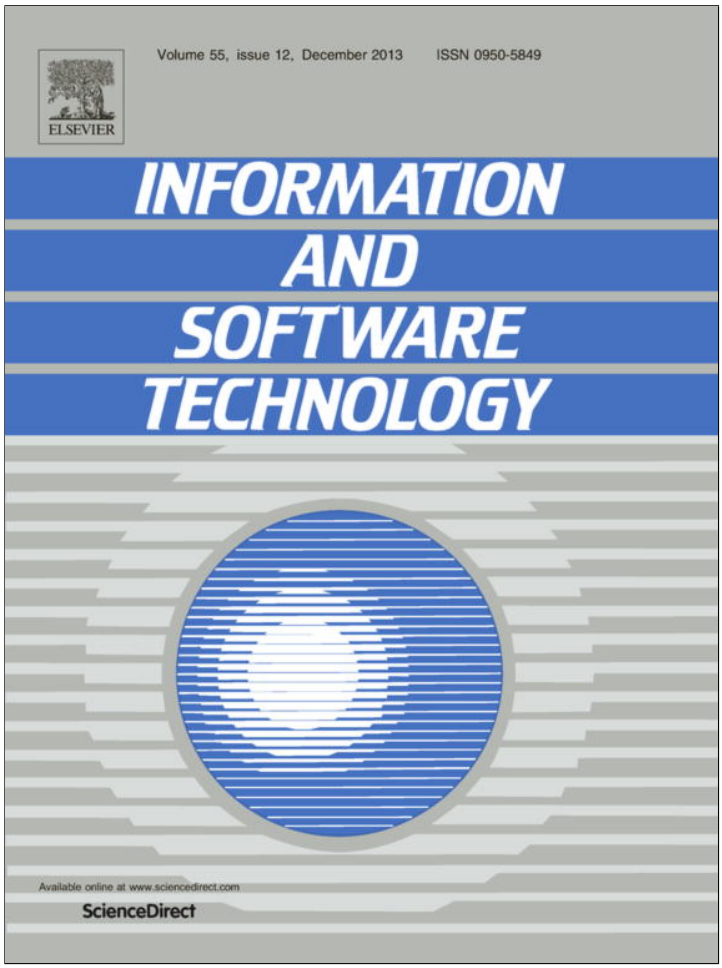

Information and Software Technology 55 (2013) 2140–2153

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

BPELDebugger: An effective BPEL-specific fault localization framework ☆

Chang-ai Sun [a,b,*], Yi Meng Zhai [a], Yan Shang [a], Zhenyu Zhang [b]

[a] School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing, China
[b] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China

ARTICLE INFO

Article history:
Received 6 December 2012
Received in revised form 25 June 2013
Accepted 23 July 2013
Available online 6 August 2013

Keywords:
Service compositions
BPEL
Fault localization
Fault localization guidelines
Integration and interaction faults

ABSTRACT

Context: Business Process Execution Language (BPEL) is a widely recognized executable service composition language, which is significantly different from typical programming languages in both syntax and semantics, and especially shorter in program scale. How to effectively locate faults in BPEL programs is an open and challenging problem.
Objective: In this paper, we propose a fault localization framework for BPEL programs.
Method: Based on BPEL program characteristics, we propose two fault localization guidelines to locate the integration and interaction faults in BPEL programs. Our framework formulates the BPEL fault localization problem using the popular fault localization problem settings, and synthesizes BPEL-specific fault localization techniques by reuse of existing fault localization formulas. We use two realistic BPEL programs and three existing fault localization formulas to evaluate the feasibility and effectiveness of the proposed fault localization framework and guidelines.
Result: Experiment results show that faults can be located with the fewest code examining efforts. That is, the fault-relevant basic block is assigned the highest suspiciousness score by our fault localization method. The experiment results also show that with the use of the proposed fault localization guidelines, the code examining efforts to locate faults are extraordinarily reduced.
Conclusion: We conclude that the proposed framework is feasible in synthesizing effective fault localization techniques, and our fault localization guidelines are very effective to enhance existing fault localization techniques in locating faults in BPEL programs.

1. Introduction

Service Oriented Architecture (SOA) has been increasingly adopted to develop various distributed systems [26,30]. In the context of SOA, 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) [23] is a process-oriented executable service composition language, which can be used to construct loosely coupled systems by orchestrating a bundle of Web services. Service composition corresponds to the integration of modules in the context of traditional software development paradigms. In the meanwhile, such service composition exhibits some specific syntactic and semantic features. In detail, service compositions retain control structures such as sequences, branches and loops and at the same time introduce new elements that do not exist in typical programming languages such as C, C++ or Java. For example, *partner links*, *flows*, and *handlers* for compensations are elements, which are seldom touched in the traditional programs and need special attentions. Service compositions are represented as XML files. It means that the dynamic behaviors of a program are embedded in the XML-based specification, which lacks the effective fault localization techniques [4,29]. Web services under composition can be implemented in any programming languages and can be from different application domains. These features make BPEL programs significantly different from traditional module integrations, and testing such programs meets new challenges [7,30]. In particular, how can we effectively debug BPEL programs after detecting a fault? Answering this question calls for new fault localization framework and techniques.

Debugging is a complex and time-consuming activity, and in recent years various fault localization techniques have been proposed to improve the performance of debugging activities [17]. Their effectiveness has been validated through empirical studies on typical programs, such as C or Java programs. However, when these techniques are employed to debug BPEL programs, are they still effective and efficient? On the other hand, previous studies have the observation that larger programs have more information

☆ A preliminary version of this paper was presented at the 12th International Conference on Quality Software (QSIC 2012) [32].

* Corresponding author at: School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing, China. Tel.: +86 1062332931; fax: +86 1062332873.
E-mail addresses: casun@ustb.edu.cn (C.-a. Sun), 570282867@qq.com (Y.M. Zhai), shangyan@live.com (Y. Shang), zhangzy@ios.ac.cn (Z. Zhang).

0950-5849/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.infsof.2013.07.009

and faults in larger programs generally require fewer efforts to be located. For example, the seven extensively used Siemens programs (print_tokens, print_tokens2, replace, schedule, schedule2, tcas, and tot_info [10]) have on average 344 lines of code, while the three Unix programs (flex, grep, and gzip [10]) always have greater than 4000 lines of code. A corresponding observation is that faults in the Siemens programs can be located with an on average 46% code examining effort by Tarantula [17], which means to locate faults in a faulty program, Tarantula needs to examine 46% statements in these Siemens programs. At the same time, on average 25% code examining efforts are needed to locate a fault in the Unix programs. Similar results and analysis can be found in previous studies [10,17,20,22,27,35,39]. Since BPEL programs are often smaller in scale compared with C or Java programs, can the existing fault localization techniques effectively locate faults in BPEL programs? Up to now, there is, at least to our best knowledge, not any answers to this question.

Another key issue of debugging BPEL programs is to understand typical faults possibly made by BPEL designers. Fortunately, Estero-Botaro et al. [12] defined a set of mutation operators for BPEL with respect to its specific features. These mutation operators represent typical faults made by BPEL designers [13]. Boubeta-Puig et al. [6] further compared mutation operators for BPEL with those for the other programming languages to check whether there is any mutation operator missing for BPEL, and to illustrate the differences and similarities between BPEL and the other languages. In our previous work [31], we developed a scenario-oriented testing approach for BPEL programs, and employed the mutation operators proposed in [12] to validate the effectiveness of our approach. As a result, test suites generated using our approach can be used to detect most of seeded faults in BPEL programs.

In our previous work [32], we have investigated how to effectively locate faults in BPEL programs to address the challenges and open questions for debugging BPEL programs. We proposed two fault localization guidelines and synthesized an existing technique, Tarantula, to locate faults in BPEL programs. A preliminary evaluation showed that our approach is feasible and effective.

In this paper, we further address the challenges of locating faults in BPEL programs and intend to answer the open questions through the experiments. First, we propose a BPEL fault localization framework, which formalizes the BPEL fault localization problem using the popular fault localization problem settings. The framework is able to synthesize most existing fault localization techniques. Second, we conducted an empirical study to evaluate the effectiveness of the synthesized techniques for BPEL programs. Two real-life BPEL programs are used as the subjects and mutation operators are employed to simulate typical faults. The experiments validated the feasibility of the framework and evaluated the effectiveness of the synthesized techniques. Consequently, the synthesized techniques can successfully locate more than 50% the seeded faults. Third, the BPEL fault localization guidelines are proposed to locate faults, which are based on the resulting ranked list of program elements generated by the synthesized techniques. In this context, we are interested to know the impact of the fault localization guidelines. Experiment results show that the BPEL fault localization guidelines have positive impacts on improving the effectiveness of the synthesized fault localization techniques.

The contributions of this paper are threefold. (i) We first propose a fault localization framework for BPEL programs, which enables the synthesis of existing fault localization techniques. (ii) We empirically validate the feasibility of our framework, and experimental results show that the synthesized techniques are effective in locating faults in BPEL programs. (iii) We empirically evaluate the impact of the proposed fault localization guidelines on the effectiveness of the synthesized fault localization techniques.

The rest of the paper is organized as follows. Section 2 introduces the underlying concepts of BPEL, fault localization, and mutation analysis. Section 3 proposes a formal fault localization framework and guidelines for BPEL programs. Section 4 describes an empirical study which is used to validate the feasibility of the proposed framework and evaluate the effectiveness of the three synthesized fault localization techniques. Section 5 concludes the paper and proposes the future work.

2. Related work

In this section, we introduce the basic concepts and elements of BPEL programs, general fault localization approaches, BPEL-specific fault location issues, and the BPEL mutation mechanism related to this work.

2.1. Business Process Execution Language (BPEL)

BPEL [23] is an executable service composition language which executes complex business processes by orchestrating Web services. Like typical programming languages, BPEL has standard control structures, such as *sequence*, *switch*, and *while*. However, BPEL programs are significantly different from the traditional programs in the following aspects [32].

