An empirical study on mutation testing of WS-BPEL programs

Sun, Chang-ai; Pan, Lin; Wang, Qiaoling; Liu, Huai; Zhang, Xiangyu

Sun, Pan, L., Wang, Q., Liu, H., & Zhang, X. (2016). An empirical study on mutation testing of WS-BPEL programs. The Computer Journal, 60(1), 143–158. https://doi.org/10.1093/comjnl/bxw076
Document Version: Accepted Manuscript

Published Version: https://doi.org/10.1093/comjnl/bxw076

An Empirical Study on Mutation Testing of WS-BPEL Programs

Chang-ai Sun, Lin Pan, Qiaoling Wang
School of Computer and Communication Engineering
University of Science and Technology Beijing, China

Huai Liu
Australia-India Research Centre for Automation Software Engineering
RMIT University, Melbourne, Australia

Xiangyu Zhang
Department of Computer Science
Purdue University, West Lafayette, Indiana, USA

Abstract

Nowadays, applications are increasingly deployed as Web services in the globally distributed cloud computing environment. Multiple services are normally composed to fulfill complex functionalities. Business Process Execution Language for Web Services (WS-BPEL) is an XML-based service composition language that is used to define a complex business process by orchestrating multiple services. Compared with traditional applications, WS-BPEL programs pose many new challenges to the quality assurance, especially testing, of service compositions. A number of techniques have been proposed for testing WS-BPEL programs, but only a few studies have been conducted to systematically evaluate the effectiveness of these techniques. Mutation testing has been widely acknowledged as not only a testing method in its own right, but also a popular technique for measuring the fault-detection effectiveness of other testing methods. Several previous studies have proposed a family of mutation operators for generating mutants by seeding various faults into WS-BPEL programs. In this study, we conduct a series of empirical studies to evaluate the applicability and effectiveness of various mutation operators for WS-BPEL programs. The experimental results provide insightful and comprehensive guidance for mutation testing of WS-BPEL programs in practice. In particular, our work is the systematic study in the selection of effective mutation operators specifically for WS-BPEL programs.

Keywords: Web service, service composition, Business Process Execution Language for Web Services, mutation testing

1 Introduction

Due to the increasing popularity of cloud computing, service-oriented architecture (SOA) has become a major paradigm for developing distributed applications. Web service is the fundamental unit of SOA, which describes a series of application interfaces based on the standard Extensible Markup Language (XML). Normally, a single Web service can only implement simple functionalities, so it is necessary to compose multiple services for achieving complex and flexible processes. Business Process Execution Language for Web Services (WS-BPEL) [29] can help us orchestrate different Web services into business processes, which are then provided as composite services. WS-BPEL-based service compositions have a number of features, such as dynamic interoperability, open networking environment, loose coupling, etc. These features result in nondeterminism in the composition and execution of Web services, which in turn pose new challenges in the quality assurance of Web services and their compositions.

Testing has been acknowledged as the mainstream approach to quality assurance, validation, and verification. Though there exist a lot of testing techniques for traditional software systems, many of them are no longer applicable to Web services and their compositions. For example, white-box testing techniques, which are based on the source code of the system under test, are not applicable in the context of service testing, because Web services may be owned by the third party and deployed in a remote server. Even for the black-box testing techniques, which are based on the software specifications, some enhancements are required to fit into the specific SOA context. Various techniques have been proposed to generate test cases specifically for the testing of Web services and their compositions [10, 16, 26, 36, 34, 25].

The quality of generated test cases decides the effectiveness of a testing method. Mutation testing [9], which systematically generates a set of variants of the base program, namely mutants, by seeding a variety of faults, is a popular technique to evaluate the fault-detection effectiveness of a set of test cases. It has been demonstrated that the automatically generated mutants resemble real-life faults very well [2]. Mutation testing technique has also been used for evaluating the effectiveness of testing methods for WS-BPEL programs [36]. However, in these previous studies, most of the mutants were generated manually, that is, they lacked a systematic way of generating mutants for WS-BPEL programs.

Some research [13] has already been conducted to propose various mutation operators, which can be applied to generate mutants for WS-BPEL programs. Such studies make it possible to construct a comprehensive tool for automatically constructing WS-BPEL mutants. In addition, previous studies [14] have shown significant difference between different types of mutation operators in terms of the applicability in constructing mutants and the difficulty in killing the generated mutants. However, more comprehensive empirical studies are required to evaluate these mutation operators in depth.

In this paper, we report a series of empirical studies on various WS-BPEL programs, which help us quantitatively measure the applicability and effectiveness of different types of mutation operators for WS-BPEL. To conduct the ex-

periments, we propose a comprehensive framework for mutation testing of WS-BPEL programs, based on which, we develop a tool, namely μBPEL. With the aid of evaluation results, we can preclude a set of ineffective mutation operators, reduce the number of mutants, and thus decrease the cost of mutation testing. Similar studies have been conducted on various types of programs [5, 30, 28], whereas our work is the systematic study in selecting useful mutation operators specifically for WS-BPEL programs.

The comprehensive empirical study presented in this paper mainly makes the following contributions:

- Evaluation of applicabilities for different mutation operators, that is, how difficult/easy it is to apply a certain mutation operator to generate WS-BPEL mutants;

- Quantitative measurement of testing effectiveness for various mutation operators through two metrics, namely the mutation score and the fault discovery rate; and

- Identification of the hierarchy showing the difficulty in killing mutants generated by different mutation operators, which can serve as a basis for the selection of the most "effective" operators in the mutation analysis for WS-BPEL programs.

The rest of the paper is organized as follows. Section 2 introduces some background information on WS-BPEL, mutation testing, and mutation operators for WS-BPEL. The empirical studies and relevant results are reported in Sections 3 and 4. Section 5 discusses the work related to our study. The paper is summarized in Section 6.

2 Background

2.1 Business Process Execution Language for Web Services (WS-BPEL)

Business Process Execution Language for Web Services (WS-BPEL) [29] is an XML-based executable language for service orchestration, which can construct complex business processes through composing multiple independent Web services. The composite services implemented in WS-BPEL can provide functionalities in the form of basic services, and thus WS-BPEL programs can participate in higher-level business processes.

WS-BPEL makes use of a number of XML specifications, such as WSDL (Web Services Description Language), XML Schema, Xpath (XML Path language), and XSLT (Extensible Stylesheet Language Transformations). In WS-BPEL processes, the data model is provided by the WSDL message and XML Schema's data type definition; the data operation is supported by Xpath and XSLT; and the interface for Web services is described by WSDL.

Figure 1 shows a sample segment of WS-BPEL processes. A WS-BPEL process is mainly composed of four sections, namely *partner link statements*, *variables*, *fault handlers*, and *interaction steps*. The fundamental unit in the WS-BPEL process is *activity*, which can be further classified into *basic activity* and *structural activity*. Typical basic activities include *assign*, *invoke*, *receive*, *reply*, *throw*, *wait*, *empty*, etc. The structural activity provides the controlling structure required for process execution, and is normally composed of multiple basic and/or structural activities. Typical controlling structures include *sequence*, *switch*, *while*, *pick*, etc. In addition, WS-BPEL programs support concurrency and synchronization among activities through the *flow* structure and link tags within flows. The process receives an "input" message variable and outputs an "output" message variable, whose message types have been declared in the *variables* section.

Although WS-BPEL has standard control structures, such as *sequences*, *branches*, and *loops*, WS-BPEL programs are significantly different from the traditional programs in the following aspects [35]: (*i*) WS-BPEL provides an explicit integration mechanism to compose Web services into large-scale systems, while such integrations in traditional programs are implicit; (*ii*) Web services used by WS-BPEL programs may be implemented in different programming languages, while modules in the traditional programs are usually implemented in the same programming language; (*iii*) WS-BPEL programs are represented as XML files, and the statements are not the same as those in the traditional programs; (*iv*) WS-BPEL supports concurrency among activities via flow activities and synchronization via link tags within flows, which is not common in traditional programs. When implementing such WS-BPEL programs, people may introduce new types of faults that are different from those in traditional programs due to these new features of WS-BPEL programs.

2.2 Mutation testing

Mutation testing [9] is a fault-based software testing technique. Basically, some mutation operators are used to seed certain types of faults into the base program. The faulty versions are called *mutants*. In mutation testing, test cases are executed in both the base program and the mutants. Once a test case reveals different execution behaviors (normally different program outputs) between a certain mutant and the base program, the test case is said to kill the mutant.

