-25	{12, 13}	Y
5	84	36	26-27	{12, 26}	Y
6	44	76	17-20	{16, 17}	Y
7	116	4	21-24	{16, 21}	Y
8	40	80	13-25	{12, 13}	N
9	76	44	17-20	{16, 17}	Y
10	84	36	21-24	{16, 21}	Y
11	0	120	1-48	{1, 2}	N
12	6	114	40-45	{39, 40}	Y
13	114	6	46-47	{39, 46}	Y
14	0	120	1-48	{1, 2}	N
15	54	66	40-45	{39, 40}	Y
16	66	54	46-47	{39, 46}	Y
17	80	40	21-24	{16, 21}	Y
18	80	40	26-27	{12, 26}	Y
19	60	60	35-36	{30, 35}	Y
20	60	60	46-47	{39, 46}	Y

ACKNOWLEDGMENT

Authors thank to Tieheng Xue and Ke Wang from University of Science and Technology Beijing for their implementations of BPEL programs which are used for the empirical study. 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) and the Open Funds of the State Key Laboratory of Computer Science of Chinese Academy of Science (Grant No. SYSKF1105).

REFERENCES

[1] L. Baresi, R. Heckel, S. Thöne, and D. Varró, "Modeling and validation of service-oriented architectures: application vs. style", *Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of Software Engineering (ESEC/FSE-11)*, 2003, pp. 68–77.

[2] G. Canfora, and M. Di Penta. "*Service Oriented Architecture Testing: A Survey*", LNCS 5413, Springer, 2009, pp78–105.

[3] A. Estero-Botaro, F. Palomo-Lozano, and I. Medina-Bulo, "Mutation Operators for WS-BPEL 2.0", *Proceedings of 21th International Conference on Software & Systems Engineering and their Applications (ICSSEA 2008)*, *2008*. pp.1-7

[4] J. A. Jones and M. J. Harrold. "Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique", *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, 2005, pp. 273–282.

[5] OASIS, "Web Services Business Process Execution Language Version 2.0", http://docs. oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html, 2007.

[6] 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.

[7] 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.

[8] M. Renieris and S. Reiss. "Fault localization with nearest neighbor queries", *Proceedings of the International Conference on Automated Software Engineering*, 2003, pp.30-39.

[9] 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.

[10] C. Sun, Y. Shang, Y. Zhao, T.Y. Chen, "Scenario-Oriented Testing of Service Compositions using BPEL". to appear in Proceedings of QSIC 2012.

[11] W. E. Wong, Y. Qi, L. Zhao and K. Y. Cai. "Effective Fault Localization using Code Coverage", *Proceedings of 31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, 2007, pp.449-456.

[12] Z. Zhang, W. K. Chan, and T. H. Tse, "Fault Localization without Success Record: a Review and Proposal", to appear in IEEE Computer, 2012.