First, BPEL provides an explicit integration mechanism (architectural glues) to compose multiple Web services into a large-scale system, while such integration in traditional programs is implicit. Second, Web services composed by a BPEL program may be implemented in different programming languages and deployed in a remote service container, while modules in a traditional program are usually implemented in the same programming language and installed in the same computer. Third, BPEL programs are represented as XML files, and they demonstrate a big difference in syntax when compared with 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.

BPEL programming model is illustrated in Fig. 1. Usually, a BPEL program consists of four sections, namely

- *Partner Link Statements*, which describe the relationship among a BPEL process and invoked Web services.
- *Variable Declaration Statements*, which define input and output messages.
- *Handler Statements*, which declare the handlers when an exception or specific event occurs, and
- *Interaction Statements*, which describe how external Web Services are coordinated to execute a business process. Activities are the basic interaction units of a BPEL process, and are further

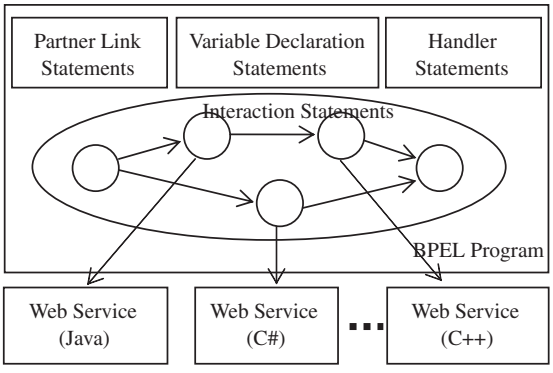

Fig. 1. The BPEL programming model.

divided into basic activities and structural activities. *Basic activities* execute an atomic execution step. *Structural activities* are composites of basic activities and/or structural activities, including *sequence*, *switch*, *while*, *flow*, *pick*, and so on.

2.2. Fault localization

Spectra-based fault localization is a big family of fault localization techniques. A representative one is Tarantula, which counts the executions of program elements in different executions, and uses the ratio of a program element being exercised in a failed execution and the one in a passed execution to calculate the suspiciousness of the program element [17]. Such techniques are empirically evaluated to be effective in previous studies [1,16,20,28]. Naish et al. [22] listed 33 different such techniques. In this paper, we discuss the synthesization of some of these techniques for the fault localization of BPEL programs.

Another trend is to install predicates in programs, and capture and/or sample their execution behaviors to efficiently identify fault-relevant program elements. CBI [19] ranked the predicates according to the probability that the program under study will fail when those predicates are observed to be true. Arumuga Nainar et al. [3] used compound Boolean predicates to locate faults. Zhang et al. [37] investigated the impact of short-circuit rules in the evaluation of Boolean expressions, and propose DES [37] to address it accordingly. HOLMES [8] uses execution paths as fault predicators. Zhang et al. [36] proposed CP to capture the propagation of infected program states via edges of a control flow graph. They located suspicious edges and mapped the suspiciousness of edge to that of basic blocks. Santelices et al. [28] investigated the integration of different methods in locating faults. Zhang et al. [38] proposed a non-parametric predicate-based statistical fault-localization framework, one of which contribution is to set up a fair comparison between statement-level techniques and predicate-based techniques. In this paper, we focus on the fault location of BPEL programs. Since a BPEL program is typically much shorter than a C or Java program, we investigate all the statements, rather than working on predicates.

Different from the above general approaches, some efforts have been devoted to the fault localization of concurrent programs [21]. The sources of concurrency faults are mainly from *data race*, *atomicity violations* and *order violations*, and some methods have been proposed to detect *data races* through static and dynamic analysis techniques [14]. More recent work has tried to identify and detect dynamic interleaving patters that could result in an atomicity violation or order violation [15,25]. The concurrency is one of major features of BPEL, which is supported in the following ways: (i) The *flow* activity allows defining a set of activities, which are executed in parallel. (ii) The *receive* and *pick* activities receive message from an extern partner, and may result in the concurrency when there are more than one activity whose *createInstance* attribute is set to "yes". (iii) The *forEach* activity iterates over a set of child activities, and may result in the concurrency when its *parallel* attribute is set to "yes". At the same time, BPEL provides a mechanism to protect the concurrent access to the global data. This can be done by specifying the *isolated* attribute of the scope to "yes". The existing fault localization techniques for concurrent programs provide an aid to locate concurrency faults of BPEL programs.

Finally, BPEL is an XML-based language, which combines the concepts or terms from several domains, such as Service Oriented Architecture (SOA), Business Process Management (BPM) and data representation. BPEL is simple in syntax, and some tools (such as Eclipse [11]) are available to provide an aid to check the syntax of BPEL programs. On the other hand, BPEL is complex in semantics, and it is difficult to write error-free BPEL programs. One has to check the behaviors specified by the BPEL program. Unfortunately, little work was reported on debugging XML-based programs, although XML are widely used in most areas of software development. Bae and Baily [4] investigated the debugging issue of XSLT transformations. Song and Tilevich [29] proposed a metadata invariant-based method to locate faults in programs with XML-based metadata. To our knowledge, there is not yet one approach devoted to debugging XML-based programs.

2.3. Mutation analysis and BPEL mutation operators

Mutation analysis [2,9] is widely used to assess the adequacy of a test suite and the effectiveness of testing techniques. It applies some mutation operators to seed various faults into the program under test, in order to generate a set of variants, namely mutants. If a test case causes a mutant to show a behavior different from the program under test, the mutant is said to be "killed". An equivalent mutant refers to one whose behaviors are always the same as those of program under test. In this paper, we use mutation analysis technique to generate mimicking faults for measuring the effectiveness of fault localization techniques.

Recently, people investigate the possible faults related to BPEL programs in terms of mutation operators. Estero-Botaro et al. [12] defined 26 mutation operators for BPEL 2.0. These mutation operators are classified into four classes, namely (1) Identifier Mutation Operators, (2) Expression Mutation Operators, (3) Activity Mutation Operators, and (4) Exception and Event Mutation Operators. Boubeta-Puig et al. [6] further compared these mutation operators with the existing mutation operators for the most popular traditional languages, such as C, Fortran, Ada, C++, C#, ASP.NET, Java, SQL and XSLT [6], and discovered that only 13 (50%) of the 26 operators for WS-BPEL 2.0 are available in the traditional languages. This suggests that WS-BPEL 2.0 has many specific types of mistakes when it is used. In this paper, we will employ these mutation operators to simulate possible faults in BPEL programs.

3. The BPEL fault localization framework

In this section, we first discuss the major concerns of debugging BPEL programs. Then, we provide a formal basis for the interactions of BPEL programs. Next, we propose a fault localization framework for BPEL programs and guidelines for improving its effectiveness. Finally, we use an example to demonstrate the proposed framework and guidelines.

3.1. Debugging concerns of BPEL programs

Debugging is a 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.

Since BPEL programs are significantly different from traditional programs in both syntax and semantics, how to effectively locate a fault in BPEL programs is still open. Based on the BPEL programming model in Section 2, we present two of the most important concerns when debugging BPEL programs.

3.1.1. Integration level debugging

As mentioned before, BPEL is a kind of architectural glues which is used to build an executable process by assembling Web services. In the traditional programs, modules (such as functions and classes) are integrated by implicit function invoking, and hence modules are closely coupled. As a result, modules and module integrations are often written in the same programming language. However, 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 faults at the service integration level when debugging BPEL programs.

3.1.2. Interaction debugging

Among four sections of BPEL program as illustrated in Fig. 1, 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 a BPEL program. In this context, we focus on how to effectively locate faults related to the interactions.

When a BPEL program is executed, the Web services under composition will be invoked. In this sense, debugging BPEL programs covers the fault localization at the integration level (namely BPEL code) and at the component level (source code of service implementation). As we discussed, the faults associated with the two levels are significantly different from each other. In this study, we focus on the fault localization at the integration level, and assume that the Web services are correctly implemented. Definitely, some faults may happen to Web services. How to ascertain faults in a specific module or in the integration level is a new challenging and open issue, which is beyond the scope of this paper and hence left for future work.

3.2. Preliminaries

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, and describes the interaction through specifying the activity type, operation name, input variables, output variables, partner link, port type, target link names and source link names. Fig. 2 illustrates an example of the *invoke* statement block.

To simplify the fault localization of BPEL programs, 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 and/or non-atomic statement blocks, including *sequence*,*switch*,*while*,*flow* and*pick*. To make the discussion easy, we 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 statement blocks*, which refer to those ones whose child statement blocks are executed in a sequential order, such as *sequence* activity.
- *Optional statement blocks*, which refer to those ones among whose child statement blocks, only one can be executed, such as *switch*, *if/else/elseif*, and *pick* activity.

```
<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>
```

Fig. 2. An illustration of BPEL statement blocks.

- *Parallel statement blocks*, which refer to those ones whose child statement blocks are executed simultaneously, such as *flow* activity, and
- *Loop statement blocks*, which refer to those ones whose child statement blocks are executed all the time until some conditions are satisfied, such as *while*, *untilWhile* and *forEach* activity.

We next formalize the interactions of BPEL programs.

Definition 1. A BPEL program P is defined as an *atomic*, *sequential*, *optional*, *parallel*, *loop*, or their composites:

$$P ::= (atomic|sequential|optional|parallel|loop)^{+} \tag{1}$$

Definition 2. An atomic statement block *atomic* is defined as a segment of continuous logic statements:

$$\begin{aligned} atomic ::= \{&statement_{i..j}|(i \leqslant j)\wedge \\ &\neg\exists n.((n < i) \wedge (statement_n \prec statement_i) \\ &\wedge(type(statement_n) \in EnumtypeSet))\wedge \\ &\neg\exists m.((j < m) \wedge (statement_j \prec statement_m) \\ &\wedge(type(statement_m) \in EnumtypeSet))\wedge \\ &(\forall k.(i \leqslant k \leqslant j) \wedge (type(statement_k) \notin EnumtypeSet))\} \end{aligned} \tag{2}$$

where $statement_i$ is the ith logical statement in P, $statement_{i..j}$ is a continuous statement segment form logical statement i to j, $statement_x \prec statement_y$ refers to that $statement_y$ can be executed if and only if $statement_x$ is immediately executed. *EnumtypeSet* is a set of types {*sequential*, *optional*, *parallel*, *loop*}.