Mutation testing has two basic hypotheses, namely *competent programmer* and *coupling effect*. The competent programmer hypothesis implies that the programmer can create at least nearly correct programs, which only contain few subtle faults. The coupling effect hypothesis states that complex faults can be coupled into a set of simple faults in a way that a test suite detecting all the simple faults is very likely to detect the complex faults. Based on these two hypotheses, most studies on mutation testing focus on first-order mutants, that is, those generated by seeding one single fault by applying one and only one mutation operator to the base program.

The original purpose of mutation testing was to generate test cases that are

Figure 1: A sample segment of WS-BPEL process

sufficient to kill all mutants. Fraser and Zeller [17] investigated how to construct unit test cases and oracles based on the mutation technique. In addition, the basic idea of mutation testing was applied in the testing of SQL injection vulnerabilities [4]. Recently, mutation testing technique was applied to the evaluation of different code coverage criteria [19, 20].

Mutation testing has been widely used to evaluate the fault-detection effectiveness of a given test suite, which is normally measured by how many mutants have been killed by the test suite[33]. In practice, there exist some mutants that cannot be killed by any possible test case. This type of mutants are called *equivalent mutants*, which means that they are equivalent to the base program given any possible program input. Identifying equivalent mutants is an important issue, because it affects the calculation of mutation metrics. Fortunately, one recent work reported a novel technique for detecting equivalent mutants, namely trivial compiler equivalence (TCE) [31], and conducted a large-scale empirical study to evaluate the effectiveness of TCE. In the ideal case, a good test suite should be able to kill all the non-equivalent mutants for a given base program. Andrews et al. [2] have empirically justified that compared with hand-seeded faults, automatically generated mutants show higher similarity to the real-life faults. In other words, mutation testing is a good indicator of the testing effectiveness in reality. Numerous studies in software testing have used mutation testing to evaluate the effectiveness of various testing techniques [8, 23, 37, 39, 40]. Researchers in the testing of Web services also used mutation testing to evaluate the testing techniques proposed by them [36, 34].

A major drawback of mutation testing is its high computation overhead mainly due to the large number of mutants. Research [5, 30, 28] has been conducted to find the "subsuming" relations between different mutation operators, remove part of operators that are not strong enough, and thus reduce the cost of mutation testing.

2.3 Mutation operators for WS-BPEL Programs

In order to mimic the faults programmers may introduce into WS-BPEL programs, Estero-Botaro et al. [13] have proposed a total of 26 mutation operators for WS-BPEL, as summarized in Table 1. These operators are classified into four categories, namely *identifier replacement* mutation operator, *expression* mutation operator, *activity* mutation operator, *exception and event* mutation operator. The activity operator is further classified into two sub-categories, related to concurrency and non-concurrency, respectively. Since the inception of these operators, they have been applied to generating mutants as the subject programs in the experimental studies for evaluating the fault-detection effectiveness of the testing techniques for WS-BPEL programs [36]. Recently, García-Domínguez and Medina-Bulo [18] updated mutation operators for WS-BPEL. They introduced EIN, EIU, EAP and EAN in the "Expression mutation operator" category and added the "Coverage operator" category, which includes CFA and CDE.

Figure 2 illustrates a possible ISV fault that may be injected into the WS-

Table 1: 32 mutation operators for WS-BPEL programs proposed by Estero-Botaro et al. [13] and by García-Domínguez and Medina-Bulo [18]

Operator	Description
Identifier replacement mutation operator	
ISV	Replace a variable identifier by another variable identifier
Expression mutation operator	
EAA	Replace an arithmetic operator by another arithmetic operator
EEU	Remove the unary minus operator from an expression
ERR	Replace a relational operator by another relational operator
ELL	Replace a logical operator by another logical operator
ECC	Replace a path operator by another path operator
ECN	Modify the value of a numeric constant by incrementing/decrementing it by one, or adding/removing one digit
EMD	Modify a duration expression by replacing it by 0 or half of its initial value
EMF	Modify a deadline expression by replacing it by 0 or half of its initial value
EIN	Insert the XPath negation function (not) in logic expressions
EIU	Insert the XPath unary minus operator in aritmethic expressions
EAP	Replace a subexpression by its positive absolute value
EAN	Replace a subexpression by its negative absolute value
Activity mutation operator – related to concurrency	
ACI	Change the `createInstance` attribute from an inbound message activity to *no*
AFP	Replace a sequential `forEach` activity by a parallel one
ASF	Replace a `sequence` activity by a `flow` activity
AIS	Change the `isolated` attribute of a `scope` to *no*
Activity mutation operator – related to non-concurrency	
AEL	Delete an activity
AIE	Delete an `elseif` element or the `else` element from an `if` activity
AWR	Replace a `while` activity by a `replaceUntil` activity or vice versa
AJC	Remove the `joinCondition` attribute from an activity
ASI	Exchange the order of two `sequence` child activities
APM	Remove an `onMessage` element from a `pick` activity
APA	Remove the `onAlarm` element from a `pick` activity or from an event handler
Exception and event mutation operator	
XMF	Remove a `catch` element or the `catchAll` element from a fault handler
XMC	Remove a compensation handler definition
XMT	Remove a termination handler definition
XTF	Replace the fault thrown by a `throw` activity
XER	Remove a `rethrow` activity
XEE	Remove an `onEvent` element from an event handler
Coverage operator	
CFA	Replace an activity by an **exit** activity
CDE	Replace a decision condition with TRUE or FALSE

BPEL program segment shown in Figure 1. In this example, the correct variable reference of the *reply* activity should be "output", while it is mistakenly implemented as "input". As a result, the output of mutated program is always equal to its input.

```
//original "reply" activity
 <reply name="reply" operation="OrderGoods"
   partnerLink="ordergoodsPL" portType="order:OrderGoodsPT"
   variable="output">
 </reply>
```

```
//Mutated "reply" activity
 <reply name="reply" operation="OrderGoods"
   partnerLink="ordergoodsPL" portType="order:OrderGoodsPT"
   variable="input">
 </reply>
```

Figure 2: Illustration of the ISV fault in the WS-BPEL program segment

3 Empirical Study

We have conducted a series of empirical studies to quantitatively evaluate the applicability and effectiveness of WS-BPEL mutation operators. The design and settings of the experiments are described in this section.

3.1 Research questions

In this study, we aim to answer three research questions.

RQ1 To what extent is a mutation operator applicable for mutant generation on a given WS-BPEL program?

Since each mutation operator for WS-BPEL programs actually imitates a type of fault, it is interesting to know how likely the faults imitated by different mutation operators occur in real-life WS-BPEL programs and their distribution. If a mutation operator can help us easily generate a large number of mutants, we can say that it has high applicability in mutation testing; otherwise, the applicability of the operator is very limited. By answering RQ1, we can identify a list of applicable mutation operators, based on which we can generate a sufficient number of WS-BPEL mutants.

RQ2 How difficult/easy is it to kill the mutants that are generated based on a certain mutation operator?

If almost all test cases can kill any mutant generated by an operator, we can say that the operator is not very effective in distinguishing the qualities of different test suites. By answering RQ2, we can identify a set

of effective mutation operators, and give them higher priority in mutation testing.

RQ3 Is there any "subsuming" relation between two mutation operators such that a test suite capable of killing mutants associated with one operator will definitely also kill the mutants associated with the other operator?

Obviously, if such a "subsuming" relation exists, the subsumed operator can be simply removed in the mutation testing. By answering RQ3, we can further reduce mutation operators, and thus improve the efficiency of mutation testing.

The first question is to evaluate the applicability of WS-BPEL mutation operators, for which similar results could be found but has not been formally answered in previous studies [14]. The other two research questions are about the effectiveness of these operators, which has always been the focus of the studies on mutation testing [11, 14, 24], but we measured the effectiveness using slightly different metrics from a different perspective, as detailed in this section.

3.2 Subject programs

To address the above research questions, we conducted our study based on six WS-BPEL programs in various domains. Table 2 gives the basic information of these subject programs. In the table, LOC is the abbreviation of Lines of Code. Since a WS-BPEL program is represented as an XML file, we use the number of XML lines to represent the size of the WS-BPEL program. In the `SupplyChain` program [35], the customer is required to input the name and number of products, and the retailer will provide feedback based on the order and the status of warehouse. For the `SmartShelf` program [35], the input is some information of commodity, such as name and number, and the output includes the quantity in warehouse, the location of shelf, and the status of commodity. In the `SupplyCusomter` program [29], the customer inputs the items and address of the order, and the system replies the validation result. The `LoanApproval` [29] examines the personal information and loan amount provided by the customer, and returns whether the loan application is approved or rejected. In the `CarEstimate` program [1], the system provides initial, simple, complex, or formal assessment reports for repairing cars based on the request from the customer. The `TravelAgency` program [32] is composed of hotel booking, agency booking, air-ticket booking, and banking. The selected six WS-BPEL programs have been well described in the existing literature, and thus could be regarded as representative. Ideally, including a set of larger subject programs for evaluation will deliver more convincing results. Unfortunately, there is not still such a benchmark for evaluation. Although WS-BPEL has been adopted to implement various complex business processes, it is impossible to include them for evaluation due to the commercial issue.

