Fault Localisation for WS-BPEL Programs based on Predicate Switching and Program Slicing

Chang-ai Sun[a,*], Yufeng Ran[a], Caiyun Zheng[a], Huai Liu[b], Dave Towey[c], Xiangyu Zhang[d]

[a]*School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing 100083, China*
[b]*College of Engineering and Science, Victoria University, Melbourne 8001 VIC, Australia*
[c]*School of Computer Science, University of Nottingham Ningbo China, Zhejiang, 315100, China*
[d]*Department of Computer Science, Purdue University, West Lafayette, Indiana, USA*

Abstract

Service-Oriented Architecture (SOA) enables the coordination of multiple loosely coupled services. This allows users to choose any service provided by the SOA without knowing implementation details, thus making coding easier and more flexible. Web services are basic units of SOA. However, the functionality of a single Web service is limited, and usually cannot completely satisfy the actual demand. Hence, it is necessary to coordinate multiple independent Web services to achieve complex business processes. Business Process Execution Language for Web Services (WS-BPEL) makes the coordination possible, by helping the integration of multiple Web services and providing an interface for users to invoke. When coordinating these services, however, illegal or faulty operations may be encountered, but current tools are not yet powerful enough to support the localisation and removal of these problems. In this paper, we propose a fault localisation technique for WS-BPEL programs based on predicate switching and program slicing, allowing developers to more precisely locate the suspicious faulty code. Case studies were conducted to investigate the effectiveness of the proposed technique, which was compared with predicate switching only, slicing only, and one existing fault localisation technique, namely Tarantula. The experimental results show that the proposed technique has a higher fault localisation effectiveness and precision than the baseline techniques.

Keywords: Fault localisation, Debugging, Business Process Execution Language for Web Services, Web Services

1. Introduction

In recent years, Service-Oriented Architecture (SOA) [1] has been widely adopted to develop distributed applications in various domains. Web services, as basic units in SOA, are often developed and owned by a third party, and are published and deployed in an open and dynamic environment. Since a single Web service normally provides limited functionalities, multiple Web services are coordinated to implement complex and flexible business processes. Business Process Execution Language for Web Services (WS-BPEL) [2] is a widely recognised language for service compositions. In the context of WS-BPEL, all communications among Web services are via standard eXtensible Markup Language (XML) messages [3]. Compared with traditional programs written in C or Java, WS-BPEL programs have many new features. For instance, a WS-BPEL program is represented as an XML file, and dynamic behaviours of the program are embedded into structural XML elements; Web services composed by WS-BPEL programs may be implemented in hybrid programming languages; and WS-BPEL supports concurrency and synchronisation that is not common in the traditional programs.

The above unique features make the debugging of WS-BPEL programs significantly different from that of traditional programs. Unfortunately, very little research in this direction has been reported. In our previous work [4], we presented a block-based fault localisation framework for WS-BPEL programs, and synthesised the three well-known spectrum-based fault localisation techniques: Tarantula [5], Set-Union [6], and Code Coverage [7, 8]. We also conducted an empirical study to evaluate the effectiveness of the synthesized WS-BPEL-specific fault localisation techniques. The result showed that the Tarantula technique was the

---

[*] Corresponding author
Email address: `casun@ustb.edu.cn` (Chang-ai Sun)

most effective technique, demonstrated by the highest accuracy in localising the blocks that contain the faulty statement. However, these techniques could only report those suspicious faulty blocks without a deeper analysis inside the suspicious block.

There also exist some development platforms for WS-BPEL, such as *ActiveBPEL Designer* [9] and *Eclipse BPEL Designer* [10]. Unfortunately, these platforms usually only provide support for WS-BPEL syntax checking, while the assistance with logical errors that most developers expect is missing. If such assistance were available, perhaps as a plug-in to the platform, the debugging efficiency might be substantially improved.

In this work, we attempt to develop a technique to further improve the effectiveness and efficiency of fault localisation for WS-BPEL programs. In particular, we propose a new fault localisation technique for WS-BPEL programs, based on two popularly used techniques, namely predicate switching [11] and program slicing [12]. The proposed technique first employs predicate switching to narrow the range of blocks to be checked for the fault localisation, and then makes use of slicing to go more deeply into the block for a more precise analysis of the fault. In particular, we focus on the following unique challenges in applying predicate switching and program slicing into WS-BPEL programs.

- Predicate switching for WS-BPEL programs: For C, C++, or Java programs, predicate switching is normally implemented through instrumentation. However, WS-BPEL programs are basically the workflow specifications based on XML, and their executions normally rely on a specific interpreter. For instance, Apache ODE [13] is a popular WS-BPEL engine that compiles all standard BPEL elements: the compiled VxBPEL is represented as an object model containing all necessary resources for execution. A runtime component is responsible for the execution of compiled processes. Such an execution mode means that dynamic changes to WS-BPEL programs are not allowed. In contrast, the original implementation of predicate switching for C programs is based on Valgrind [1]. Valgrind supports dynamic instrumentation by calling the instrumentation functions. These functions, in turn, instrument the provided basic block and return the new basic block to the Valgrind kernel (refer to [11] for more details). Clearly, because no interfaces are reserved for calling instrumentation functions, it is not possible to implement dynamic instrumentation in Apache ODE. Furthermore, in this context, instrumentation is not a suitable solution for WS-BPEL programs. In order to implement an instrumentation function in WS-BPEL programs, it would be necessary to make significant modifications to the original WS-BPEL programs. Such modifications would definitely affect the whole program, including the partner link, variable, and interaction sections. Furthermore, these modifications would change the semantics of the original program, which violates the fundamental principle of instrumentation technique. On the contrary, the original instrumentation for predicate switching in C programs (again refer to [11]) does not face this challenge, because it implements the instrumentation functions in binaries, and the modifications to the original program include only introducing a new basic block. Finally, collecting the execution traces of instrumentation for WS-BPEL programs is also challenging. It is easy to collect execution traces in the context of C programs, which can be done by writing data into a file or memory. In contrast, WS-BPEL programs normally return a response message through a specific activity (i.e. reply), and there is no channel for throwing trace data.
- Dynamic program slicing for WS-BPEL programs: Variables in WS-BPEL programs can be either an atomic data type or a complex composite type whose definitions are normally distributed in various namespaces represented in XML files. Thus, dynamic slicing of WS-BPEL programs must be able to deal with recursive parsing and querying of composite variables. Furthermore, it is necessary to analyse the interpreter's logs in order to obtain the execution traces of the WS-BPEL program. These issues all pose challenges for the dynamic slicing of WS-BPEL programs.