Definition 3. A sequential block *sequential* is a composite of *atomic* and *no-atomic* statement blocks:

$$\begin{aligned} sequential ::= \{&block_1 \cdots block_m|(m \geqslant 1)\wedge \\ &\forall i((1 \leqslant i \leqslant m-1) \wedge \quad block_i \in \{atomic, optional, parallel, loop\}\wedge \\ &(block_i \prec block_{i+1})) \end{aligned} \tag{4}$$

where $block_i \prec block_{i+1}$ refers to that $block_{i+1}$ is immediately executed if and only if $block_i$ is executed.

Definition 4. An optional block *optional* selects only one block for execution under some condition φ.

$$\begin{aligned} optional ::= &(\varphi?) \prec block_{i..j}|\exists k((i \leqslant k \leqslant j)\wedge \\ &(\varphi \rightarrow block_k)\wedge \\ &block_k \in \{atomic, sequential, optional, loop\} \end{aligned} \tag{5}$$

where φ is a formula whose value is evaluated to be False or True, $\varphi \rightarrow block_k$ refers to that $block_k$ is executed only if φ is evaluated to be True.

Definition 5. A parallel block *parallel* is a set of blocks that are executed simultaneously:

$$\begin{aligned} parallel ::= \{&block_{1..m}|m \geqslant 1 \wedge \forall i,j(1 \leqslant i,j \leqslant m) \\ &\wedge(i \neq j)((block_i||block_j)\wedge \\ &block_i \in \{atomic, sequential, optional, loop\}\wedge \\ &block_j \in \{atomic, sequential, optional, loop\})\} \end{aligned} \tag{6}$$

where $block_i||block_j$ refers to that $block_i$ and $block_j$ may be executed simultaneously.

Definition 6. A loop block *loop* repeats the execution of a block $block_i$ until some condition φ is satisfied:

$$\begin{aligned} loop ::= &((\varphi?)block_i)^{+}| \\ &block_i \in \{atomic, optional, parallel, loop\} \end{aligned} \tag{7}$$

where φ is a formula whose value is evaluated to be False or True.

With such a formal basis, we further propose the fault localization guidelines and the fault localization algorithm for BPEL programs, which is to be discussed later.

3.3. Fault localization guidelines

According to the fault types and semantic specialties, the following guidelines are helpful to accurately locate faults. We formalize these guidelines based on the statement blocks, and then provide their explanations.

[Guideline I]

$$\forall b((b \in P) \wedge (b \text{ is a kind of loop} \vee \text{ b is a kind of optional}) \wedge \forall c((c \in P) \wedge (c \neq b) \wedge (\textit{suspiciousness}(b) \geqslant \textit{suspiciousness}(c))) \rightarrow b.\varphi \quad (8)$$

where P is a BPEL program, $b.\varphi$ refers to the condition of block b.

The above guideline suggests that for statement blocks in an *optional* or *loop* statement block have the highest suspiciousness, one should debug faults at the condition part.

Usually, fault localization techniques can locate faults accurately in the traditional programs. This may not 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. This is why we should check the condition part of *optional* or *loop* statement block provided that its branch statement blocks have the highest suspiciousness.

[Guideline II]

$$\forall b((b \in P) \wedge (b \text{ is a kind of sequential}) \wedge \forall c((c \in P) \wedge (c \neq b) \wedge (\textit{suspiciousness}(b) \geqslant \textit{suspiciousness}(c))) \rightarrow d((d \in b) \wedge \text{not}\exists e((e \in b) \wedge (e \neq d) \wedge (e \prec d)) \quad (9)$$

where $e \prec d$ refers to that d is immediately executed if and only if e is executed.

The above guideline suggests that if a *sequential* statement block has the highest suspiciousness, one should debug faults from the first block inside the *sequential* statement block in a forward way. This is rather natural to suspect the starting statement block in a sequential statement block when a fault happens to the sequential block.

3.4. BPEL fault localization framework

Based on the BPEL programming model and debugging concerns discussed above, we propose a statement block-oriented fault localization framework for BPEL programs as illustrated in Fig. 3.

Next, we propose a fault localization procedure with the framework as follows. Given a BPEL program $bp = \langle s_1, s_2, \ldots, s_n \rangle$, where $s_1, \ldots, s_n$ denote a set of statement blocks. Let $ts = \langle t_1, t_2, \ldots, t_m \rangle$ be a set of test cases, our aim is to find the most suspicious statement block that causes the observed failures. The fault location process with our fault localization framework consists of the following four phases.

- [**Phase 1**]: When a failure f is reported during the execution of a BPEL program bp, we first manage to restore the test suite ts that reveals f.
- [**Phase 2**]: For each test case t in ts, we run bp to capture the coverage status cs of each statement blocks with respect to the execution of t, which is accordingly identified as a "pass" or "fail".

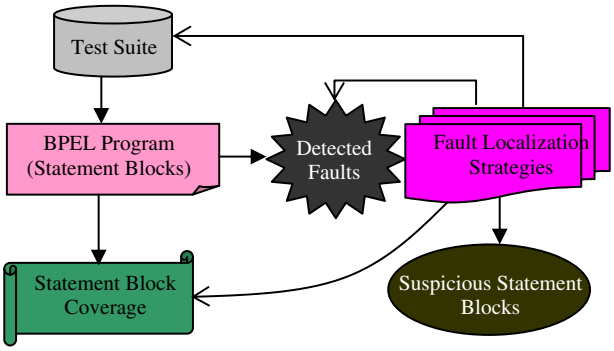

Fig. 3. An illustration of the statement block-oriented fault localization framework for debugging BPEL programs.

- [**Phase 3**]: We use cs as input of an existing fault localization formula r to calculate the suspiciousness scores for each statement block. The most suspicious statement block is termed as *mssb*.
- [**Phase 4**]: According to the type of *mssb*, the possible position set *pps* is recommended by following the fault localization guidelines introduced in Section 3.3.

In the framework, we do not limit the use of different fault localization formulas. However, the effectiveness of our fault localization framework is related to the choice of fault localization formulas. To differentiate the use of various fault localization formulas, we deem an instantiating of the framework based on a given formula as a *synthesized fault localization technique*.

Suppose a fault localization formula r is used in our framework, the fault localization process of the synthesized fault localization technique is described using the algorithm in Fig. 4. The algorithm accepts as input a set of blocks P and a test suite TS, and outputs a possible position set *pps*. Note that the input should also include testing information such as block coverage by each test case and success or failure of P with respect to each test case. To make it simple, such testing information is not represented explicitly. The algorithm first initiates *pps* to empty, then calculates the number of failed test cases *failed* ($block_i$), and the number of passed test cases *failed* ($block_i$), respectively (Step 2). Next, it calculates the total number of failed test cases *totalfailed* and the total number of passed test cases *totolpassed* (Step 3). With the above information available, it calculates the suspiciousness *suspiciousness*($block_i$) using the fault localization formula r (Step 4). Then, it orders the suspiciousness scores for all blocks in P and selects a set of blocks *mssb* whose suspiciousness scores is the largest (Step 5). Here, *getMaxSuspiciousBlock(P)* returns a set of $block_i$ who have the largest suspiciousness scores. Note that the algorithm is able to select a specific ratio of the first largest suspiciousness scores as required. Next, the fault location guidelines are used to recommend the possible position set (Step 6). In detail, for each block $block_i$ in *mssb*, if the type of $block_i$ is *optional* or *loop*, add the position of its condition part to *pps*; if the type of $block_i$ is *sequential*, add the position of the first atomic block to *pps*. Finally, the algorithm returns *pps*.

The proposed framework is very generic and can incorporate a variety of fault localization techniques (algorithms) that have been already developed for typical programs. When these techniques were developed, testing history information is usually expected to be available, such as what test cases are used and what statements are covered. When these fault localization strategies are employed to debug BPEL programs, they must be adapted to the context of statement blocks.

Regarding the localization of concurrency faults in BPEL programs, we need to solely focus on the treatment of *data race*

Fault location algorithm for BPEL programs using the fault location formula r

INPUT:
P:{$block_i$ | 1≤i≤n, and the type of $block_i$ {*atomic*, *sequential*, *optional*, *parallel*, *loop*} }
TS: {t_i| 1≤i≤m, and t_i is a test case of test suite for P}.

OUTPUT:
pps: the *possible position set* of the most suspicious statement blocks.

PROCEDURE
1. Set *pps* to Φ.
2. *FOR EACH* $block_i \in P$,
   failed ($block_i$) ← |{t_i |p(t_i) reveals a failure and $block_i$ is invoked during the execution of p with t_i }| , and
   passed($block_i$) ←|{t_i |p(t_i) passes and $block_i$ is invoked during the execution of p with t_i }|.
   END IF
3. *totalfailed*←|{t_i|p(t_i) reveals a failure}|, and *totolpassed* ←|{t_i |p(t_i) passes}|.
4. *FOR EACH* $block_i \in P$,
   suspiciousness($block_i$) ←*r* (*failed* ($block_i$), *passed*($block_i$), *totalfailed*, *totolpassed*).
   END FOR EACH
5. *mssb*←*getMaxSuspiciousBlock(P)*.
6. *FOR EACH* $block_i$ in *mssb*,
   IF $block_i$ is a kind of *optional* or *loop*
   pps ←*ApplyGuidelineI*($block_i$);
   ELSE IF $block_i$ is a kind of *sequential*
   pps ←*ApplyGuidelineII*($block_i$).
   END IF
   END FOREACH
7. Return *pps*.