Table 2: WS-BPEL programs as experimental subject programs

Program name	Basic functionality	No. of services composed	Size (LOC)
`SupplyChain`	Management of supply chains	2	50
`SmartShelf`	Management of commodity shelves	14	194
`SupplyCustomer`	Management of project orders	5	122
`LoanApproval`	Examination of loan applications	3	120
`CarEstimate`	Assessment of car repairs	7	121
`TravelAgency`	Booking of travels	9	543

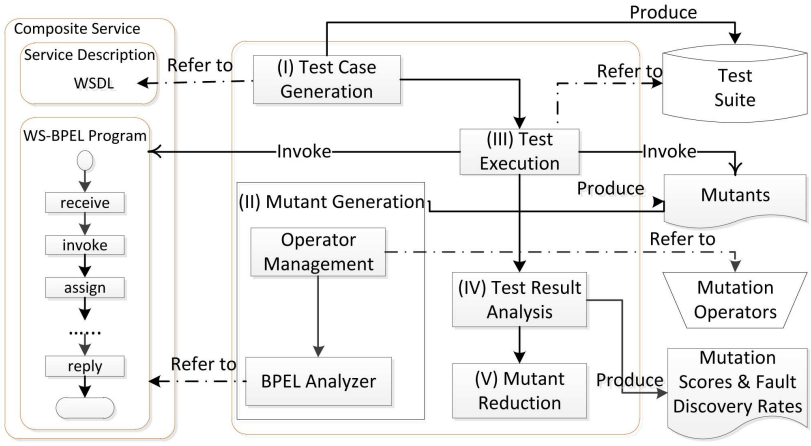

Figure 3: Framework for executing mutation testing of WS-BPEL programs

3.3 Mutation Testing Framework for WS-BPEL

In this study, we propose a framework for executing WS-BPEL mutation testing, as given in Figure 3. As mentioned before, a WS-BPEL program is a composite service derived by orchestrating Web services and other composite services. Just like Web services, the description of such a composite service is also specified by WSDL, from which one can derive its interfaces, including operations, input and output parameters of each operation.

The basic procedure of this framework is as follows.

(I) Test Case Generation:

- (a) Analyze the WS-BPEL program's service description, and identify the operations provided by the program;
- (b) Analyze the types and constraints of input parameters of the operation under test;
- (c) Generate test suites based on some testing techniques or coverage criteria.

(II) Mutant Generation:

- (a) Analyze the WS-BPEL program, and identify the elements that can be mutated;
- (b) Based on the analysis result, match breakpoints inside the source code with corresponding mutation operators;
- (c) Construct mutants according to the transformation rules specified in the mutation operators.

(III) Test Execution:

- (a) Input test cases, execute the original WS-BPEL program, and record the output;
- (b) Execute mutants using the same test cases, and record their outputs.

(IV) Test Result Analysis:

- (a) Calculate the mutation scores and fault discovery rates;
- (b) Summarize, analyze, and report test results.

(V) Mutant Reduction:

- (a) Find and remove those mutants that can be killed by any test case;
- (b) Find mutants that are virtually identical to each other in terms of their failure behaviors (details can be found later in Section 3.6.2), and remove redundant mutants.

In order to automate the aforementioned framework as much as possible, we have implemented a comprehensive system, μBPEL, which supports mutation testing for WS-BPEL programs in the following ways:

- **Mutant Generation**: μBPEL accepts a WS-BPEL program as input, and outputs a set of mutants for the program. The system first makes use of the XML files reader to analyze the WS-BPEL program, and finds the elements that can be mutated. Then, it executes the conversion rules within the mutation operators. Finally, it makes use of the XML files writer to seed faults into the original WS-BPEL program according to the conversion rules.

- **Testing Execution**: μBPEL receives a set of test cases, the original WS-BPEL program, and mutants, while its output is the results for the testing of mutants. The system first reads out the test cases that testers have defined and stored, then executes both the original WS-BPEL program and mutants using the test cases, and records the corresponding outputs. It is implemented by extending an existing WS-BPEL engine, Apache ODE [3].

- **Test Result Analysis**: μBPEL receives the test results on mutants, while its output is a test report. The system compares the outputs of the mutants with those of the base program, and obtain the basic information on which test case kills which mutant. It also provides the statistics on the fault detection rate for each mutant and mutation scores. The generated report will contain the following information: name of mutant, total number of test cases, number of test cases that can kill a certain mutant, etc.

Besides, μBPEL also supports the evaluation of existing testing techniques for WS-BPEL programs. In the current version of μBPEL, two well-known testing techniques for WS-BPEL programs are supported. The derived test suites can be seamlessly used by the test execution.

- **Scenario-oriented testing** [36]: This technique first converts a WS-BPEL program into an abstract graph model, which consists of nodes (corresponding to an activity) and edges (corresponding to a transition between nodes). From this graph model, we can then derive test scenarios with respect to a specific coverage criterion, and generate a test suite for each test scenario with the aid of constraint solvers. We particularly developed an automatic scenario-oriented testing technique, and the technique has been integrated into μBPEL as a component to aid the test case generation for mutation testing of WS-BPEL programs.

- **Random testing**: μBPEL first parses the WSDL file of WS-BPEL programs, and then randomly generates test data to satisfy the manually entered constraints over the input parameters.

With the aid of μBPEL, the mutant generation, the test execution, and the test result analysis in the mutation testing framework for WS-BPEL programs can be automated. Nevertheless, some manual work is still required. For example, in (IV) Test Result Analysis, it is sometimes necessary for testers to

precisely decide whether an alive mutant is an equivalent one or not. In addition, it has not been fully automated to identify the mutants that have identical failure behaviors (in (V) Mutant Reduction). It should be pointed out that such manual work is quite important, particularly because mutant reduction can help increase the efficiency of mutation testing. Some WS-BPEL mutation tools [6, 12, 18, 15] have been developed before our μBPEL tool. The detailed comparison between μBPEL and the existing tools will be presented in Section 5.

3.4 Generation of mutants

Our μBPEL tool was used to generate mutants for each subject program. Table 3 gives the statistics in the mutant generation. To ensure the validity of our empirical studies, the selection of mutation operators was independent from that of subject programs, that is, we did not attempt to match the mutation operators and subject programs on purpose. In our empirical studies, we chose totally 32 mutation operators for WS-BPEL from the literature [13, 18, 15]. However, as to be shown in Section 4, only part of the operators can be used for each subject program. Such an observation is consistent with what has been observed in previous studies [14]. In addition, different mutation operators can lead to different numbers of mutants for different subject programs. Relevant details will be given in Section 4. In mutation analysis, there are two ways to judge whether a mutant is killed. For strong mutation analysis, it requires the following three conditions to be met: (1) a test must reach the mutated statement; (2) test input data should infect the program state by causing different program states for the mutant and the original program; (3) the incorrect program state must propagate to the program's output and be checked by the test. Weak mutation analysis requires that only the first and second conditions are satisfied. Strong mutation analysis is more powerful, since it ensures that the test suite can really catch the problems, while weak mutation analysis is closely related to code coverage methods and thus not widely used in practice. In our experiment, it is not trivial to catch the program state. Therefore, we used strong mutation to judge whether a mutant was killed or not, that is, a mutant was said to be killed when a test case produced different outputs between the mutant and the base program. To ensure that our evaluation is comprehensive, we first generated mutants using our μBPEL tool and then appended some mutants using the other mutant generation system called MuBPEL [18].

3.5 Generation of test cases

We mainly made use of four typical testing techniques, namely *equivalence class partitioning* [27], *boundary value analysis* [27], random testing [27], and scenario oriented testing [36], to generate test cases for the subject programs. The equivalence class partitioning technique divides the program input space into a number of partitions. All possible inputs within one partition are expected to trigger the same execution behavior, so each partition is also called an equiv-

Table 3: Mutant generation for subject programs

Program name	Number of generated mutants	Number of used operators
`SupplyChain`	31	10
`SmartShelf`	159	11
`SupplyCustomer`	67	11
`LoanApproval`	69	15
`CarEstimate`	60	9
`TravelAgency`	163	16

alence class. At least one test case should be selected from each equivalence class. The boundary value analysis technique focuses on the boundaries between different equivalence classes. It specifically selects test cases on or near the boundaries. The scenario oriented testing first transforms the WS-BPEL program into a graph model, then derives test scenarios from the model, finally generates test cases according to different scenarios. The random testing generates test cases based on the input parameters and the constraints on their ranges.