Based on some new concepts, the above challenges are effectively addressed in our proposed technique.

In order to evaluate the performance of the proposed technique, we conducted a comprehensive empirical study where three WS-BPEL programs were used as object programs, and a total of 166 mutated versions were used to simulate various faults. The effectiveness and precision of the new technique were compared with the Tarantula technique, which had shown the best performance in fault localisation for WS-BPEL programs in our previous work [4]. Experimental results showed

---

[1] Valgrind is a well-known memory debugger and profiler for x86-kubyx binaries. For more details, please refer to: http://valgrind.org/

that the proposed technique demonstrates better effectiveness and higher precision than the previous technique.

The rest of this paper is organised as follows. Section 2 introduces underlying concepts of WS-BPEL, and some related fault localisation techniques. Section 3 describes the main idea of our new fault localisation technique. Details on how to apply the technique to WS-BPEL programs are also discussed. Section 4 describes an empirical study that was conducted to evaluate the proposed fault localisation technique. Section 5 presents the results of the empirical study, and offers an analysis. Section 6 discusses some important work related to our study. Finally, Section 7 concludes the paper, and discusses the future work.

2. Background

2.1. WS-BPEL

The Business Process Execution Language for Web Service (WS-BPEL) is a widely used language for composing Web services [2]. It can integrate multiple Web services to form a business process, and make this available in the form of Web services [4]. In this sense, a WS-BPEL Web service is actually a composite Web service whose invocation interface can be described using the Web Service Description Language (WSDL) [14]. WS-BPEL programs aim to integrate all Web services in one line to reduce the program redundancy, without requiring details of the actual implementation of the service [15].

WS-BPEL programs are usually composed of four sections: *variable section*, *partner link section*, *handler section*, and *interaction section* [4]. The variable section defines input and output messages. The partner link section describes the relationship among the WS-BPEL process and invoked Web services. The handler section declares the handlers when an exception or specific event occurs. The interaction section describes how external Web Services are coordinated to execute a business process.

The basic interaction unit of a WS-BPEL program is an *activity*, which can be either a *basic activity* or a *structured activity*. Basic activities describe an atomic execution step (such as *assign*, *invoke*, *receive*, *reply*, *throw*, *wait*, and *empty*); and structured activities are composed of several basic activities or other structured activities (such as *sequence*, *switch*, *while*, *flow*, and *pick*). Figure 1 shows an interaction segment of a WS-BPEL program.

```
<bpel:squence name=“main”>
  <bpel:receive name=“receiveInput”
        partnerLink= “client”/>
 <bpel:assign name= “a1”> … </bpel:assign>
  <bpel:flow name=“a2-4”>
      <bpel:invoke name=“a2”>… </bpel:invoke>
     <bpel:invoke name=“a3”>… </bpel:invoke>
   <bpel:invoke name=“a4”>… </bpel:invoke>
 </bpel:flow>
     …
  <bpel:reply name= “replyOutput”, …/>
</bpel:squence >
```

Figure 1: Illustration of the interaction of a WS-BPEL Program

2.2. Fault localisation techniques

Many fault localisation techniques have been proposed and examined empirically [16]. These techniques explore the fault localisation problem in different ways. The reported approaches include those based on program analysis, on program execution, on predicates, and also using data mining or machine learning [17]. Among them, spectrum-based fault localisation is a family of fault localisation techniques based on program execution that 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. We next introduce one representative spectrum-based technique that will be included for evaluation in our experiments reported in Section 4.

Jones [5] proposed a program execution-based fault localisation technique, using statistics, called Tarantula. Tarantula involves multiple test cases and executions, recording the pass and fail status for each program element a test case executes. The suspiciousness value is calculated according to Formula 1 below:

$$suspicion(s) = \frac{\frac{failed(s)}{totalfailed}}{\frac{passed(s)}{totalpassed} + \frac{failed(s)}{totalfailed}}, \tag{1}$$

where $passed(s)$ is the number of test cases that have executed the program element s with the output as expected; $failed(s)$ is the number of test cases that have executed the program element s with the output *not* as expected; $totalpassed$ is the total number of passing test cases; and $totalfailed$ is the total number of failing test cases. Program elements are ranked according to these suspicion values, with higher values indicating a higher likelihood of containing the fault. Limitations of the Tarantula method include that it requires a large set of tests, with the pass or fail status known, and that if either *totalpassed* or *totalfailed* is zero, then the formula is invalid.

In our previous work [4], we evaluated the performance of three traditional spectrum-based fault localisation techniques, namely Tarantula, Set-Union, and Code Coverage, on two WS-BPEL programs. The fault localisation effectiveness was mainly measured by the correctness percentage, which refers to the percentage of possible position sets that really contain the faulty statements. For one program (SupplyChain), Tarantula, Set-Union, and Code Coverage can successfully locate 7 (53.8%), 7 (53.8%), and 5 (38.5%) of 13 faults, respectively. For the other program (SmartShelf), Tarantula, Set-Union, and Code Coverage can successfully locate 10 (50%), 8 (40%), and 8 (40%) of 20 faults, respectively. Such observations implied that Tarantula was the most effective among these three fault localisation techniques. However, it should be noted that the performance of Tarantula (as well as Set-Union and Code Coverage) on WS-BPEL programs is not as good as that on traditional programs. For instance, Tarantula can achieve a score of 90% (i.e. the fault was found by examining less than 10% of the executed code) for the 55.7% faulty versions in seven C programs in the Siements suite [18]. In other words, there is a need for more advanced techniques specifically for localising faults in WS-BPEL programs.

In order to further improve the fault localisation effectiveness and efficiency of WS-BPEL programs, we explore predicate switching and program slicing-based fault localisation for WS-BPEL programs, and address the key issues of the proposed technique. We also compare the fault localisation effectiveness and efficiency of the proposed technique with that of predicate switching only, slicing only, and Tarantula, since Tarantula was evaluated to be the most effective technique in our previous work [4].