END

Fig. 4. The sketch of a fault location algorithm.

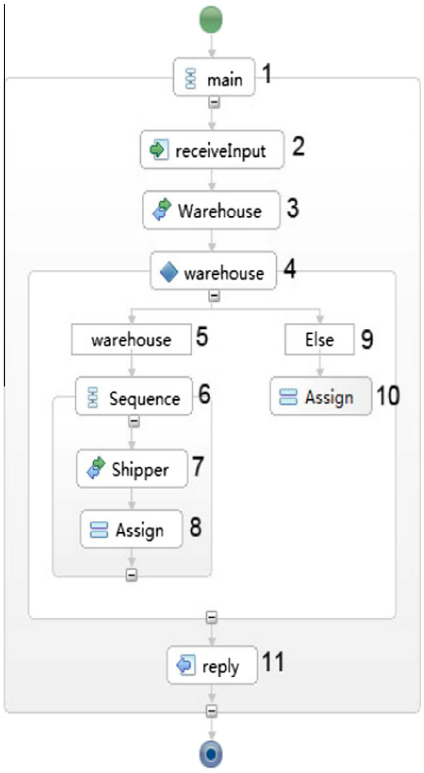

Fig. 5. The BPEL flowchart of the SupplyChain process.

and *order violation*, while *atomicity violation* is often left for a transaction middleware [33]. As to the *data race*, it only happens to global data access without synchronization. As to the *order violation*, it is usually manifested as some conflicting interleaving patterns as summarized in [25]. To locate such faults, we need to further develop techniques to identify and detect the conflicting patters. We do not focus on concurrency fault localization in this paper, though the framework can include it. The reasons include that (i) a BPEL program can be much shorter than a C or Java program, the chance of a BPEL program containing multiple faults is low; (ii) the chance of concurrency faults is very low, which has been observed by the experiments to be reported later. That is, the relevant mutation operators designed for concurrency faults (such as *ASF*, *ACI* and *AFP* proposed in [12]) have a limited applicability, and most of mimicked mutants are equivalent ones.

3.5. Example

In this section, we use an example to demonstrate the proposed framework and guidelines. BPEL program for *SupplyChain* [5] and the Tarantula [17] technique are used for demonstration, and their detailed descriptions can be found in Section 4.

Firstly, this BPEL program involves two Web services and consists of 11 statement blocks. Fig. 5 illustrates its flowchart labeled with the statement block identifiers. Considering statement block 4 which is shown as follows:

```
<condition>$input.name = 'coca' and $input.amount
  &lt; $warehouseAmessage.WarehouseAResponse
</condition>
```

Assume a fault happens to this statement block, where the statement "$input.name = 'coca'" is incorrectly written as "$input.name ! = 'coca'".

Secondly, we have recorded the testing information summarized in Table 1 before debugging the fault. Note that the "statement block ID" column represents a list of 11 statement blocks, the 2nd to 9th column represents the coverage of the statement block with respect to the specific test case (a "●" denotes *covered*; otherwise *not covered*), and the bottom row represents the test result of each test case ("F" denotes*failed*; "T" denotes *passed*). In this context, a test suite of eight test cases was used, and each test case is composed of the*name* and *amount* of goods. For example, the first test case "coca#0" means that the name of goods is "coca" and the amount is zero. These test cases generated using the scenario-oriented approach proposed in [31] can guarantee the branch coverage of BPEL programs.

Thirdly, with the testing information, one can calculate the suspiciousness score of each statement block using the Tarantula formula (refer to detailed discussions in Section 4.3.1). Considering block statement 4, both the *failed* and *total failed* is 4, and both *passed* and *total passed* is 4, so its suspiciousness score can be calculated as follows:

$$suspiciousness(4) = \frac{4/4}{4/4 + 4/4} = \frac{1}{2} = 0.5$$

Similarly, we can calculate the suspiciousness scores of all other statement blocks, which are summarized in the "*suspiciousness*" column in Table 1. From the suspiciousness ranking result of all statement (shown in the "*rank*" column), we observe that statement

Table 1
An illustration of Suspiciousness Calculation and ranking using the Tarantula formula.

Statement block ID	coca#0	coca#12	milk#0	milk#12	coca#1000	coca#1200	milk#1000	milk#1200	Suspiciousness	Rank
1	●	●	●	●	●	●	●	●	0.5	3
2	●	●	●	●	●	●	●	●	0.5	3
3	●	●	●	●	●	●	●	●	0.5	3
4	●	●	●	●	●	●	●	●	0.5	3
5	●	●			●	●	●	●	0.33	8
6	●	●			●	●	●	●	0.33	8
7	●	●			●	●	●	●	0.33	8
8	●	●			●	●	●	●	0.33	8
9			●	●					1.0	1
10			●	●					1.0	1
11	●	●	●	●	●	●	●	●	0.5	3
Test result	F	F	F	F	T	T	T	T	N/A	N/A

blocks 9 and 10 have the highest suspiciousness. So, they are expected as the most suspicious statement blocks (*MSSB*).

Fourthly, we apply the fault localization guidelines to recommend the fault's possible position set (PPS). For this case, statement blocks 9 and 10 are inside an *optional* statement, therefore the condition part (namely statement block 4) should be suspected according to Guideline I. The resulting PPS is {4, 9, 10}, and the fault is successfully located.

4. Experimental evaluation

In this section, we report an empirical study, which is used to validate the feasibility of the proposed framework and evaluate the effectiveness of the synthesized fault localization techniques when they are used for BPEL programs. First, we state the research questions. Second, we describe two subject programs and three subject fault localization techniques that will be examined in this study. Third, we discuss how the experiments are designed and report results according to the research questions. Finally, we discuss observations and threats to the validity of our study.

4.1. Research questions

In this study, we attempt to answer the following questions about the proposed framework and guidelines on locating faults in BPEL programs.

- RQ1: Is the proposed framework able to locate faults in BPEL programs, and how about the effectiveness of existing fault localization techniques when they are synthesized in the framework?

We proposed a formal fault location framework for BPEL programs. The framework and guidelines are illustrated by a sample BPEL program and the Tarantula technique. In this study, we would further examine the feasibility of the framework and the effectiveness of the synthesized fault localization techniques. We selected two real-life BPEL programs as subject programs, and employed mutation operators to simulate possible faults of BPEL programs. The effectiveness is measured in terms of fault localization success rate.

- RQ2: How about the impact of the proposed guidelines on the effectiveness of the synthesized fault localization techniques?

In the proposed framework, the guidelines are proposed to recommend the possible fault locations, which are based on the resulting ranked list generated by the synthesized fault localization techniques. It is interesting to know the impact of the fault localization guidelines. We answer this question by comparing the effectiveness of the synthesized fault localization techniques with and without following the guidelines.

4.2. Subject programs

BPEL programs for *SupplyChain* and *SmartShelf* are chosen as subject programs, and they demonstrate most of major features of BPEL.

(1) SupplyChain

SupplyChain [5] is widely used to demonstrate common features of BPEL. The process receives an order, which is represented by an input message consisting of *name* and *amount* of goods. The process returns an output message to indicate whether the warehouse can accept the order. The process first calls a *Warehouse* web service to get the name and amount of goods available in warehouse. If the required goods (indicated by the *name* of the input message) are available and the required amount (indicated by the *amount* of the input message) is smaller than the amount of goods available in warehouse, the process calls a *Shipper* web service to transport goods, and assign the output message with "yes"; Otherwise, the output message is assigned with "The warehouse cannot receive the order".

(2) SmartShelf

SmartShelf [24] is complex, and it demonstrates some other features of BPEL. For instance, it contains the concurrency behavior. BPEL program for *SmartShelf* involves 14 Web services' interactions and consists of 48 statement blocks. Its flowchart is illustrated in Fig. 6. It receives an input message called *commodity*, which is composed of *name*, *amount* and *status*. The process returns an output message which is composed of *quantity*, *location* and *status*. The whole process is enabled after receiving an input message. It then checks with available shelf items and decides whether the *amount*, *location* and *status* of the available items meet the expected requirements. If the amount of the available goods on shelf is larger than the *amount* of *commodity*, the quantity of message is "Quantity is enough"; Otherwise, it transfers the goods from the warehouse. If the amount of the available goods in the warehouse is larger than the *amount* of *commodity*, the *quantity* of message is "Quantity is enough"; otherwise, the *quantity* of message is "Warehouse quantity is not enough". If the *name* of *commodity* is not the same as the name of the available goods on shelf, it re-arranges the goods and returns "Rearrange is done" as the *location* of message; otherwise it returns "Location is OK". If the *status* of *commodity* is larger than the available *status* of shelf, it sends status to the