In our experiments, we first partitioned the program input space into a number of equivalence classes. Then, we identified the boundaries between different classes as well as the critical values on or close to the boundaries. Based on these classes, boundaries, and values, we constructed a test suite such that (1) each equivalence class has been covered at least once; and (2) most test cases in the suite are on or close to the boundaries. In our study, we created five test suites with different sizes, namely T_x, T_y, T_z, T_u, and T_w, for each subject program. Table 4 reports the size of each test suite (denoted by $|T_x|$, $|T_y|$, $|T_z|$, $|T_u|$, and $|T_w|$, respectively) for each program. Note that the size of a test suite depends on the size of program under test and the concrete test case generation method. On one hand, it is natural to have a smaller test suite for a smaller program. For example, the programs `SupplyChain` and `SmartShelf` have the smallest and largest numbers of composed services (i.e. 2 and 14, refer to Table 2), respectively, among all six subject programs. Correspondingly, their associated test suites have the smallest and largest sizes among all suites. On the other hand, different testing methods can lead to different sizes of test suites. For the first three test suites (namely T_x, T_y, and T_z), we used a hybrid method of equivalence class partitioning and boundary value analysis. Obviously, the number of equivalence classes and the number of boundaries between classes also affect the size of test suites constructed in our experiments. In order to guarantee the generality of our study, we also used other two test case generation techniques, namely scenario-oriented testing [36] and random testing, to construct two more test suites (namely T_u and T_w).

Table 4: Test case generation for subject programs

Program name	$\|T_x\|$	$\|T_y\|$	$\|T_z\|$	$\|T_u\|$	$\|T_w\|$
`SupplyChain`	6	10	15	19	30
`SmartShelf`	22	35	50	74	90
`SupplyCustomer`	7	12	18	24	30
`LoanApproval`	12	18	25	31	40
`CarEstimate`	10	20	30	36	40
`TravelAgency`	4	7	10	14	20

3.6 Variables and measurements

3.6.1 Independent variables

There are two independent variables in our experiments. One is the mutation operator. During the mutant generation process, we have considered all 32 mutation operators [13][18][15]. However, as have been discussed above and will be detailed in Section 4, only part of them could be applied to generate mutants for our subject programs. The other independent variable is the test case generation technique. As presented above, we used a hybrid technique that integrates equivalence class partitioning and boundary value analysis.

3.6.2 Dependent variables

For RQ1, we made use of three metrics to evaluate the applicability of a mutation operator. They are the total number of mutants that can be generated based on the operator (denoted by N_o), the percentage of mutants generated by the operator out of all generated mutants (denoted by P_o), and the number of subject programs the operator can be applied to for generating mutants (denoted by N_p). Obviously, the larger values these metrics have, the more applicable a mutation operator is for WS-BPEL programs.

For RQ2, we employed two metrics, namely *mutation score* (MS) and *fault discovery rate* (FDR), to measure the effectiveness of a test suite in killing mutants. MS is defined as

$$\mathrm{MS}(p, TS) = \frac{N_k}{N_m - N_e}, \tag{1}$$

where p refers to the program under test, TS refers to the test suite used for testing the mutants, N_k refers to the number of mutants killed by TS, N_m refers to the total number of generated mutants, and N_e refers to the number of equivalent mutants. The identification of equivalent mutants is an undecidable problem. In theory, there is no way of fully automatic identification of equivalent mutants. In our study, we paid much attention to the mutants that were still alive after executing all test cases and manually checked them to decide whether they are equivalent to the base program or not. This kind of manual effort is feasible due to the relatively small number of surviving mutants [38]. MS

intuitively indicates the capability of a test suite killing mutants. The larger the MS is, the more effective a test suite is in killing mutants for the given program.

FDR is defined as

$$\mathrm{FDR}(m, TS) = \frac{N_f}{N_{ts}}, \tag{2}$$

where m refers to a certain mutant, TS refers to the test suite, N_f refers to the number of test cases that can kill m, and N_{ts} refers to the total number of test cases in TS. Intuitively speaking, FDR indicates how effective a test suite is in killing a certain mutant. The larger the FDR is, the more effective a test suite is to kill the given mutant.

For RQ3, we introduced the subsumption relation to reduce the mutation operators. We compared two mutation operators based on each test case and involved mutants, and observed their failure behaviors. Suppose that there are two mutation operators MO_A and MO_B. For a given subject program, MO_A can generate a set of mutants, denoted by $\mathbf{MU_A} = \{MU_1^A, MU_2^A, \cdots, MU_i^A\}$, while the set of mutants generated by MO_B is denoted by $\mathbf{MU_B} = \{MU_1^B, MU_2^B, \cdots, MU_j^B\}$. We can say that MO_B is subsumed by MO_A, if $\forall\ MU_l^B \in \mathbf{MU_B}$, $\exists\ MU_k^A \in \mathbf{MU_A}$ such that $\forall$ test case tc that can kill MU_k^A, MU_l^B must also be killed by tc. In other words, MO_B is subsumed by MO_A when $\exists\ \mathbf{MU'_A} \subseteq \mathbf{MU_A}$ such that $\mathbf{MU'_A}$ and $\mathbf{MU_B}$ are virtually identical to each other in terms of failure behaviors.

3.7 Threats to validity

The threat to the *internal validity* is mainly concerned with the implementation of experiments, which require moderate amount of programming work. Most of the programming was conducted by one co-author, under the supervision and management of another co-author. The source code was further cross-checked by different individuals, who confirmed that both the μBPEL tool and our experiments were correctly implemented. In our experiments, some equivalent mutants were manually identified. All of them were checked by different individuals, and the reasons why they are equivalent to the base programs was also investigated and justified. Therefore, we are confident that the threats to the internal validity have been minimized in our study.

The major threat to the *external validity* relates to the selection of subject programs. Though we have chosen six WS-BPEL programs as the experimental subject programs, we cannot say that our conclusion will be valid for any other program without further study. Such a limitation is applicable to any empirical study. One solution for the problem is a comprehensive theoretical study to justify our conclusions, especially those with regard to the "subsuming" relation between different mutation operators. Another threat to the external validity is about the faults seeded in the subject programs: These faults may not be able to fully represent the defects in practice.

There is little threat to the *construct validity*, which is regarding the measurements employed in the empirical studies. Most of the metrics, especially

MS and FDR, are very straightforward to understand and easy to use.

The threat to the *conclusion validity* is due to the limited number of experimental data. Tens of mutants were automatically constructed based on each subject program using our μBPEL tool. Two typical testing techniques were employed to generate three test suites for each program. Common observations were made across different programs and different test suites. However, we cannot guarantee that our conclusion is valid in a more general sense.

4 Experimental Results

4.1 Answer to RQ1

Table 5 summarizes the values of N_o, P_o, and N_p (the definitions of which have been given in Section 3.6.2) for each mutation operator.

From Table 5, we can observe that seven mutation operators (AFP, AIS, AWR, XMT, XTF, XER, and XEE) have not been used for any of the six subject programs. An in-depth investigation showed that these operators require very specific structures of WS-BPEL programs. For example, the XER operator will be applicable only when there exists a `rethrow` activity in the program under test, which may not be a common case in reality. There exist another nine mutation operators (EAA, EEU, EMD, EMF, AJC, APM, APA, XMF and XMC), each of which can only generate one mutant from only one subject program. We found that they have similar features to the previous seven operators: They also have very specific requirements in the structures of WS-BPEL programs. For instance, there must be a `catch` or `catchAll` element in a `fault handler` for XMF to be applicable.

Different from the above 16 mutation operators (which are "not so applicable"), some operators can be widely used across different subject programs. The AEL and CFA operators helped generate the largest number of mutants among all operators, and they can be applied to all six programs. Each of the five operators ACI, ASF, AEL, ASI, and CFA, is applicable to all six subject programs. By examining the definitions of these five operators, we found that none of them requires specific structures in the program under test. In other words, the more general the mutation operator is, the more applicable it is to generate mutants.