3. BPELswice: A Fault Localisation Technique for WS-BPEL Programs

Normally, traditional programs written in Java or C focus more on trivial operations on various data structures, from the simple data types such as char, integer, boolean, and real to the complex data types such as array, struct/union, pointer, and their composites. Different from them, WS-BPEL programs specify a workflow with coarse-grained activities, which usually involve simple operations such as invoking an external Web services or a variable assignment. The transitions between the activities are implemented through some common control logic such as sequence, optional, loop, and also newly introduced concurrency and synchronisation. These unique features of WS-BPEL programs pose challenges for fault localisation, and thus call for new techniques.

3.1. Overview

A fault is considered to be detected when a test case causes the program to have an incorrect output. The fault-revealing test case is also called the failed test case, the counterpart of which is called the successful test case that results in correct output. Each failed test case corresponds to the execution of a particular path, which can help us precisely localise the fault.

A typical program usually contains a number of branches, as part of its logical structure. These branches are controlled by some conditional slice, called a predicate, which evaluates to either true or false. If we force a predicate to change its true or false status, then we have a process called *predicate switching*. The goal of this process is to find a predicate that has strong influence on the data flow, and if the branch outcome of the predicate is switched and execution is continued, the output of the program may be changed from “incorrect” to “as expected”, thereby providing a valuable clue as to the location of the fault. Such a predicate, if it exists, is called a *critical predicate*. Statements that change the values of variables related to this critical predicate, which exist between it and the start of the program, are called *the backward slice*. Because the critical predicate may be strongly influenced by the backward slice, the analysis is necessary, and may provide further guidance to precisely locate the actual fault which caused the predicate’s incorrect status.

We hereby propose a fault localisation technique for WS-**BPEL** programs based on the predicate **sw**itching and backward sl**ice**s, which is abbreviated as *BPELswice* in the rest of the paper. Figure 2 shows the basic framework of the technique, for which it is assumed that at least one test case demonstrates the presence of a fault in the WS-BPEL program.

As shown in Figure 2, BPELswice includes the following five major steps:

1. *Parsing the WS-BPEL program* enables an enumeration of all the possible paths through the program and all predicates associated with each path, which facilitates the backward slice analysis.
2. *Predicate switching* revises a predicate of the WS-BPEL program and then deploys the revised WS-

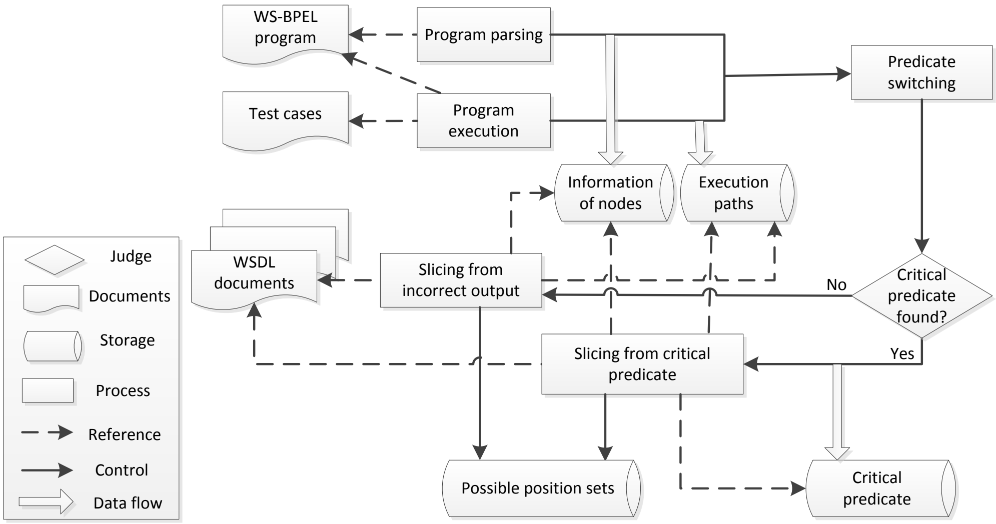


Figure 2: Framework of the proposed BPELswice technique

BPEL program for execution. In order to execute the different conditional part, this stage actually changes the predicate value from "true" to "false", or vice versa. The switching is implemented by negating the predicate of the original WS-BPEL program, and accordingly a WS-BPEL program variant is derived in this stage. During the switching, we can use some strategies to decide the switching ordering of the predicates.

3. *Execution* includes passing the failed test cases as input for the deployed WS-BPEL program variant and obtaining the actual output. This stage usually involves a WS-BPEL engine, such as Apache ODE [13], which produces a series of events in a log file. Through the analysis of the log file, one can extract all executed path nodes and variable's values during the current execution.
4. *Evaluation* involves comparison of the actual output with what was expected (i.e. the oracle). In this stage, the main goal is to observe how the predicate switching impacts on the output of the revised WS-BPEL program. If the output is different from the expected output, then we continue to switch the remaining ordered predicates (i.e. repeat *predicate switching* and *evaluation* steps). This switching process is repeated until the actual output becomes the same as the oracle, at which moment the critical predicate is found and then we can go to the next, *slicing*, step.
5. *Slicing* aims to further reduce the possible position set of the fault. In this step, the main task is to find backward slices between the critical predicate and the start node (i.e. the *receiveInput* node of the WS-BPEL program). Note that the backward slices are those nearest statements that directly affect the values of the elementary variables in the critical predicate. Through comparing the values achieved at run-time with what we expected, we can find the variable and its statement node that are different from the expected ones.

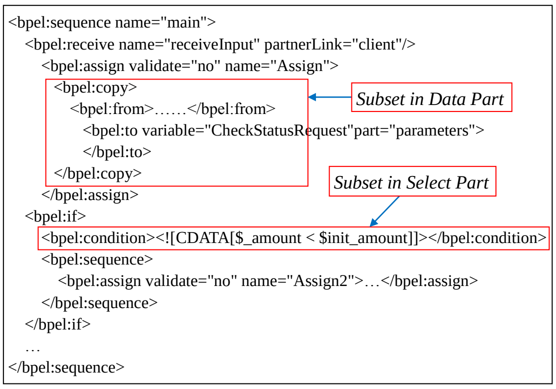
Each major step is detailed in the following sections.