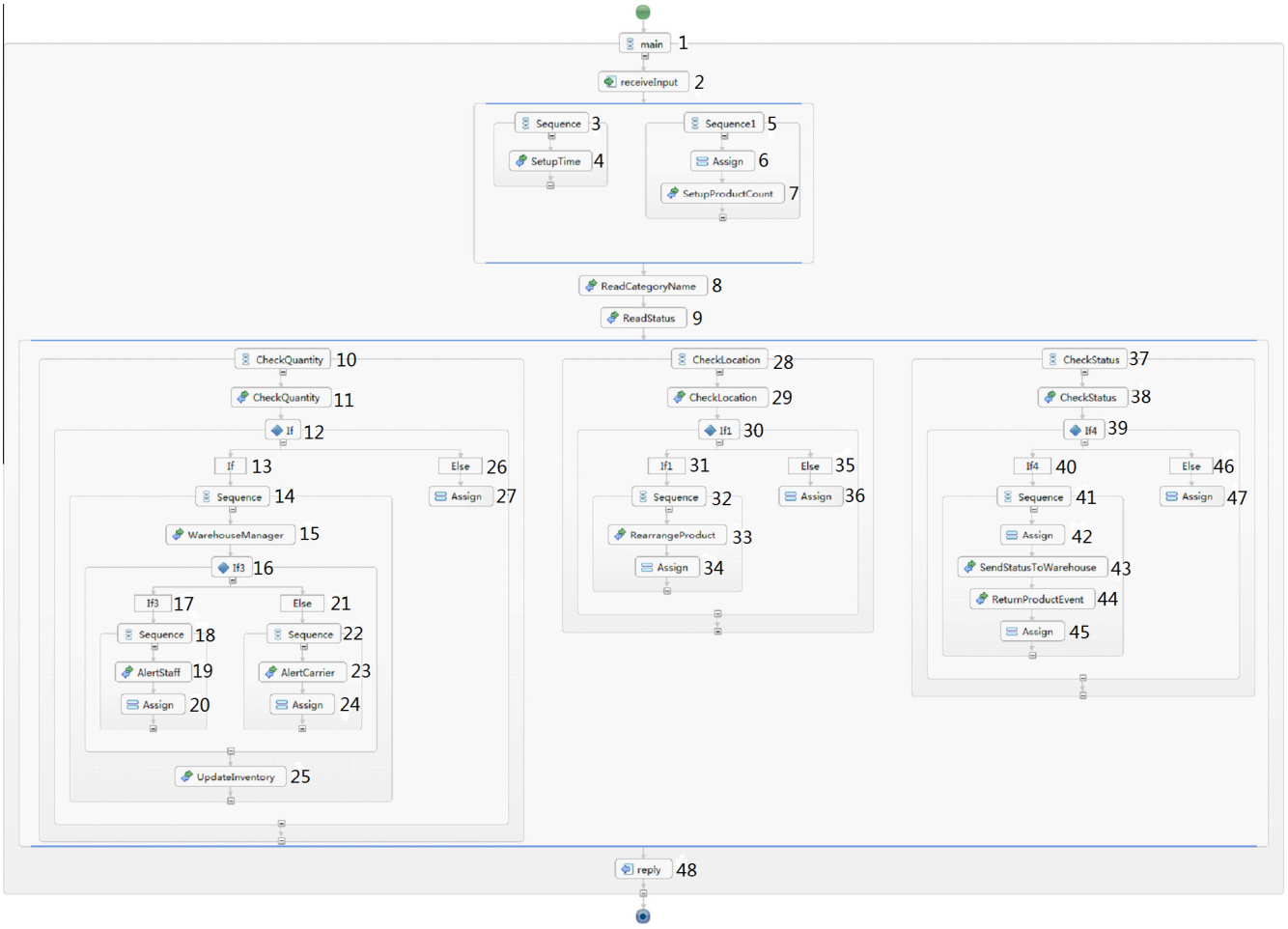

Fig. 6. The BPEL flowchart of the SmartShelf Process.

warehouse and returns "Status is fine now" as *status* of message; otherwise, it returns "Status is ok". The above comparisons are done in parallel.

4.3. Subject techniques

In this section, we first introduce three representative fault localization techniques (namely Tarantula [18], set-union [27] and code coverage [34]), and then discuss their synthesization with the proposed fault localization framework for BPEL programs. We choose Tarantula as a subject technique since it is one of the most representative spectrum-based fault localization techniques. We choose Set-union for comparison because it has the simplest form. We include the third one in the experiment because it is previously evaluated to be very effective [34].

4.3.1. Tarantula

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.

$$suspiciousness(e) = \frac{\frac{failed(e)}{totalfailed}}{\frac{passed(e)}{totalpassed} + \frac{failed(e)}{totalfailed}} \quad (10)$$

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. If any of these denominators equals 0, we assign 0 to the fraction. Using the suspiciousness score, we can compute each entity's likelihood of being faulty whose values range from 0 to 1. For an entity e, if its suspiciousness score is evaluated to be 0, the entity is the least suspicious; if its suspiciousness score is evaluated to be 1, the entity is the most suspicious. As a result, the technique recommends the entities that have the highest suspiciousness scores to check when debugging faults.

4.3.2. Set-union

The set-union technique recommends the suspicious statements by means of the set computation technique. The technique first constructs a set of statements that are executed by one "failed" test case. It then removes the union of all statements executed by all "passed" test cases from the set, finally the remaining statements are recommended as the suspicious statements. The construction of suspicious statements is defined by the following equation.

$$E_{initial} = E_f - \bigcup_{p \in P} E_p \quad (11)$$

where p is one "passed" test case, P is a set of "passed" test cases in the test suite, E_p is a set of statements executed by the "passed" test case p, and E_f is a set of statements executed by the "failed" test case f.

4.3.3. Code coverage

The intuition behind this technique is that the statements executed by more "failed" test cases are more likely to be faulty; if the statements are executed by "passed" test cases, their likelihood of being faulty should be reduced, and the impact of the successive "passed" test cases should be gradually weakened. The intuition can be described as three heuristics. The comprehensive heuristic for calculating the likelihood of the j^{th} statement being faulty is described as follows.

$$\sum_{i=1}^{N} C_{i,j} \times r_i - f\left(\sum_{i=1}^{N} C_{i,j} \times (1 - r_i)\right) \tag{12}$$

Here, if the ith test case executes the jth statement,$C_{i,j} = 1$; Otherwise,$C_{i,j} = 0$. If the ith test case detects faults, $r_i = 1$; otherwise, $r_i = 0$. N is the number of test cases in the test suite. $f(k)$ is defined as follows.

$$f(k) = \begin{cases} k; & k = 0, 1, 2 \\ 2 + (k-2) \times 0.1; & 3 \leqslant k \leqslant 10 \\ 2.8 + (k-10) \times \alpha; & k \geqslant 11 \end{cases} \tag{13}$$

where α is a small number.

4.3.4. Synthesization with the proposed framework

Although all these techniques were developed for typical programs (such as C, C++, or Java programs), which are usually composed of statements, they can be employed to locate faults in BPEL programs. This is because all testing information required by these techniques is available in our fault localization framework as illustrated in Fig. 3. Such required testing information includes "passed" test cases, "failed" test cases, and the coverage information about which entities are executed by "failed" or "passed" test cases. However, some adaptations are necessary when they are used to locate faults in BPEL programs. First, the calculation of suspiciousness must be with respect to a statement block rather than a single statement. This is because the proposed framework for BPEL programs is based on statement blocks. Second, the recommendation of the fault's possible position should follow the proposed guidelines. We have examined the synthesization of Tarantula, when it was used to illustrate the framework to locate faults in BPEL programs (in Section 3.4). The synthesizations of *set-union* and *code coverage* are similar except that different formulas are used.

4.4. Experiment design

4.4.1. Mutant generation

To compare the effectiveness of the three synthesized techniques when they are used to locate faults in BPEL programs, we first seed some faults into two subject BPEL programs. Among the 26 mutation operators proposed in [12], only four of them are applicable to the original subject programs, and hence selected to generate the mimicking faults. In order to cover as many as possible fault types, we have changed the implementation of the subject programs (thereinafter called variants) to derive extra mutants, and these extra mutants except equivalent ones are also included for evaluation (shown at the bottom of evaluation results). In addition, all these faults have been manually seeded into BPEL programs because up to now there is not yet an automatic and practical mutation system for this task.

Table 2 provides a summary of the resulting mutants for *SupplyChain*. The left three columns describe the identifier, the type of mutation operator, and the mutation operation of a mutant, respectively. In this case, ten types of mutation operators are covered, namely *ERR*, which refers to "replacing a relational operator by another of the same type", *ELL*, which refers to "replacing a logical operator by one of the same type", *ASF*, which refers to "replacing a sequence by a flow activity", *AIE*, which refers to "removing an else if element or an else element of an activity", *EAA*, which refers to "replacing an arithmetic operator by another of the same type", *EUU*, which refers to "removing the unary minus operator from an expression", *ECN*, which refers to "replacing the numeric constant by one of the same type", *ISV*, which refers to "replacing a variable identifier by another of the same type", *EMD*, which refers to "replacing a duration expression by 0 or half of the initial value", and *XTF*, which refers to "replacing the *faultname* attribute in the *reply* activity". For the original BPEL program of the SupplyChain, four types of mutation operators are applicable, and accordingly mutants 1 to 13 are generated. For the BPEL program variants, ten mutants (namely mutants 1′ to 10′) are generated. Among them, mutants 7′ to 10′ are equivalent ones and hence

Table 2
Description of the resulting Mutants for SupplyChain.

No.	Type	Description
1	ERR	replace "=" in "$input.name = 'coca'" with "!="
2	ERR	replace "=" in "$input.name = 'coca'" with "<"
3	ERR	replace "=" in "$input.name = 'coca'" with "<="
4	ERR	replace "=" in "$input.name = 'coca'" with ">"
5	ERR	replace "=" in "$input.name = 'coca'" with ">="
6	ERR	replace "<" in "$input.amount < $warehouseAmessage.WarehouseAResponse" with "<="
7	ERR	replace "<" in "$input.amount < $warehouseAmessage.WarehouseAResponse" with ">"
8	ERR	replace "<" in "$input.amount < $warehouseAmessage.WarehouseAResponse" with ">="
9	ERR	replace "<" in "$input.amount < $warehouseAmessage.WarehouseAResponse" with "="
10	ERR	replace "<" in "$input.amount < $warehouseAmessage.WarehouseAResponse" with "!="
11	ELL	replace "and" in "$input.name = 'coca' and $input.amount < $warehouseAmessage.WarehouseAResponse" with "or"
12	ASF	replace a sequence with a flow activity
13	AIE	remove "else" element in "warehouse" activity
1′	EAA	replace "$input.amount + 500" with "$input.amount∗500"
2′	EAA	replace "$input.amount + 500" with "$input.amount div 500"
3′	EAA	replace "$input.amount + 500" with "$input.amount mod 500"
4′	EUU	remove the unary minus operator in "-$warehouseAmessage.WarehouseAResponse"
5′	ECN	replace "$input.amount + 500.00" with "$input.amount + 50.000"
6′	ECN	replace "$input.amount + 500.00" with "$input.amount + 5000.0"
7′	ISV	replace "input" with the same type "order"
8′	EMD	repalce a duration expression with 0
9′	EMD	replace a duration expression with half
10′	XTF	replace " faultName = "missingReply"" with "faultName = " missingRequest"

are excluded for evaluation. Note that for each mutant, only one fault is seeded.