In summary, different mutation operators have different applicability for the mutant generation of WS-BPEL programs. On one hand, the operators ACI, AEL, CFA, ASF, ASI, and ERR could be widely applied to generate a large number of mutants for many subject programs. On the other hand, the operators AFP, AIS, AWR, XMT, XTF, XER, XEE, EAA, EEU, EMD, EMF, AJC, APM, APA, XMF and XMC have very limited applicability. Note that it is not surprising to observe the variation in the applicability of different operators on various programs. Similar observations have been made in previous studies for WS-BPEL mutation testing [5, 14] and even for other types of programs such as Java [24].

Table 5: Measurements of applicabilities of WS-BPEL mutation operators

Operator	N_o	P_o	N_p
ISV	7	1.28%	2
EAA	4	0.73%	1
EEU	1	0.18%	1
ERR	75	13.66%	5
ELL	2	0.36%	2
ECC	6	1.09%	2
ECN	24	4.37%	2
EMD	2	0.36%	1
EMF	2	0.36%	1
EIN	15	2.73%	5
EIU	6	1.09%	2
EAN	6	1.09%	2
EAP	6	1.09%	2
ACI	6	1.09%	6
AFP	0	0.00%	0
ASF	34	6.19%	6
AIS	0	0.00%	0
AEL	118	21.49%	6
AIE	11	2.00%	4
AWR	0	0.00%	0
AJC	1	0.18%	1
ASI	71	12.93%	6
APM	1	0.18%	1
APA	1	0.18%	1
XMF	1	0.18%	1
XMC	1	0.18%	1
XMT	0	0.00%	0
XTF	0	0.00%	0
XER	0	0.00%	0
XEE	0	0.00%	0
CDE	30	5.46%	5
CFA	118	21.49%	6

4.2 Answer to RQ2

Table 6 gives the value of MS on each subject program. Note that for each subject program, all five test suites (T_x, T_y, T_z, T_u, and T_w) always had the same value of MS, so we do not distinguish different test suites in Table 6. In addition, equivalent mutants existed for some subject programs. For ease of presentation, Table 6 also includes the number of equivalent mutants (denoted by N_e, as defined in previous Section 3.6.2) and the mutation operators that generated these equivalent mutants.

In our experiments, several operators could lead to equivalent mutants. ASF replaces a sequence activity by a flow activity. If the flow activity and the sequence activity have the same structure, we may not be able to distinguish the resulting mutant from the base program. ASI exchanges the order of two sequence child activities. If these two activities are not sensitive to time, the resulting mutant may be equivalent to the base program. ECC replaces a path operator by another path operator. If the original path is the root of a path, and the path operator "\" is replaced with "\\", then the resulting mutant is an equivalent one. EAP replaces a subexpression by its positive absolute value. When the value of replaced subexpression is positive, we may not be able to distinguish the resulting mutant from the base program. AEL deletes an activity, which may produce equivalent mutants. For instance, if an "assign" activity is deleted, and the assigned variable does not affect the subsequent activities, it is impossible to distinguish the resulting mutant from the base program. CDE replaces a decision condition with TRUE or FALSE. When the decision is always TRUE and it is replaced with TRUE, an equivalent mutant is generated.

Though Table 6 shows that the value of MS is constantly high across different subject programs, there still exist some mutants that could not be killed by our test suites. The "alive" mutants include two mutants generated by the ERR operator from `SmartShelf`, three mutants generated by ERR, one mutant by ELL, and one by AJC from `SupplyCustomer`, and one mutant generated by the XMC operator from `TravelAgency`. In other words, these mutation operators can produce "stubborn" mutants that would be quite effective in the experiments for evaluating the effectiveness of a testing method. Such an observation implies that we should pay special attention to these operators in the mutation testing of WS-BPEL programs.

Table 7 summarizes some basic statistics on the values of FDR. For each subject program and every possible mutation operator, we give the value range of FDR, represented by their minimum and maximum values in the square brackets. Note that the operators ECC and EAP only produced equivalent mutants, so the FDR for them is not available, denoted by "NA" in Table 7. Based on Table 7, we have observed that mutants generated by the ACI operators are normally very "weak", that is, it is very easy to kill these mutants. Since almost every test suite can achieve high FDR on these weak mutants, we cannot distinguish different test suites with respect to the fault-detection effectiveness. Thus, the ACI operator may not be very effective for the mutation analysis in

Table 6: MS on subject programs

Program name	MS	N_e	Mutant operators resulting in equivalent mutants
SupplyChain	100%	0	
SmartShelf	99.3%	11	ASI, ASF
SupplyCustomer	96.8%	5	ASF
LoanApproval	95.3%	5	ECC, EAP, CDE
CarEstimate	100%	7	ASI, AEL
TravelAgency	99.3%	11	ASF, ECC, EAP

practice.

Compared with ACI, some other operators are associated with varying FDR values. In other words, for some mutation operators (for example, ERR), it is difficult to anticipate how likely their associated mutants can be killed. Such operators would be quite effective in distinguishing different testing methods with respect to their fault-detection capabilities.

4.3 Answer to RQ3

We examined the detailed results and found two pairs of mutation operators that have the "subsuming" relations:

- AIE is subsumed by AEL.
- ASI is subsumed by ASF.

Tables 8 and 9 show some typical results of FDR for these operators, which can partially reflect their "subsuming" relations. As shown in Table 5, there were 11 and 71 mutants associated with AIE and ASI, respectively. Among them, only those AIE mutants associated with the AIE and AEL pairs and ASI mutants associated with the ASI and ASF pairs can be discarded for mutation testing, as summarized in Table 10. As a result, 11 AIE mutants and 61 ASI mutants were reduced in our experiment resulting to a reduction rate of 14.1% (i.e. (11+61)/(549-39)*100%). Reduction in the mutants can largely improve the efficiency of WS-BPEL mutation testing in practice.

Based on the above empirical evaluation, we further summarize some guidelines for mutation testing of WS-BPEL programs, especially when testing resources are limited. First, one should avoid employing mutants associated with those "subsumed" mutation operators. For instance, if ASF is used, then ASI is not necessary. Second, one should carefully check whether those "not so applicable" mutation operators can be applied. On one hand, selectively discarding some specific operators may help us improve the efficiency of mutation testing. On the other hand, it may also miss some mutants that might be quite useful in distinguishing the effectiveness of different test suites. Tester's experience may

Table 7: FDR on each subject program

(a) `SupplyChain`

Operator	Value range of FDR				
	T_x	T_y	T_z	T_u	T_w
ACI	[100%, 100%]	[100%, 100%]	[100%, 100%]	[100%,100%]	[100%, 100%]
AEL	[50%, 100%]	[40%, 100%]	[40%, 100%]	[42.1%, 100%]	[36.7%, 100%]
AIE	[50%, 50%]	[60%, 60%]	[60%, 60%]	[57.9%, 57.9%]	[63.3%, 63.3%]
ASI	[50%, 100%]	[40%, 100%]	[40%, 100%]	[42.1%, 100%]	[36.7%, 100%]
ASF	[50%, 100%]	[40%, 100%]	[40%, 100%]	[42.1%, 100%]	[36.7%, 100%]
ERR	[16.7%, 100%]	[10%, 100%]	[6.7%, 100%]	[5.3%, 100%]	[3.3%, 100%]
EIN	[100%, 100%]	[100%, 100%]	[100%, 100%]	[100%,100%]	[100%, 100%]
ELL	[16.7%,16.7%]	[20%, 20%]	[20%, 20%]	[21.1%,21.1%]	[26.7%, 26.7%]
CFA	[50%, 100%]	[40%, 100%]	[40%, 100%]	[42.1%, 100%]	[36.7%, 100%]
CDE	[50%, 50%]	[40%, 60%]	[40%, 60%]	[42.1%, 57.9%]	[36.7%, 63.3%]