3.2. *Parsing the WS-BPEL program*

It is necessary to parse the WS-BPEL program for user's understanding of its structure. The document root is critical for the parsing, and is the entry (the so-called "main node") to the program. Typically, there are two traverse methods to read a program: Depth-First Traversal and Breadth-First Traversal. The choice of traversal method does not affect the information we obtain from a WS-BPEL program. Once all the WS-BPEL node information has been obtained, it is inserted on a JTree [19], which shows all paths through the program.

The computation in a WS-BPEL program which calculates an output can be divided into two categories: the

Data Part (DP) and the *Select Part* (SP) [11]. The Data Part includes execution instructions which help in computing data values or defining variables that are involved in computing the output of the program. Sometimes these are important parts for backward slices. The Select Part includes instructions which cause the selection of program branches — for example, in the conditionals of “if”, “switch”, and “while” activities. Different program executions may involve different data slices, leading to generation of differently computed output values. Sometimes the conditions in Select Parts may depend on the values computed in the Data Parts. Furthermore, because the output values are determined by the selection of program branches, it is necessary to analyse the Select Parts to locate the faulty code. Figure 3 illustrates the Data Part and the Select Part in a sample WS-BPEL program.



```
<bpel:sequence name="main">
   <bpel:receive name="receiveInput" partnerLink="client"/>
      <bpel:assign validate="no" name="Assign">
        <bpel:copy>
           <bpel:from>……</bpel:from>
              <bpel:to variable="CheckStatusRequest"part="parameters">
              </bpel:to>
        </bpel:copy>
      </bpel:assign>
   <bpel:if>
     <bpel:condition><![CDATA[$_amount < $init_amount]]></bpel:condition>
      <bpel:sequence>
         <bpel:assign validate="no" name="Assign2">…</bpel:assign>
      </bpel:sequence>
   </bpel:if>
   …
</bpel:sequence>
```

Figure 3: Data Part and Select Part in WS-BPEL Program

In summary, the output of a program is influenced by two things in the code: the data dependence part, and the selection part. Altering code in the selection part may lead to a change in the output, which may enable us to track down where in the code an error was made. The selection part of the code which controls the flow is called a predicate, and a predicate whose outcome is changed, e.g., from false to true, resulting in a change to the overall program output changing to the expected output, is called a critical predicate. The question of how to find the critical predicate will be addressed in the following.

3.3. Predicate Switching

We aim to reduce the possible location range of the fault using the critical predicate technique [11], a central part of which is predicate switching. Predicate switching involves going through a sequence of predicates in any executed path, switching the boolean status of each (e.g. changing “true” to “false”), and examining the impact on the output: if the output changes to the expected one, then the critical predicate is the predicate whose status was most recently switched.

Before starting to switch predicates, we first order them, to reduce the amount of time required to identify the critical predicate. We hereby illustrate one typical and widely-used ordering strategy, called Last Executed First Switching ordering (LEFS) [11].

The LEFS ordering strategy is based on the observation that a failure (that is, the incorrect output different from expectation) usually occurs not far away from the execution of the faulty code. This leads to the decision to reverse the order of predicates such that the first one to be checked will be the most recently executed. Suppose that a test case t caused a failure of a WS-BPEL program. We first identify the sequence σ_n of predicates when executing t on the program, saying $p_1, p_2, ..., p_{n-1}, p_n$ where p_n is the predicate closest to the point of program failure, while p_1 is the predicate farthest from the point of failure. LEFS would therefore reorder σ_n to $\sigma'_n : p_n, p_{n-1}, ..., p_2, p_1$, with the result that the last encountered conditional branch is the first to be switched.

The detailed predicate switching procedure is described next. Given a WS-BPEL program $BPEL$, a failed test case t, its expected output O, and its associated LFES reordered predicate sequence $\sigma'_n : p_n, p_{n-1}, ..., p_2, p_1$, the following steps will be taken.

1. Set $i = n$, where i is used to index the order of predicates in σ'_n.
2. Mutate the WS-BPEL program $BPEL$ by negating p_i (e.g. change the Boolean value of p_i from TRUE to FALSE, or vice versa) and derive the mutated WS-BPEL program $BPELVariant$.
3. Redeploy $BPELVariant$.
4. Execute $BPELVariant$ with t and obtain its output O',
    - If O' is the same as O, p_i is identified as the critical predicate and the procedure is terminated;
    - If O' is different from O and i is equal to 1, there is no critical predicate and the procedure is terminated;
    - Otherwise, decrease i by one and go back to Step 2.

As an illustration, in Figure 4, the output remains incorrect until p_{n-4} is switched. In other words, p_{n-4} is the critical predicate. Note that it is possible that none

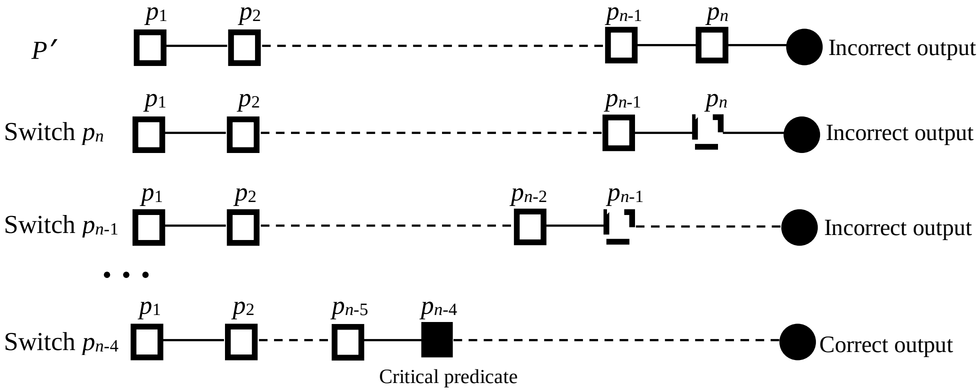


Figure 4: Illustration of searching critical predicate

of the predicates could be identified as the critical predicate even after all of them have been switched. In our BPELswice technique, the backward slicing will be implemented either from the critical predicate (if it is identified) or from the end node of the program (if no critical predicate is identified).