Table 3 shows the resulting mutants for SmartShelf. For the original BPEL program of SmartShelf, only two typical mutation operators are applicable, namely *ERR*, which refers to "replacing a relational operator by another of the same type" and *AIE*, which refers to "removing an else if element or an else element of an activity". To cover concurrency faults, we applied another two mutation operators to the BPEL program variants, namely *AFP* which refers to "changing the value of the *parallel* attribute in the *forEach* activity from 'no' to 'yes'" and *AIS* which refers to "changing the value of the *isolated* attribute of a scope from 'yes' to 'no'". Accordingly, two mutants (namely mutant 1′ and mutant 2′) are generated. However, these two mutants are equivalent ones and hence excluded for evaluation.

4.4.2. Test case generation

To apply the above synthesized fault localization techniques, 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 [31] 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 input 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.

4.4.3. Data collection

During the tests, we need to know which statement blocks are executed. This can be done by writing IDs of covered statement blocks into a log file, and analyzing these block IDs after the tests. In our experiments, we developed scripts to track and analyze the execution of BPEL programs with respect to each test case. 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; otherwise, it is said to be a "failed" test case.

4.4.4. Effectiveness metrics

With the above test history information, we now can employ three synthesized techniques to locate faults detected at the testing stage. Since a BPEL program is often smaller in scale compared with a conventional C or Java program, we tend to use a more strict effectiveness metrics, rather than the popular *expense* [17,35]. The effectiveness of a synthesized fault localization technique *s* is evaluated as follows.

> "The techniques is effective in locating a fault f if the generated pps contains f; otherwise, it is ineffective."

That is, for each fault (mutant), we figure out the most suspicious statement blocks. If the actual fault (namely mutation) occurs to these statement blocks, the fault localization is successful; otherwise, it failed. Finally, we use the average fault localization success to measure the effectiveness of the fault localization techniques.

4.5. Results analysis

4.5.1. Effectiveness of the Synthesized Techniques

In this section, we report the effectiveness of three synthesized techniques when they are used to locate faults in *SupplyChain* and *SmartShelf* BPEL programs.

(1) Tarantula

When the Tarantula technique is used to locate faults in *SupplyChain* and *SmartShelf* BPEL programs, its evaluation results are summarized in Tables 4 and 5, respectively. Note that *No* refers to mutants described in Table 5 and 5, *NE* refers to "Number of test cases whose output is the same as the Expected output", *NNE* refers to "Number of test cases whose output is Not same as Expected output", *MSSB* refers to "Most Suspicious Statement Block", *PPS* refers to "fault's Possible Position Set", and *LF* refers to "whether Locate the Fault" ("*Y*" means success, while "*N*" means failure).

From Tables 4 and 5, we observe that

- When the Tarantula technique is used to locate faults in the original *SupplyChain* BPEL program, it can successfully locate 7 of 13 faults. The correctness percentage of fault localization is

Table 3
Description of the resulting Mutants for smartshelf.

No.	Type	Description
1	ERR	replace "<" in "$quantity.CheckQuantity < $commodity.amount" with ">"
2	ERR	replace "<" in "$quantity.CheckQuantity < $commodity.amount" with "<="
3	ERR	replace "<" in "$quantity.CheckQuantity < $commodity.amount" with ">="
4	ERR	replace "<" in "$quantity.CheckQuantity < $commodity.amount" with "="
5	ERR	replace "$lt;" in "$quantity.CheckQuantity < $commodity.amount" with "!="
6	ERR	replace "<" in "$warehouse.supply > $commodity.amount" with ">"
7	ERR	replace "<" in "$warehouse.supply > $commodity.amount" with "<="
8	ERR	replace "<" in "$warehouse.supply > $commodity.amount" with ">="
9	ERR	replace "<" in "$warehouse.supply > $commodity.amount" with "="
10	ERR	replace "<" in "$warehouse.supply > $commodity.amount" with "!="
11	ERR	replace "!=" in "$location.CheckLocation ! = $categorynameinformation.categoryName" with "="
12	ERR	replace "<" in "$status.CheckStatus < $statusinformation.status" with ">"
13	ERR	replace "<" in "$status.CheckStatus < $statusinformation.status" with "<="
14	ERR	replace "<" in "$status.CheckStatus < $statusinformation.status" with ">="
15	ERR	replace "<" in "$status.CheckStatus < $statusinformation.status" with "="
16	ERR	replace "<" in "$status.CheckStatus < $statusinformation.status" with "!="
17	AIE	remove "else" element of "IF3" activity
18	AIE	remove "else" element of "IF" activity
19	AIE	remove "else" element of "IF1" activity
20	AIE	remove "else" element of "IF4" activity
1′	AFP	replace "forEach parallel = "no"" with "forEach parallel = "yes""
2′	AIS	replace "scope isolated = "yes"" with "scope isolated = "no""

Table 4
Evaluation results of of Tarantula technique for SupplyChain.

NO	NE	NNE	MSSB	PPS	LF
1	20	20	9–10	{4, 9, 10}	Y
2	30	10	1–8, 11	{1, 2, 11}	N
3	30	10	1–8, 11	{1, 2, 11}	N
4	30	10	1–8, 11	{1, 2, 11}	N
5	30	10	1–8, 11	{1, 2, 11}	N
6	39	1	9–10	{4, 9, 10}	Y
7	21	19	9–10	{4, 9, 10}	Y
8	20	20	9–10	{4, 9, 10}	Y
9	29	11	9–10	{4, 9, 10}	Y
10	31	9	9–10	{4, 9, 10}	Y
11	20	20	9–10	{4, 9, 10}	Y
12	30	10	1–12	{1–12}	N
13	10	30	1–12	{1–12}	N
1′	37	3	5–8	{4, 5}	Y
2′	24	16	9–10	{4, 9}	Y
3′	24	16	9–10	{4, 9}	Y
4′	21	19	9–10	{4, 9}	Y
5′	36	4	9–10	{4, 9}	Y
6′	36	4	1–8, 11	{1, 2}	N

Table 5
Evaluation results of Tarantula technique for SmartShelf.

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

53.8%. For *SupplyChain* BPEL program variants, it can successfully locate 5 of 6 faults. The correctness percentage of fault localization is 83.33%. For the *SmartShelf*, it can successfully locate 10 of 20 faults. The correctness percentage of fault localization is 50%.

- When the Tarantula technique is used for*SupplyChain,* its effectiveness is evidently higher than that for *SmartShelf*.
- There is not an evident clue showing the effectiveness restriction of the Tarantula technique with respect to some types of faults. For instance, as for the *AIE* faults, it failed to locate all the faults in both *SmartShelf* and *SupplyChain* BPEL programs. As for the *ERR* faults, it located some of them in both *SmartShelf* and *SupplyChain* BPEL programs.

(2) Set-union

When the set-union technique is used to locate faults in *SupplyChain* and *SmartShelf* BPEL programs, its effectiveness depends on the selection of E_f. In our experiments, its effectiveness is evaluated to 0% in the worst case for both two subject programs. Its best evaluation results are summarized in Tables 6 and 7, respectively.

From Tables 6 and 7, we observe that

Table 6
Evaluation results of the set-union technique for SupplyChain.

NO	NE	NNE	MSSB	PPS	LF
1	20	20	9–10	{4, 9, 10}	Y
2	30	10	∅	{∅}	N
3	30	10	∅	{∅}	N
4	30	10	∅	{∅}	N
5	30	10	∅	{∅}	N
6	39	1	9–10	{4, 9, 10}	Y
7	21	19	9–10	{4, 9, 10}	Y
8	20	20	9–10	{4, 9, 10}	Y
9	29	11	9–10	{4, 9, 10}	Y
10	31	9	9–10	{4, 9, 10}	Y
11	20	20	9–10	{4, 9, 10}	Y
12	30	10	∅	{∅}	N
13	10	30	∅	{∅}	N
1′	37	3	5–8	{4, 5}	Y
2′	24	16	9–10	{4, 9}	Y
3′	24	16	9–10	{4, 9}	Y
4′	21	19	9–10	{4, 9}	Y
5′	36	4	9–10	{4, 9}	Y
6′	36	4	∅	{∅}	N

Table 7
Evaluation results of set-union technique for SmartShelf.

NO	NE	NNE	MSSB	PPS	LF
1	4	116	13–15, 21–25	{12, 13}	Y
2	116	4	∅	{∅}	N
3	0	120	∅	{∅}	N
4	36	84	13–15, 21–25	{12, 13}	Y
5	84	36	13–15, 21–25	{12, 13}	Y
6	44	76	17–20	{16, 17}	Y
7	116	4	17–20	{16, 17}	Y
8	40	80	13–20	{12, 13}	N
9	76	44	17–20	{16, 17}	Y
10	84	36	∅	{∅}	N
11	0	120	∅	{∅}	N
12	6	114	40–45	{39, 40}	Y
13	114	6	∅	{∅}	N
14	0	120	∅	{∅}	N
15	54	66	40–45	{39, 40}	Y
16	66	54	∅	{∅}	N
17	80	40	∅	{∅}	N
18	80	40	∅	{∅}	N
19	60	60	∅	{∅}	N
20	60	60	∅	{∅}	N

- When the set-union technique is used to locate faults in the original *SupplyChain*BPEL program, it can successfully locate at most 7 of 13 faults. The correctness percentage of fault localization is 53.8%. For *SupplyChain*BPEL program variants, it can successfully locate 5 from 6 faults. The correctness percentage of fault localization is 83.33%.
- When the set-union technique is used to locate faults in the *SmartShelf*BPEL program, it can successfully locate at most 8 of 20 faults. The correctness percentage of fault localization is 40%.
- For *SupplyChain*, the best effectiveness of the set-union technique is the same to the one of the Tarantula technique. For *SmartShelf*, the best effectiveness of the set-union technique is evidently worse than that of the Tarantula technique. More interestingly, in the case of *SupplyChain* the faults that cannot be successfully located by the set-union technique subsume the ones that cannot be successfully located by the Tarantula technique, while this is not true with the case of *SmartShelf*.