(b) `SmartShelf`

Operator	Value range of FDR				
	T_x	T_y	T_z	T_u	T_w
ACI	[100%, 100%]	[100%, 100%]	[100%, 100%]	[100%, 100%]	[100%, 100%]
AEL	[18.2%, 100%]	[20%, 100%]	[20%, 100%]	[21.6%, 100%]	[24.4%, 100%]
AIE	[40.9%, 81.8%]	[37.1%, 80%]	[36%, 80%]	[33.8%, 82.4%]	[30%, 75.6%]
ASI	[18.2%, 100%]	[20%, 100%]	[20%, 100%]	[21.6%, 100%]	[24.4%, 100%]
ASF	[18.2%, 100%]	[20%, 100%]	[20%, 100%]	[21.6%, 100%]	[24.4%, 100%]
ERR	[0%, 100%]	[0%, 100%]	[0%, 100%]	[0%, 100%]	[0%, 100%]
EIN	[59.1%, 100%]	[60%, 100%]	[64%, 100%]	[63.5%, 100%]	[66.7%, 100%]
EAA	[63.6%, 63.6%]	[62.9%,62.9%]	[68%, 68%]	[56.8%,56.8%]	[55.6%, 55.6%]
EEU	[63.6%, 63.6%]	[62.9%,62.9%]	[68%, 68%]	[56.8%,56.8%]	[55.6%, 55.6%]
CFA	[18.2%, 100%]	[20%, 100%]	[20%, 100%]	[21.6%, 100%]	[24.4%, 100%]
CDE	[18.2%, 81.8%]	[20%, 80%]	[20%, 80%]	[21.6%, 78.4%]	[24.4%, 75.6%]

(c) `SupplyCustomer`

Operator	Value range of FDR				
	T_x	T_y	T_z	T_u	T_w
ACI	[100%, 100%]	[100%, 100%]	[100%, 100%]	[100%, 100%]	[100%, 100%]
AEL	[14.3%, 100%]	[8.3%, 100%]	[11.1%, 100%]	[12.5%, 100%]	[13.3%, 100%]
AIE	[42.9%, 42.9%]	[50%, 50%]	[50%, 50%]	[50%, 50%]	[53.3%, 53.3%]
ASI	[14.3%, 100%]	[8.3%, 100%]	[11.1%, 100%]	[12.5%, 100%]	[13.3%, 100%]
ASF	[57.1%, 100%]	[50%, 100%]	[50%, 100%]	[50%, 100%]	[46.7%, 100%]
AJC	[0%, 0%]	[0%, 0%]	[0%, 0%]	[0%, 0%]	[0%, 0%]
ERR	[14.3%, 100%]	[8.3%, 100%]	[11.1%, 100%]	[12.5%, 100%]	[13.3%, 100%]
EIN	[57.1%, 100%]	[50%, 100%]	[50%, 100%]	[50%, 100%]	[46.7%, 100%]
CFA	[14.3%, 100%]	[8.3%, 100%]	[11.1%, 100%]	[12.5%, 100%]	[13.3%, 100%]
CDE	[14.3%, 85.7%]	[8.3%, 91.7%]	[11.1%, 88.9%]	[12.5%, 87.5%]	[13.3%, 86.7%]
ELL	[0%, 0%]	[0%, 0%]	[0%, 0%]	[0%, 0%]	[0%, 0%]

Table 7: FDR on each subject program (continued)

(d) `LoanApproval`

Operator	Value range of FDR				
	T_x	T_y	T_z	T_u	T_w
ISV	[8.3%,8.3%]	[11.1%, 11.1%]	[12%, 12%]	[12.9%, 12.9%]	[12.5%, 12.5%]
ACI	[100%, 100%]	[100%, 100%]	[100%, 100%]	[100%, 100%]	[100%, 100%]
AEL	[0%, 100%]	[0%, 100%]	[0%, 100%]	[0%, 100%]	[0%, 100%]
ASI	[100%, 100%]	[100%, 100%]	[100%, 100%]	[100%, 100%]	[100%, 100%]
ASF	[8.3%, 8.3%]	[11.1%, 11.1%]	[12%,12%]	[12.9%, 12.9%]	[12.5%,12.5%]
ECN	[8.3%, 50%]	[5.6%, 50%]	[4%,48%]	[3.2%, 38.7%]	[2.5%,32.5%]
ERR	[8.3%, 83.3%]	[5.6%, 77.8%]	[4%, 72%]	[3.2%, 71%]	[2.5%, 65%]
EIU	[25%, 75%]	[22.2%, 66.7%]	[20%, 50%]	[19.4%, 58.1%]	[17.5%, 52.5%]
EIN	[8.3%, 83.3%]	[11.1%, 77.8%]	[12%, 72%]	[12.9%, 71%]	[12.5%, 65%]
EAN	[25%, 75%]	[22.2%, 66.7%]	[20%, 50%]	[19.4%, 58.1%]	[17.5%, 52.5%]
ECC	NA	NA	NA	NA	NA
EAP	NA	NA	NA	NA	NA
XMF	[0%, 0%]	[0%, 0%]	[0%, 0%]	[0%, 0%]	[0%, 0%]
CFA	[0%, 100%]	[0%, 100%]	[0%, 100%]	[0%, 100%]	[0%, 100%]
CDE	[8.3%, 75%]	[11.1%, 66.7%]	[12%, 60%]	[12.9%, 58.1%]	[12.5%, 52.5%]

(e) `CarEstimate`

Operator	Value range of FDR				
	T_x	T_y	T_z	T_u	T_w
ACI	[100%, 100%]	[100%, 100%]	[100%, 100%]	[100%, 100%]	[100%, 100%]
AEL	[100%, 100%]	[100%, 100%]	[100%, 100%]	[100%, 100%]	[100%, 100%]
ASI	[100%, 100%]	[100%, 100%]	[100%, 100%]	[100%, 100%]	[100%, 100%]
ASF	[100%, 100%]	[100%, 100%]	[100%, 100%]	[100%, 100%]	[100%, 100%]
APM	[100%, 100%]	[100%, 100%]	[100%, 100%]	[100%,100%]	[100%, 100%]
APA	[40%, 40%]	[35%, 35%]	[46.7%, 46.7%]	[47.2%,47.2%]	[47.5%, 47.5%]
EMD	[40%, 100%]	[45%, 100%]	[33.3%, 100%]	[30.6%,100%]	[27.5%, 100%]
EMF	[60%, 100%]	[65%, 100%]	[53.3%, 100%]	[52.8%,100%]	[52.5%, 100%]
CFA	[100%, 100%]	[100%, 100%]	[100%, 100%]	[100%, 100%]	[100%, 100%]

(f) `TravelAgency`

Operator	Value range of FDR				
	T_x	T_y	T_z	T_u	T_w
ISV	[50%, 50%]	[42.9%, 57.1%]	[50%, 50%]	[50%, 50%]	[45%, 55%]
ACI	[100%, 100%]	[100%, 100%]	[100%, 100%]	[100%, 100%]	[100%, 100%]
AEL	[50%, 100%]	[42.9%, 100%]	[50%, 100%]	[50%, 100%]	[45%, 100%]
AIE	[50%, 50%]	[42.9%, 57.1%]	[50%, 50%]	[50%, 50%]	[45%, 55%]
ASI	[50%, 100%]	[42.9%, 100%]	[50%, 100%]	[50%, 100%]	[45%, 100%]
ASF	[50%, 100%]	[42.9%, 100%]	[50%, 100%]	[50%, 100%]	[45%, 100%]
ERR	[25%, 100%]	[14.3%, 100%]	[10%, 100%]	[7.1%, 100%]	[5%,100%]
ECN	[25%, 50%]	[14.3%, 57.1%]	[10%, 50%]	[7.1%, 50%]	[5%, 50%]
EAN	[50%, 50%]	[42.9%, 42.9%]	[50%, 50%]	[50%,50%]	[55%, 55%]
EIU	[50%, 50%]	[42.9%, 42.9%]	[50%, 50%]	[50%,50%]	[55%, 55%]
EIN	[100%, 100%]	[100%, 100%]	[100%, 100%]	[100%, 100%]	[100%, 100%]
ECC	NA	NA	NA	NA	NA
EAP	NA	NA	NA	NA	NA
XMC	[0%, 0%]	[0%, 0%]	[0%, 0%]	[0%, 0%]	[0%, 0%]
CFA	[50%, 100%]	[42.9%, 100%]	[50%, 100%]	[50%, 100%]	[45%, 100%]
CDE	[50%, 50%]	[42.9%, 57.1%]	[50%, 50%]	[50%, 50%]	[45%, 55%]

Table 8: AIE vs. AEL on `SupplyCustomer`

Oper -ator	Mutant	Value of FDR				
		T_x	T_y	T_z	T_u	T_w
AIE	MU_1^{AIE}	42.9%	50%	50%	50%	53.3%
AEL	MU_1^{AEL}	42.9%	50%	50%	50%	53.3%
	MU_2^{AEL}	42.9%	50%	50%	50%	53.3%

Table 9: ASI vs. ASF on `SupplyChain`

Oper -ator	Mutant	Value of FDR				
		T_x	T_y	T_z	T_u	T_w
ASI	MU_1^{ASI}	100%	100%	100%	100%	100%
	MU_2^{ASI}	100%	100%	100%	100%	100%
	MU_3^{ASI}	100%	100%	100%	100%	100%
	MU_4^{ASI}	50%	40%	40%	42.1%	36.7%
ASF	MU_1^{ASF}	100%	100%	100%	100%	100%
	MU_2^{ASF}	50%	40%	40%	42.1%	36.7%