Unlike the original predicate switching, our method employs a mutation-based technique to implement predicate switching, i.e. we first mutate a predicate in the WS-BPEL program according to the LEFS strategy and then redeploy the mutated WS-BPEL program. This treatment is different from the implementation of predicate switching for traditional programs. For instance, a dynamic instrumentation technique was used to implement predicate switching for C programs [11]. We do not believe instrumentation is suitable for WS-BPEL programs, because it may introduce significant changes to the original program and require modifications of the existing WS-BPEL engine, as discussed in Section 1.

3.4. Execution

During the execution process, we use an Apache ODE engine [13] to redeploy the WS-BPEL service. The ODE engine is capable of talking to Web services, sending and receiving messages, handling data manipulation and performing error recovery, as described in the process definition. It supports both long and short duration process executions to facilitate all services in an application, and enables WS-BPEL programs to be invoked as services. The execution process consists of two steps: service deployment and process compiling. During the service deployment, the package (including all individual services and description files) is copied to the server, where the ODE engine deploys the service and outputs the deploy.xml file. We used Apache Tomcat for this. The second step (process compiling) helps ensure that the service can be executed and invoked successfully. When a new WS-BPEL program variant is copied into the process directory of the engine, previous ones are deleted, and the engine redeploys it at once.

Communication between the client and server depends on Apache Axis2 [20], which encapsulates the test case in a soap message, and sends it to the server. After sending these messages, Axis2 parses and passes them to *receiveInput* in the WS-BPEL program. The engine then assigns values and invokes some services to complete the operation. As soon as the engine produces the output, Axis2 parses and encapsulates it into a soap message, and passes it to the client side.

3.5. Evaluation

Following the execution, an output is produced, and one of the following two situations exists: either this is the first execution, in which case the next step is to begin the switching process; or it is necessary to compare the output with what was expected to confirm whether or not the current predicate under evaluation is the critical predicate — output being consistent with expectation means that the critical predicate has been found. If the outputs remain different, then the critical predicate has not yet been found, and the predicate switching process continues.

3.6. Slicing

Once the critical predicate has been identified, we first examine whether the fault is located in the critical predicate. If yes, the fault localisation process can be terminated. Otherwise, the related program slices should be examined to localise the most suspicious statements related to the fault. The procedure for identifying these slices is discussed as follows.

The input of the slicing procedure includes

- $P(T)$, the execution trace when executing the program P with a failed test case. $P(T) = < X_1, X_2, ..., X_i, ..., X_n >$, where X_i is a node in the execution path, $1 \leq i \leq n$, and n is the total number of nodes in the path.
- X_q, the critical predicate, which is an element of $P(T)$, that is, $1 \leq q \leq n$.
- $US(X_i)$, the set of def-use pairs of a node X_i. $US(X_i) = \{< v^i_{d1}, v^i_{u1} >, < v^i_{d2}, v^i_{u2} >, \cdots\}$. It can be obtained from the parsing of WS-BPEL program and WSDL documents, as detailed in the following.

According to the basic concepts of data flow analysis [21], the *def* of a variable refers to the operation where a concrete value is allocated to the variable, while the *use* operation means that the variable is utilised either in a calculation (*c-use*) or a predicate (*p-use*). Since the data flow in WS-BPEL is different from that in traditional programming languages, we have the following definitions for the def and use in WS-BPEL programs.

- In WS-BPEL, the def of a variable normally happens at
  - The *Receive* activity: the “variable” attribute,
  - The *Invoke* activity: the “outputVariable” attribute, and
  - The *Assign* activity: the left part of the expression in the “to” element.
- In WS-BPEL, the c-use of a variable normally happens at
  - The *Invoke* activity: the “inputVariable” attribute,
  - The *Reply* activity: the “variable” attribute, and
  - The *Assign* activity: the right part of the expression in the “from” element.
- In WS-BPEL, the p-use of a variable normally happens at
  - The *Switch* activity: the Boolean expression in the “case” statement,
  - The *While* activity: the “condition” attribute, and
  - The *If* activity: the “condition” attribute.

```
backwardSlice ($P(T)$, $X_q$, $US(X_i)$)
{
  set a set $C = \Phi$;
  initialise a set $V = \{v^q_{u1}, v^q_{u2}, ...\}$, where $v^q_{uj}$ is
    used in $X_q$;
  for each $v^q_{uj}$ ($j$ = 1, 2, ...) {
    if ($v^q_{uj}$ is of basic type) {
      for each $X_k$ ($k = q$-1, $q$-2, ...) in $P(T)$ {
        if ( $v^k_{dh} == v^q_{uj}$ ) {
          /*<$v^k_{dh}$, $v^q_{uj}$> is a def-use pair. */
          add $X_k$ into $C$;
          break;
        }
      }
    }
  else {
    find all variables of basic type $\{p_1, p_2, ...\}$
      that compose $v^q_{uj}$;
      for each $p_l$ ($l$ = 1, 2, ...){
        for each $X_k$ ($k = q$-1, $q$-2, ...) in $P(T)$ {
          if ( $v^k_{dh} == p_l$ ) {
            add $X_k$ into $C$;
            break;
          }
        }
      }
  }
  return $C$;
}
```

Figure 5: Procedure of backward slicing

The basic procedure of backward slicing is as follows. Assume that the critical predicate is X_q, and all the variables used in X_q are $\{v^q_{u1}, v^q_{u2}, \cdots\}$. For each v^q_{uj}, we search the nearest node backward (that is, first X_{q-1}, then X_{q-2}, ...) until we find the node X_k where v^q_{uj} (or the variables of basic type that comprise v^q_{uj}) is defined (that is, the latest definition of v^q_{uj} before it is used in X_q). The collection of all X_k will be the set of slices that are expected to contain the fault. The detailed backward slicing procedure is given in Figure 5. Note that, unlike programs written in traditional languages, WS-BPEL programs normally involve variables of complex type. For a variable v^q_{uj} of complex type, we need to first decompose it into variables of basic type ($\{p_1, p_2, \ldots\}$ in Figure 5), and find the nearest node where each p_l is defined. Such a process, as shown by Statements 12 to 21 in Figure 5, is specific to WS-BPEL.