(3) Code coverage

Table 8
Evaluation results of the code coverage technique For SupplyChain.

NO	NE	NNE	MSSB	PPS	LF
1	20	20	1–4, 11	{1, 2, 11}	N
2	30	10	1–8, 11	{1, 2, 11}	N
3	30	10	1–8, 11	{1, 2, 11}	N
4	30	10	1–8, 11	{1, 2, 11}	N
5	30	10	1–8, 11	{1, 2, 11}	N
6	39	1	9–10	{4, 9, 10}	Y
7	21	19	1–4, 11	{1, 2, 11}	N
8	20	20	1–4, 11	{1, 2, 11}	N
9	29	11	1–4, 11	{1, 2, 11}	N
10	31	9	9–10	{4, 9, 10}	Y
11	20	20	9–10	{4, 9, 10}	Y
12	30	10	9–10	{4, 9, 10}	Y
13	10	30	5–8	{4, 5}	Y
1′	37	3	5–8	{4, 5}	Y
2′	24	16	9–10	{4, 9}	Y
3′	24	16	9–10	{4, 9}	Y
4′	21	19	1–4, 11	{1, 2}	N
5′	36	4	9–10	{4, 9}	Y
6′	36	4	1–8, 11	{1, 2}	N

When the code coverage technique is used to locate faults in *SupplyChain* and *SmartShelf* BPEL programs, its evaluation results are summarized in Tables 8 and 9, respectively. The parameter α here is set to 0.001, as suggested in [27].

From Tables 8 and 9, we observe that

- When the code coverage technique is used to locate faults in the original *SupplyChain* BPEL program, it can successfully locate 5 of 13 faults. The correctness percentage of fault localization is 38.5%. For *SupplyChain* BPEL program variants, it can successfully locate 4 of 6 faults. The correctness percentage of fault localization is 66.67%.
- When the code coverage technique is used to locate faults in the *SmartShelf* BPEL program, it can successfully locate 8 of 20 faults. The correctness percentage of fault localization is 40%.

The parameter α in the code coverage technique may have an impact on the effectiveness of the technique. To investigate such an impact, we further evaluate the effectiveness of the code coverage technique with the changing values of the parameter α. Tables 10 and 11 summarize the evaluation results of the code coverage technique for *SupplyChain* and *SmartShelf*, respectively.

Table 9
Evaluation results of the code coverage technique for SmartShelf.

NO	NE	NNE	MSSB	PPS	LF
1	4	116	1–12, 28–30, 37–39	{1, 28, 37}	N
2	116	4	21–24	{16, 21}	N
3	0	120	1–12, 28–30, 37–39	{1, 28, 37}	N
4	36	84	1–12, 28–30, 37–39	{1, 28, 37}	N
5	84	36	21–24	{16, 21}	N
6	44	76	13–16	{12, 13}	N
7	116	4	21–24	{16, 21}	Y
8	40	80	13–16	{12, 13}	N
9	76	44	13–16	{12, 13}	N
10	84	36	17–20	{16, 17}	Y
11	0	120	1–12, 28–30, 37–39	{1, 28, 37}	N
12	6	114	1–12, 28–30, 37–40	{1, 28, 37}	N
13	114	6	40–45	{39, 40}	Y
14	0	120	1–12, 28–30, 37–40	{1, 28, 37}	N
15	54	66	1–12, 28–30, 37–41	{1, 28, 37}	N
16	66	54	40–45	{39, 40}	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

Table 10
A summary of the effectiveness of the code coverage technique for SupplyChain.

Parameter	Number of total faults	Number of located faults	Number of un-located faults	Correctness percentage (%)
$\alpha = 0.1$	13	5	8	38.5
$\alpha = 0.05$	13	5	8	38.5
$\alpha = 0.01$	13	5	8	38.5
$\alpha = 0.005$	13	5	8	38.5
$\alpha = 0.001$	13	5	8	38.5
$\alpha = 0.1$	6	5	1	83.33
$\alpha = 0.05$	6	5	1	83.33
$\alpha = 0.01$	6	4	2	66.67
$\alpha = 0.005$	6	4	2	66.67
$\alpha = 0.001$	6	4	2	66.67

From Tables 10 and 11, we observe that

- For the original *SupplyChain* BPEL program, the effectiveness of the code coverage technique keeps stable, and the correctness percentage of fault localization is always 38.5%. While for *SupplyChain* BPEL program variants, the correctness percentage of fault localization is from 66.67% to 83.33%. For *SmartShelf*, the correctness percentage of fault localization is always 40%.
- The parameter α does not affect the effectiveness of the code coverage technique when it was used for the two original BPEL programs, while α does affect its effectiveness when it was used for *SupplyChain* BPEL program variants.

4.5.2. Impact of the fault localization guidelines

In order to know the impact of the fault localization guidelines on the effectiveness of the synthesized techniques, we further compare their effectiveness when they are used with and without following the fault localization guidelines. The comparisons are summarized in Table 12.

From Table 12, we observe that

- The fault localization guidelines significantly affect the effectiveness of the synthesized techniques. When the guidelines are not followed, all the synthesized techniques cannot successfully locate faults in two subject programs (namely their effectiveness is 0%). However, their effectiveness increases significantly when the fault localization guidelines are followed. These results confirm the value of the guidelines we proposed with respect to specific features of BPEL programs.

One may be curious about why the effectiveness of the synthesized fault localization techniques is 0%. Recall mutant generation in Section 4.4.1, among mutation operators proposed in [12], only four ones are applicable to original *SupplyChain* subject program, and two ones are applicable to *SmartShelf* subject program. By looking into the BPEL programs, we discover that all simulated faults are seeded into the condition parts. This means that if the fault localization guidelines are not followed, those statement

Table 11
A Summary of the effectiveness of The Code Coverage Technique for SmartShelf.

Parameter	Number of total faults	Number of located faults	Number of un-located faults	Correctness percentage (%)
$\alpha = 0.1$	20	8	12	40
$\alpha = 0.05$	20	8	12	40
$\alpha = 0.01$	20	8	12	40
$\alpha = 0.005$	20	8	12	40
$\alpha = 0.001$	20	8	12	40

Table 12
Comparison of effectiveness of three synthesized techniques with and without following the fault localization guidelines.

Techniques	Subject programs	Effectiveness	
		Without guidelines (%)	With guidelines (%)
Tarantula	SupplyChain	0	53.8
	SmartShelf	0	50
Set-union	SupplyChain	0	53.8
	SmartShelf	0	40
Code coverage ($\alpha = 0.001$)	SupplyChain	0	38.5
	SmartShelf	0	40

blocks inside a *loop* or *optional* block are recommended as the most suspicious statement blocks. This results in the incorrectness.

4.6. Summary and observations

Through this empirical study, we have validated the feasibility of the proposed fault localization framework for BPEL programs. We also evaluated the effectiveness of three synthesized fault localization techniques. From the evaluation, we have the following observations:

- Among three synthesized fault localization techniques, the Tarantula technique has the best effectiveness and can locate the largest number of seeded faults for both subject programs. However, this technique may be inapplicable in some situation where the test suite only contains either all "passed" test cases or all "failed" test cases. If this happens, the technique cannot differentiate the suspicious statement blocks.
- Compared with the Tarantula technique, the effectiveness of the set-union technique is lower. This technique also subjects to several disadvantages. First, it is hard to be automated and requires more manual decisions. Second, the effectiveness of this technique is heavily influenced by the selection of "failed" test cases, which is crucial to the construction of suspicious statement blocks. Third, it is possibly inapplicable, that is, it may result in an empty set of suspicious statement blocks if "passed" test cases in the test suite drive all execution paths of the selected "failed" test case.
- Compared with the Tarantula technique, the effectiveness of the code coverage technique is also lower. This observation is very interesting because the code coverage technique has a very good effectiveness when it is employed to typical programs (such C or Java) [34]. The reason behind this observation can be explained as follows. On one hand, faults in BPEL programs are different from that in typical programs in that the control logic in the former (i.e. dealing with the simple and high-level business logics) seems simpler than the one in the latter (i.e. dealing with the very complex and low-level calculation logic). In this context, the ratio of "failed" test cases against "passed" test cases is not a very tiny number (such as 0.001). On the other hand, the code coverage technique may have a good effectiveness when the ratio of "failed" test cases against "passed" test cases is very tiny (such as 0.001). This observation further indicates that the code coverage technique is more suitable for locating those faults that are hard to detect. Compared with the set-union technique, the code coverage technique is easier to be automated.

Besides the above observations, we also observed that when the mutation technique is used for BPEL programs, the chance of derived mutants being equivalent ones is larger than that for the traditional programs (such as C or Java). This hereby restricted the number of non-equivalent mutants for evaluation when the mutation technique was used to simulate mimicking faults in our experiments.

4.7. Threats to validity

In our empirical study, the feasibility of the proposed fault localization framework has been validated through two subject programs and three representative fault localization techniques. However, the validity of effectiveness observed from the controlled experiments may suffer several threats.