Table 10: A summary of mutant reduction in the experiment

Program name	Number of AIE and AEL mutant pairs	Number of ASI and ASF mutant pairs
SupplyChain	1	4
SmartShelf	4	19
SupplyCustomer	1	6
LoanApproval	0	1
CarEstimate	0	15
TravelAgency	5	16
Total	11	61

be critical when selecting or discarding a specific operator. Third, one should give the higher priorities to mutants associated with mutation operators who have relatively lower FDR values, as shown in Table 7. For instance, mutation operators such as ERR, ELL, ECN, XMC, and XMF should have higher priorities, while mutation operators such as ACI, AMP, and ASF should have lower priorities. The μBPEL tool developed by us supports the identification, selection, and prioritization of mutation operators for a given WS-BPEL program such that the proper and effective operators will be applied as early as possible to generate a sufficient number of useful mutants. In summary, the work presented in this paper not only explores mutation testing of WS-BPEL programs through presenting a mutation testing framework, but also enhances its efficiency through developing a mutation tool and conducting an empirical study.

5 Related Work

In this section, we discuss the closely related work and provide a comparison between our work and them.

It is well known that different mutation operators are associated with different applicabilities and effectiveness. For example, Ma et al. [24] have examined various mutation operators for Java programs. They found that it was easy to generate mutants based on some operators, while for other operators, it was impossible to construct any mutant for the given programs. Estero-Botaro et al. [14] conducted empirical studies to evaluate the WS-BPEL mutation operators based on three BPEL compositions. They observed that (1) some operators could be used to generate mutants for all three programs, while other operators were only applicable to specific programs; (2) several operators could generate the so-called "stillborn" mutants, which are syntactically incorrect and thus cannot be executed; (3) a large number (12) of operators could produce equivalent mutants; and (4) only two operators (AEL and ASI) could result in "weak" mutants, which were killed by any test case. In addition, Boubeta-Puig et al. [7] compared the mutation operators between WS-BPEL and other traditional programming languages, based on the analysis of their specific program structures. It was observed that due to the unique features of WS-BPEL, for only half of the 26 operators, we can find similar counterparts in other traditional languages, that is, half of the mutation operators are specific to WS-BPEL. On the other hand, some mutation operators that can be used for other languages are not applicable to WS-BPEL programs. It was also suggested that the WS-BPEL mutation operators could be further improved.

To evaluate the quality of the WS-BPEL mutation operators, Estero-Botaro [15] proposed a set of metrics. Among these metrics, the core is the mutant quality metric (Q_m), whose calculation equation is based on the total number of test cases in a test suite (T), the total number of mutant killed by the test set which can kill the mutant (denoted as X), and the number of non-equivalent mutants ($|M| - |E|$). For a given mutant set and a test suite, the larger X is, the smaller

Q_m is. This means that for the test set that can kill the mutant with a smaller value of Q_m must have a bigger value of X (that is to say, it can also kill many other mutants), indicating the mutant is harder to kill. Unlike Q_m, FDR (Fault Discovery Rate) is used to measure the difficulty of killing a mutant, which is defined as the ratio of the number of test cases that can kill the mutant (denoted as Y) to the total number of test cases ($|T|$). For a given mutant set and a test suite, the smaller Y is, the smaller FDR is, indicating its quality is better. For the mutant with a smaller value of FDR (i.e. better quality), it means that a smaller size of test set can kill this mutant, which in general indicates a smaller value of X, thus we have a larger Q_m (i.e. better quality). After comparing Q_m and FDR, we can have the following observations: (1) on one hand, mutant quality metrics proposed in [15] can provide somehow more accurate measurement on the quality of mutation operators than FDR used in our paper; on the other hand, their computational costs are much more heavy than that of FDR, which accordingly results in a low practicability; (2) these metrics and FDR hold the similar intuition and accordingly their measurement will deliver the similar evaluation results. Based on the above analysis and the fact that FDR and mutation scores are commonly used in mutation analysis, we did not use Q_m and other related metrics in [15] in our experiment.

In our study, we conduct a series of in-depth empirical studies based on six WS-BPEL programs to quantitatively evaluate the existing mutation operators. To our best knowledge, this empirical evaluation is the most comprehensive one, not only because of the largest scale and number of subject programs, but also due to the concrete conclusions (to be given in Section 4) that provide insightful and comprehensive guidance for WS-BPEL mutation testing in practice. More specifically, our study helps identify (*i*) a set of mutation operators that have very limited applicability to WS-BPEL programs; (*ii*) some "weak" operators that can only generate easy-to-kill mutants; and (*iii*) certain "subsuming" relations between some mutation operators. With the aid of these results, we can preclude a set of ineffective mutation operators, reduce the number of mutants, and thus decrease the cost of mutation testing. Similarly, a recent study revealed that a large proportion of mutants are duplicated (i.e. mutants have identical behaviours to each other rather than to the original program) [31]. Such identical mutants imply some subsumption relationship among individual mutants, which are fundamentally different from the subsumption of mutation operators proposed in our work. Though similar studies have been conducted on various types of programs [5, 30, 28, 21, 22], our work is the systematic study in selecting useful mutation operators for WS-BPEL programs.

There exist several tools to support the automatic generation of mutants for WS-BPEL programs. Domínguez-Jiménez et al. [12] developed a tool, namely GAmera, based on the 26 mutation operators proposed in [13]. GAmera first analyzes the WS-BPEL program under test to help users decide which mutation operators to be used in mutant generation. Though it is possible to generate all possible mutants, GAmera also makes use of the generic algorithm to select part of mutants to be generated. In addition, GAmera automatically executes the generated mutants for a given test suite, and identify whether a

mutant is killed, still alive, or syntactically incorrect. Another tool, namely WeMuTe, was developed by Boonyakulsrirung and Suwannasart [6] to implement the similar functionalities to GAmera. More recently, García-Domínguez and Medina-Bulo developed a more comprehensively mutant generation system called MuBPEL [18]. MuBPEL has the following features: (i) automatic mutant generation; (ii) execution of mutants with test cases; (iii) statistical results on killed mutants, alive mutants and invalid mutants; (iv) statement/branch/path coverage statistics via other plug-in tools. Compared with these existing tools, our μBPEL tool not only implements the automatic generation of mutants for WS-BPEL, but also provides the quantitative measurements of the effectiveness for any given test suite. Furthermore, μBPEL has the following new features: (i) an integrated environment that facilitates the integration and/or evaluation of a test case generation technique; (ii) a more comprehensive report for mutation testing results, i.e. including not only mutation scores, but also fault discovery rates; and (iii) a new component that is helpful for reducing the number of mutants.

6 Conclusion

WS-BPEL is an XML-based language that orchestrates multiple Web services to fulfill complex business processes. Due to the loose coupling and open environment for Web services, it is challenging to test WS-BPEL programs. Mutation testing, a fault-based testing technique, has been widely used to evaluate the fault-detection effectiveness of a test suite. In this paper, we reported a series of empirical studies on a family of mutation operators for WS-BPEL programs. Our experiments were conducted based on a framework and a tool that we proposed and developed for implementing the mutation testing of WS-BPEL programs. The experimental results on six WS-BPEL programs helped us identify which mutation operators would be applicable (or inapplicable) for generating mutants. It was also observed that some operators are very ineffective in terms of always generating weak mutants, and thus could be eliminated from the WS-BPEL mutation testing. In addition, we found some mutation operators that have subsuming relations, and part of them could be ignored for increasing the efficiency of mutation testing. Our research has resulted in comprehensive and insightful guidance for WS-BPEL mutation testing in practice.

For our future work, we are interested in continuing our efforts in the following aspects. First, we will explore optimization techniques for mutation testing of WS-BPEL programs, such as high order mutation testing. Second, we intend to include more complex and larger WS-BPEL programs for evaluation. Such work will benefit the broader area of the WS-BPEL service testing, as there does not exist a benchmark for evaluation in this area. Third, we are interested in extending the evaluation to report the time needed to apply mutation analysis and saved due to the subsumption of mutation operators. Finally, it is worthwhile to investigate the practical usefulness of mutation analysis in the context of WS-BPEL programs. One possible experiment is to compare mutation anal-

ysis and some coverage criteria, such as "branch" or "statement" coverage, in terms of their fault detection effectiveness and costs.