3.7. Illustration

We use the SmartShelf WS-BPEL program to illustrate how our BPELswice technique works. SmartShelf is composed of 53 nodes and 13 services, as shown in Figure 6. Every service uses some parameters which come from the WS-BPEL program ReceiveInput part: SmartShelf uses the three parameters named "name", "amount" and "status", the first two of which come from ReceiveInput, and the third from the ReadStatus.

For the CheckStatus service, SmartShelf invokes different services according to the variable "amount", and refers to the status to judge whether the product is expired or not, returning "*Expired commodity has been replaced*" or "*Commodity is in good status*", respectively. The CheckLocation service returns whether or not the product is available on the shelf. If not, it invokes another service to correct the location. The CheckQuantity service checks whether or not a sufficient amount of the good is available, returning either "*Quantity is sufficient*", or "*Warehouse levels are insufficient, alert staff to purchase*", as appropriate.

The test case with "name"= *candy*, and "amount" = 100, "*candy*&&100", can be passed as input to this WS-BPEL service. Because of the initial settings in the database, the executed flow structure sequence involves the services "CheckStatus", "CheckLocation" and "CheckQuantity". The executed runtime path is $Path_1 = \{1-6, 7-10, 16-17, 18-21, 27-28, 29-32, 33-37, 38-42, 48-53\}$, as obtained from the ODE engine. Next, the predicate nodes are identified in the path: the predicate set in $Path_1$ is $Pre_1 = \{10, 21, 32, 37\}$. These predicates can then be switched. In this study, we use the LEFS ordering strategy, which reorders Pre_1 to $Pre'_1 = \{37, 32, 21, 10\}$. The predicate details for $Path_1$ are shown in Table 1.

When switching the predicates according to the order in Pre'_1, two steps are involved. First, we identify the target predicate, and switch its status (e.g., $\_status = 0$ becomes $\_status\ != 0$). Then, the previous WS-BPEL file is deleted and replaced with this new one, and the ODE engine is used to redeploy the service.

After each predicate is switched, the same test case ("*candy*&&100") is input, and the output is recorded and compared with the expected output. Suppose that when the predicate with "$\_location = 0$" (that is, If2 at node 21) is switched to "$\_location\ != 0$", the resulting output becomes the same as the expected output — this predicate is therefore the critical predicate, and is therefore suspected to be strongly connected to the fault. If the fault is found in this predicate, the localisation process can be successfully ended. Otherwise, some backward slices can be captured at runtime using the ODE engine. In this example, from the critical predicate (at node 21), we search backward and identify Assign16 (node 20), CheckLocation (node 18), and Assign (node 3) as the slices that are expected to contain the fault.

4. Empirical Study

We conducted a series of experiments to evaluate the performance of the proposed BPELswice technique. The empirical study was designed as follows.

4.1. Research Questions

Our empirical study was mainly focused on answering the following four research questions.

RQ1 How effectively does BPELswice localise the fault in a WS-BPEL program?

One critical criterion for evaluating the effectiveness of a fault localisation technique is whether or not it can successfully identify the statements/blocks that contain the fault. In our study, we applied BPELswice in the debugging of hundreds of mutants of three object programs, and then measured its effectiveness through examining how many faults BPELswice successfully localised.

RQ2 How precise is BPELswice in fault localisation?

In addition to the high fault localisation effectiveness, it is also important for a technique to have a high precision in the localisation. One naive way to maximise the high fault localisation effectiveness is to simply identify all executable statements in the program, which must contain the fault. Obviously, such a way is useless and very inefficient. In order to improve the efficiency of debugging, the number of statements identified by a fault localisation technique should be as small as possible, without eliminating the faulty statement. In our study, we measured the precision of BPELswice by evaluating how many statements it identified for each mutant.

RQ3 How quickly can BPELswice localise the fault?

If a technique is very time-consuming, its applicability would be greatly hindered. Therefore, it is necessary to investigate the execution time of