- *Internal threats*: In our study, the faults of BPEL programs were mimicked by means of mutation operators. Although mutation analysis has been widely used to evaluate the effectiveness of various fault localization techniques, the mimicked faults are possibly different from the real-life faults. Furthermore, the occurrences of different types of real-life faults are varying, while faults mimicked by means of mutation operators were randomly generated. Such difference may result in a deviation of the effectiveness result of the synthesized fault localization techniques when they are employed in practice.
- *External threats*: In our study, although we attempted to include more types of mutation operators, only 12 mutation operator types were applicable to subject programs or their variants, and 39 mutants were generated for experiments. The limited number of mutants and applicable mutation operators may threat the observations on the effectiveness of the synthesized fault localization techniques.
- *Conclusion threats*: The effectiveness of the synthesized fault localization techniques would be more convincing if more subject programs were used for evaluation. Unfortunately, it is difficult to include a large number of BPEL programs for the experimental evaluation. Although some literature does mention open source code BPEL programs, we downloaded them and found that they are not useful because no actual business logic is really implemented in these BPEL programs. The limited number of subject programs may threat the effectiveness observed in this study.

To address the threats discussed above, the key is to conduct the effectiveness evaluation based on a benchmark of BPEL programs and associated faults, which can manifest most features and faults of practical BPEL programs. Currently, such a benchmark is absent, which calls for further efforts of the BPEL research community. On the other hand, the threats above do not prevent the proposed framework from adoption in any scenarios where BPEL is used. Furthermore, there is not a limitation on the size of BPEL programs when the synthesized fault location techniques are used.

5. Conclusions and future work

We have proposed an effective fault localization framework to address the challenges of debugging BPEL programs. Unlike the traditional framework where the basic localization unit is single executable statements, the proposed framework is based on statement blocks. Such a framework is devoted to BPEL programs which demonstrate features of both typical programming languages and architectural gluing languages. This framework is capable of synthesizing typical fault localization techniques that are not developed for BPEL programs, such as the Tarantula techniques, the set-union technique, and the code coverage technique.

We have also conducted an empirical study which is used to validate the feasibility of the proposed framework and evaluate the effectiveness of these synthesized fault localization techniques

when they are used for BPEL programs. We employed mutation operators to simulate mimicking faults of two realistic BPEL programs. The results of our empirical study demonstrate the feasibility of the proposed framework, and also show that among the three fault-localization techniques, the Tarantula technique has the best effectiveness and can locate the largest number of seeded faults in BPEL programs. This observation further indicates that the Tarantula technique should be the best choice that can be used to locate faults of BPEL programs although it was not originally developed for BPEL programs.

In our future work, we plan to develop more efficient fault localization techniques based on the observations reported in this work, and implement an automatic mutation system for BPEL. We also want to involve more BPEL subject programs and types of faults by means of mutation operators to evaluate the effectiveness of more fault localization techniques.

Acknowledgments

Authors thank Tieheng Xue and Ke Wang 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 Nos. 60903003, 61003027), 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 Sciences (Grant No. SYSKF1105), the Beijing Municipal Training Program for Excellent Talents (Grant No. 2012D009006000002), and the National Science and Technology Major Project of the Ministry of Science and Technology of China (Grant No. 2012ZX01039-004). Thanks to the anonymous reviewers who provided useful suggestions on earlier versions of this paper.

References

[1] R. Abreu, P. Zoeteweij, A.J.C. van Gemund, On the accuracy of spectrum-based fault localization, in: Proceedings of the Testing: Academic and Industrial Conference: Practice and Research Techniques (TAICPART-MUTATION 2007), IEEE Computer Society, 2007, pp. 89–98.

[2] J.H. Andrews, L.C. Briand, Y. Labiche, Is mutation an appropriate tool for testing experiments?, in: Proceedings of the 27th International Conference on Software Engineering (ICSE 2005), 2005, pp. 402–411.

[3] P. Arumuga Nainar, T. Chen, J. Rosin, B. Liblit, Statistical debugging using compound Boolean predicates, in: Proceedings of the 2007 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2007), ACM, 2007, pp. 5–15.

[4] E. Bae, J. Bailey, CodeX: An Approach for Debugging XSLT Transformation, in: Proceedings of the 4th International Conference on Web Information Systems Engineering (WISE 2003), 2003, pp. 309–312.

[5] L. Baresi, R. Heckel, S. Thöne, 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.

[6] J. Boubeta-Puig, A. García-Domínguez, I. Medina-Bulo, Analogies and Differences between Mutation Operators for WS-BPEL 2.0 and Other Languages, in: Proceedings of 6th International Workshop on Mutation, Analysis, 2011, pp. 398–407.

[7] G. Canfora, M. Di Penta, Service Oriented Architecture Testing: A Survey, LNCS 5413, Springer, 2009, pp. 78–105.

[8] T.M. Chilimbi, B. Liblit, K. Mehra, A.V. Nori, K. Vaswani, HOLMES: effective statistical debugging via efficient path profiling, in: Proceedings of the 31st International Conference on Software Engineering (ICSE 2009). IEEE Computer Society; 2009, pp. 34–44.

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

[10] H. Do, S.G. Elbaum, G. Rothermel, Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact, Empirical Software Engineering 10 (4) (2005) 405–435.

[11] Eclipse, 2013. <www.eclipse.org>.

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

[13] A. Estero-Botaro, F. Palomo-Lozano, I. Medina-Bulo, Quantitative Evaluation of Mutation Operators for WS-BPEL Compositions, in: Proceedings of Third International Conference on Software Testing, Verification, and Validation Workshops, 2010, pp. 142–150.

[14] C. Flanagan, S.N. Freund, Type-based Race Detection for Java, in: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI 2000), 2000, pp. 219–232.

[15] C. Hammer, J. Dolby, M. Vaziri, F. Tip, Dynamic detection of aotmic-set-serializability violations, in: Proceedings of the 30th International Conference on Software Engineering (ICSE 2008), 2008, pp. 231–240.

[16] D. Jeffrey, N. Gupta, R. Gupta, Fault localization using value replacement, in: Proceedings of the 2008 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008), ACM, 2008, pp. 167–178.

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

[18] J.A. Jones, M.J. Harrold, J. Stasko, Visualization of test information to assist fault localization, in: Proceedings of the 24th International Conference on Software Engineering (ICSE 2002), 2002, pp. 467–477.

[19] B. Liblit, M. Naik, A.X. Zheng, A. Aiken, M.I. Jordan, Scalable statistical bug isolation, in: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2005), ACM, 2005, pp. 15–26.

[20] C. Liu, L. Fei, X. Yan, S.P. Midkiff, J. Han, Statistical debugging: a hypothesis testing-based approach, IEEE Transactions on Software Engineering 32 (10) (2006) 831–848.

[21] S. Lu, S. Park, E. Seo, Y. Zhou, Learming from mistakes: a Comprehensive Study on Real World Concurrency Bug Characteristics, in: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2008), 2008, pp. 329–339.

[22] L. Naish, H.J. Lee, K. Ramamohanarao, A model for spectra-based software diagnosis, ACM Transactions on Software Engineering and Methodology 20 (3) (2011). Article no. 11.

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

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

[25] S. Park, R.W. Vuduc, M.J. Harrold, Falcon: fault localization in concurrent programs, in: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE 2010), vol. 1, ACM, New York, NY, 2010, pp. 245–254.

[26] M. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, Service-oriented computaing: a research roadmap, International Journal on Cooperative Information Systems 17 (2) (2008) 223–255.

[27] M. Renieris, S. Reiss, Fault localization with nearest neighbor queries, in: Proceedings of the 18th International Conference on Automated Software Engineering (ASE 2003), IEEE Computer Society, 2003, pp. 30–39.

[28] R. Santelices, J.A. Jones, Y. Yu, M.J. Harrold, Lightweight fault-localization using multiple coverage types, in: Proceedings of the 31st International Conference on Software Engineering (ICSE 2009), IEEE Computer Society, 2009, pp. 56–66.

[29] M. Song, E. Tilevich, Metadata invariants: checking and inferring metadata coding conventions, in: Proceedings of the 34th International Conf. on Software Engineering (ICSE 2012), 2012, pp. 694–704.

[30] C. Sun, On Open Issues on SOA-based Software Development, China Science Paper, 2011. <http://www.paper.edu.cn/index.php/default/releasepaper/content/201107-461>.

[31] C. Sun, Y. Shang, Y. Zhao, T.Y. Chen, Scenario-oriented testing of service compositions using BPEL, in: Proceedings of the 12th International Conference on Quality Software (QSIC 2012), IEEE Computer Society, 2012, pp. 171–174.

[32] C. Sun, Y. Zhai, Y. Shang, Z.Y. Zhang, Toward effectively locating integration-level faults in BPEL programs, in: Proceedings of the 12th International Conference on Quality Software (QSIC 2012), IEEE Computer Society, 2012, pp. 17–20.

[33] C. Sun, E. Khoury, M. Aiello, Transaction management in service-oriented systems: requirements and a proposal, IEEE Transactions on Services Computing 4 (2) (2011) 167–180.

[34] W.E. Wong, Y. Qi, L. Zhao, K.Y. Cai, Effective fault localization using code coverage, in: Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC 2007), 2007, pp. 449–456.

[35] Y. Yu, J.A. Jones, M.J. Harrold, An empirical study of the effects of test-suite reduction on fault localization, in: Proceedings of the 30th International Conference on Software Engineering (ICSE 2008), ACM, New York, NY, 2008, pp. 201–210.

[36] Z. Zhang, W.K. Chan, T.H. Tse, Fault localization based only on failed runs, IEEE Computer 45 (6) (2012) 42–49.

[37] Z. Zhang, B. Jiang, W.K. Chan, T.H. Tse, X. Wang, Fault localization through evaluation sequences, Journal of Systems and Software 83 (2) (2010) 174–187.

[38] Z. Zhang, W.K. Chan, T.H. Tse, Y.T. Yu, P. Hu, Non-parametric statistical fault localization, Journal of Systems and Software 84 (6) (2011) 885–905.

[39] L. Zhao, Z. Zhang, L. Wang, X. Yin, PAFL: Fault localization via noise reduction on coverage vector, in: Proceedings of the 23th International Conference on Software Engineering and Knowledge Engineering (SEKE 2011), 2011, pp. 203–206.