Acknowledgments

This research is supported by the National Natural Science Foundation of China under Grant No. 61370061, the Beijing Natural Science Foundation (Grant No. 4162040), the Beijing Municipal Training Program for Excellent Talents under Grant No.2012D009006000002, and the Fundamental Research Funds for the Central Universities under Grant No. FRF-SD-12-015A. All correspondence should be addressed to both C.-A Sun and H. Liu.

References

[1] ActiveVOS. Car repair estimation. In *http://www.activevos.com/developers/sample-apps*, 2012.

[2] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, pages 402–411, St. Louis, Missouri, USA, 15-21 May 2005. ACM.

[3] Apache. Apache ODE. In *http://ode.apache.org*, 2006.

[4] D. Appelt, C. D. Nguyen, L. C. Briand, and N. Alshahwan. Automated testing for SQL injection vulnerabilities: An input mutation approach. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA'14)*, pages 259–269, San Jose, CA, USA, 21-26 July 2014. ACM.

[5] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi. Toward the determination of sufficient mutant operators for C. *Software Testing, Verification and Reliability*, 11(2):113–136, 2001.

[6] P. Boonyakulsrirung and T. Suwannasart. WeMuTe – a weak mutation testing tool for WS-BPEL. In *Proceedings of The International MultiConference of Engineers and Computer Scientists (IMECS'12)*, pages 810–815, Hongkong, China, 14-16 March 2012. IAENG.

[7] J. Boubeta-Puig, I. Medina-Bulo, and A. García-Domínguez. Analogies and differences between mutation operators for WS-BPEL 2.0 and other languages. In *Proceedings of the 4th International Conference on Software Testing, Verification, and Validation Workshops (ICSTW'11)*, pages 398–407, Washington, DC, USA, 21-25 March 2011. IEEE Computer Society.

[8] T. Y. Chen, F.-C. Kuo, H. Liu, and W. E. Wong. Code coverage of adaptive random testing. *IEEE Transactions on Reliability*, 62(1):226–237, 2013.

[9] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[10] G. Denaro, M. Pezzè, and D. Tosi. Test-and-adapt: An approach for improving service interchangeability. *ACM Transactions on Software Engineering and Methodology*, 22(4):28:1–28:43, 2013.

[11] A. Derezinska. Quality assessment of mutation operators dedicated for C# programs. In *Proceedings of the 6th International Conference on Quality Software (QSIC'06)*, pages 227–234, Beijing, China, 27-28 October 2006. IEEE Computer Society.

[12] J.-J. Domínguez-Jiménez, A. Estero-Botaro, A. García-Domínguez, and I. Medina-Bulo. Gamera: A tool for WS-BPEL composition testing using mutation analysis. In *Proceedings of the 10th International Conference on Web Engineering (ICWE'10)*, pages 490–493, Vienna, Austria, 5-9 July 2010. Springer Berlin.

[13] A. Estero-Botaro, F. Palomo-Lozano, and I. Medina-Bulo. Mutation operators for WS-BPEL 2.0. In *Proceedings of the 21th International Conference on Software & Systems Engineering and their Applications (ICSSEA'08)*, pages 1–7, Paris, France, 9-11 December 2008. Telecom ParisTech.

[14] A. Estero-Botaro, F. Palomo-Lozano, and I. Medina-Bulo. Quantitative evaluation of mutation operators for WS-BPEL compositions. In *Proceedings of the 3rd International Conference on Software Testing, Verification, and Validation Workshops (ICSTW'10)*, pages 142–150, Paris, France, 7-9 April 2010. IEEE Computer Society.

[15] A. Estero-Botaro, F. Palomo-Lozano, I. Medina-Bulo, J.-J. Domínguez-Jiménez, and A. García-Domínguez. Quality metrics for mutation testing with applications to ws-bpel compositions. *Software Testing, Verification and Reliability*, 25:536–571, 2015.

[16] M. Fisher II, S. Elbaum, and G. Rothermel. An automated analysis methodology to detect inconsistencies in web services with WSDL interfaces. *Software Testing, Verification and Reliability*, 23(1):27–51, 2013.

[17] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA'10)*, pages 147–158, Trento, Italy, 12-16 July 2010. ACM.

[18] A. García-Domínguez and I. Medina-Bulo. MuBPEL. In *https://neptuno.uca.es/redmine/projects/sources-fm/wiki/MuBPEL.html*, 2015.

[19] R. Gopinath, C. Jensen, and A. Groce. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on*

Software Engineering (ICSE'14), pages 72–82, Hyderabad, India, 31 May-7 June 2014. ACM.

[20] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*, pages 435–445, Hyderabad, India, 31 May - 7 June 2014. ACM.

[21] Y. Jia and M. Harman. Higher order mutation testing. *Information and Software Technology*, 51(10):1379–1393, 2009.

[22] M. Kintis, M. Papadakis, and N. Malevris. Evaluating mutation testing alternatives: A collateral experiment. In *Proceedings of the 17th Asia-Pacific Software Engineering Conference (APSEC 2010)*, pages 300–309, Sydney, Australia, 30 Nov - 3 Dec 2010. IEEE Computer Society.

[23] H. Liu, F.-C. Kuo, D. Towey, and T. Y. Chen. How effectively does metamorphic testing alleviate the oracle problem? *IEEE Transactions on Software Engineering*, 40(1):4–22, 2014.

[24] Y.-S. Ma, M. J. Harrold, and Y.-R. Kwon. Evaluation of mutation testing for object-oriented programs. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, pages 869–872, Shanghai, China, 20-28 May 2006. ACM.

[25] A. J. Maâlej and M. Krichen. Study on the limitations of ws-bpel compositions under load conditions. *The Computer Journal*, 58(3):385–402, 2015.

[26] S. Mani, V. S. Sinha, S. Sinha, P. Dhoolia, D. Mukherjee, and S. Chakraborty. Efficient testing of service-oriented applications using semantic service stubs. In *Proceedings of the 2009 IEEE International Conference on Web Services (ICWS'09)*, pages 197–204, Los Angeles, California, USA, 6-10 July 2009. IEEE Computer Society.

[27] G. J. Myers. *The Art of Software Testing*. John Wiley and Sons, Hoboken, New Jersey, USA, second edition, 2004. Revised and Updated by T. Badgett and T. M. Thomas with C. Sandler.

[28] A. S. Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 351–360, Leipzig, Germany, 10-18 May 2008. ACM.

[29] OASIS Standard. Web services business process execution language version 2.0. In *http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html*, 2012.

[30] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology*, 5(2):99–118, 1996.

[31] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*, pages 936–946, Florence, Italy, 16-24 May 2015. IEEE.

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

[33] C.-A. Sun, G. Wang, K.-Y. Cai, and T. Y. Chen. Distribution-aware mutation analysis. In *Proceedings of 9th IEEE International Workshop on Software Cybernetics (IWSC'12)*, pages 170–175, Izmir, Turkey, 16-20 July 2012. IEEE Computer Society.

[34] C.-A. Sun, G. Wang, B. Mu, H. Liu, Z. Wang, and T. Y. Chen. Metamorphic testing for web services: Framework and a case study. In *Proceedings of the 9th International Conference on Web Services (ICWS'10)*, pages 283–290, Washington, DC, USA, 4-9 July 2011. IEEE Computer Society.

[35] C.-A. Sun, Y. M. Zhai, and Z. Zhang. BPELDebugger: An effective BPEL-specific fault localization framework. *Information and Software Technology*, 55(12):2140–2153, 2013.

[36] C.-A. Sun, Y. Zhao, L. Pan, H. Liu, and T. Y. Chen. Automated testing of WS-BPEL service compositions: A scenario-oriented approach. *IEEE Transactions on Services Computing*, null, in press. Acceptance date: 02/08/2015.

[37] C. J. Wright, G. M. Kapfhammer, and P. McMinn. The impact of equivalent, redundant and quasi mutants on database schema mutation analysis. In *Proceedings of the 14th International Conference on Quality Software (QSIC'14)*, pages 57–66, Allen, TX, USA, 2-3 October 2014. IEEE Computer Society.

[38] X. Yao, M. Harman, and Y. Jia. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*, pages 919–930, Hyderabad, India, 31 May - 7 June 2014. ACM.

[39] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei. Is operator-based mutant selection superior to random mutant selection? In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE'10)*, pages 435–444, Cape Town, South Africa, 1-8 May 2010. ACM.

[40] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid. Regression mutation testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA'12)*, pages 331–341, Minneapolis, MN, USA, 16-20 July 2012. ACM.