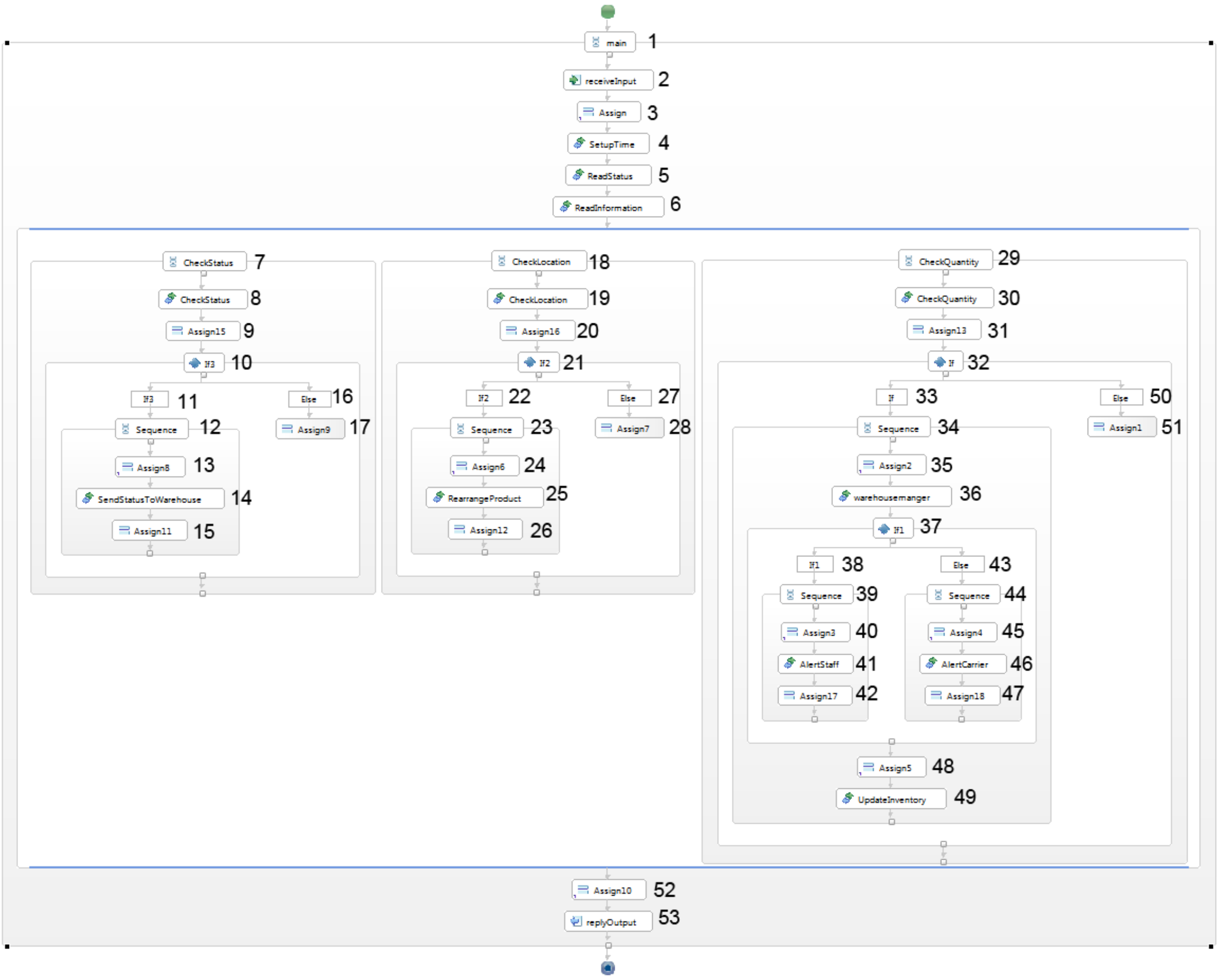


Figure 6: Structure of the SmartShelf WS-BPEL program

Table 1: SmartShelf's Predicate Set in $Path_1$ for Test Case "candy&&100"

If3 (node 10)	If2 (node 21)	If (node 32)	If1 (node 37)
$\$\_status = 0$	$\$\_location = 0$	$\$\_amount < \$init\_amount$	$warehouseManagerReturn < \$init\_amount$

the proposed BPELswice technique to see whether it can provide a high effectiveness and precision within reasonable time.

RQ4 How many times does BPELswice need to switch predicates?

It can be naturally conjectured that the computational overhead in BPELswice is mainly related to the predicate switching process. In this study, we will evaluate the concrete number of predicate switches conducted by BPELswice on WS-BPEL programs.

4.2. Variables and Measures

4.2.1. Independent variable

The independent variable of our empirical study is the fault localisation technique. Our proposed BPELswice technique was selected for this variable. Since BPELswice is based on the predicate switching and backward slicing techniques, it is natural to select each of these two techniques (denoted *switchOnly* and *sliceOnly* in the rest of the paper) as the baseline techniques for comparison. In addition, we selected the Tarantula technique as another baseline technique for a better comparison with previous work [4]. As discussed in Section 2, Tarantula was the most effective fault localisation technique for WS-BPEL programs, compared with the Code Coverage and Set-Union techniques [4].

4.2.2. Dependent variable

The dependent variable relates to the measurement. In order to answer RQ1, we used the metric *success rate* to measure the fault localisation effectiveness. Given a number of faults, the success rate of a fault localisation technique is defined as the percentage of the number of successfully localised faults out of the total number of faults, that is,

$$\text{success rate} = \frac{\text{number of localised faults}}{\text{total number of faults}} \times 100\%, \quad (2)$$

where a fault is said to be localised by a technique if the statement identified by the technique contains the fault. Obviously, the higher the success rate is, the more effective a technique is in fault localisation.

In order to answer RQ2, we measured the precision with the metric *identification ratio*. Given a fault, if a technique successfully localises it, the precision of the technique is defined as the percentage of the number of statements identified by the technique against the total number of statements of the program, that is,

$$\begin{aligned}&\text{identification ratio}\\ &= \frac{\text{number of identified statements}}{\text{total number of statements}} \times 100\%.\end{aligned} \quad (3)$$

Intuitively speaking, the lower the identification ratio is, the more precise a technique is in fault localisation.

For RQ3, we made use of the *runtime* to measure how fast a fault localisation technique can be. For the BPELswice technique, the runtime is composed of the time for finding the critical predicate and that for backward slicing. Note that the execution time of the Tarantula technique is not available, as the ranking of the program elements is purely based on suspiciousness values, which are calculated based on the information at the testing stage. Therefore, we only compared the runtime for BPELswice, switchOnly, and sliceOnly.

For RQ4, we measured the number of predicate switches (denoted N_{ps}) that BPELswice requires when it is used to localise a fault in a WS-BPEL program. Note that there is no predicate switching process in sliceOnly and Tarantula, and the switchOnly technique should have exactly the same N_{ps} as BPELswice. Hence, we will only present the N_{ps} of BPELswice.

4.3. Object Programs

We selected three WS-BPEL programs, SmartShelf, TravelAgency, and QuoteProcess, as the objects in our empirical study. The basic information of these three programs is summarised in Table 2, which gives the lines of code (LOC), the number of implemented external services (#Services), and the number nodes (#Nodes) of each program. The SmartShelf program accepts the input parameters, including the name and amount of commodity, implements various services, and returns the status of commodity, the location of shelf, the quantity in warehouse, etc. The TravelAgency program is basically a booking system, involving the selection of travel plan, hotel reservation, ticket booking, and banking. The QuoteProcess program is used to simulate the user's selection of activities: it selects different activities according to user's input parameters. Note that TravelAgency is a sample WS-BPEL program, which was first introduced by OASIS [22], and is currently available at [23], while SmartShelf and QuoteProcess, on the other hand, were created by us according to third party business scenarios (available at [24]).

Table 2: Object programs

Program	LOC	#Services	#Nodes
SmartShelf	579	13	53
TravelAgency	427	6	24
QuoteProcess	400	6	21

4.4. Mutant Generation

For each object program, we generated a family of mutants using the MuBPEL tool [25]. Each mutant contains one and only one fault, which was seeded by applying a mutation operator into a certain statement. In MuBPEL, there are totally over 30 mutation operators [26]. However, not all the operators were applicable to each object program. As a matter of fact, due to the unique features of WS-BPEL, normally only a few mutation operators can be applied to a WS-BPEL program [27]. In our study, we used seven, nine, and seven mutation operators to generate mutants for SmartShelf, TravelAgency, and QuoteProcess, respectively. After the mutants were generated, we found that some could not be executed due to syntactic errors, so we eliminated them. There were also several so-called equivalent mutants, that is, they always showed the same execution behaviours as the basic programs. These equivalent mutants were also eliminated. Finally, our empirical study used 57, 56, and 53 mutants for SmartShelf, TravelAgency, and QuoteProcess, respectively. The basic mutant generation information is summarised in Table 3.

Table 3: Mutant generation

Program	#Operators	#Mutants
SmartShelf	7	57
TravelAgency	9	56
QuoteProcess	7	53

4.5. Test Case Generation

We used random testing to generate a large amount of test cases. For a given WS-BPEL program, we first parsed the program to obtain the constraints on the input parameters. Random test data were generated for each input parameter, with the condition that the random data must satisfy the constraints. The random test case generation process was repeated until each fault was detected by at least one test case. Here, a fault was considered to be detected when a test case caused the relevant mutant to show different behaviour (more specifically, different output) to the basic program.

4.6. Experiment Procedure

Our empirical study was conducted on a laptop with a Windows 7 64-bit Operating System, an 8-core 3.4GHz CPU (i7-4790), and 16G memory. The experiments were run on Tomcat 6 and the ODE engine [13], which can provide a lot of runtime information, including the execution path, predicates on the path, and the variable values. All this information can assist us in finding the fault. The basic procedure of the experiments is as follows.

1. Start the ODE engine, and deploy one mutant of a WS-BPEL program.
2. Execute the test cases.
3. Obtain the critical information, including execution path, predicate sets, and actual output.
4. Find the test case that kills the mutant (that is, that causes the actual output to differ from the expected output).
5. Initiate predicate switching to identify the critical predicate.
6. If the critical predicate is identified, execute backward slicing from the critical predicate to localise the fault.
7. If the critical predicate is not identified, execute backward slicing from the wrong output.
8. Repeat the above steps until all mutants of each object program have been executed.

4.7. Threats to Validity

4.7.1. Internal validity

The main threat to internal validity relates to the implementation. The programming for implementing the BPELswice involved a moderate amount of work. Two of our authors conducted the programming work, one mainly for predicate switching, and the other mainly for slicing. All the source code was reviewed, cross-checked, and tested by different individuals. We are confident that the proposed BPELswice technique was correctly implemented, and thus the threat to internal validity has been minimised.

4.7.2. External validity

The threat to external validity is concerned with the selection of object programs and the fault types under study. In our study, we selected three representative WS-BPEL programs as the objects. These programs implement different functionalities, invoke different services, and have different scopes. Although we have endeavoured to maximise the diversity of object programs, we cannot guarantee that the results obtained from these three programs can be generally applied to any other WS-BPEL program. In addition, due to the nature of WS-BPEL programs, we could not study all possible fault types (mutation operators) for WS-BPEL, so it is also uncertain whether our conclusion is applicable to those fault types that were not investigated in this study. Moreover, it was assumed in this study that only one fault exists in a mutant. However, it would be very unlikely that the BPELswice technique could not work effectively for multiple faults — this is something that requires further empirical investigation.

4.7.3. Construct validity

There is little threat to construct validity in our study. The two metrics used in this study, success rate and identification ratio, are very straightforward in measuring the fault localisation effectiveness and precision.

4.7.4. Conclusion validity

In our experiments, we examined the performance of BPELswice based on 166 mutants of three object programs. A large amount of test cases were generated randomly. Therefore, a sufficient amount of experimental data was collected to guarantee the reliability of our results. In this sense, the threat to conclusion validity is very small.

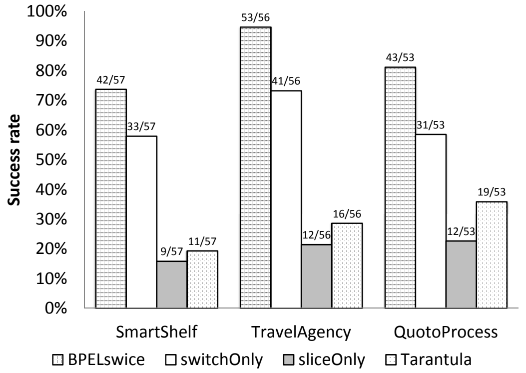


Figure 7: Comparison of success rates for BPELswice, switchOnly, sliceOnly, and Tarantula

5. Results and analysis

5.1. RQ1: Effectiveness

The experimental results of the success rates for BPELswice, switchOnly, sliceOnly, and Tarantula are given in Figure 7.

Based on Figure 7, we can observe that among 57, 56, and 53 faults (each in one mutant) for SmartShelf, TravelAgency, and QuoteProcess, respectively, BPELswice could successfully localise 42, 53, and 43 faults, giving success rates of 73.68%, 94.64%, and 81.13%, which were consistently the highest among the four fault localisation techniques. These results clearly show that BPELswice was much more effective than the other three techniques in the fault localisation for WS-BPEL programs.

We also investigated the faults that BPELswice failed to localise. We found that all these faults are of the types of “remove an activity” and “remove an element”. Since BPELswice is based on the execution path, it cannot localise fault types related to “removal”. For the same reason, Tarantula cannot localise these “removal” fault types either. In other words, all the faults localised by Tarantula were also successfully localised by BPELswice, but some faults localised by BPELswice could not be localised by Tarantula.

5.2. RQ2: Precision

The identification ratios for BPELswice, switchOnly, sliceOnly, and Tarantula are summarised in Figure 8. In these figures, box plots are used to display the distribution of identification ratios for one fault localisation technique on one object program. In each box, the lower, middle, and upper lines represent the first quartile, median, and the third quartile values of the identification ratios, respectively, while the lower and upper whiskers denote the min and max values, respectively; in addition, the mean value is depicted with the round dot.

Figure 8 clearly shows that the BPELswice technique outperformed the other three techniques in terms of identification ratio. In other words, BPELswice was the most precise technique for fault localisation of WS-BPEL programs.

In summary, the proposed BPELswice technique was not only effective in localising most faults, but also precise in identifying a small number of statements that contain the faults. This high efficacy and high precision of BPELswice would, in turn, significantly improve the cost-effectiveness of debugging WS-BPEL programs.

5.3. RQ3: Runtime

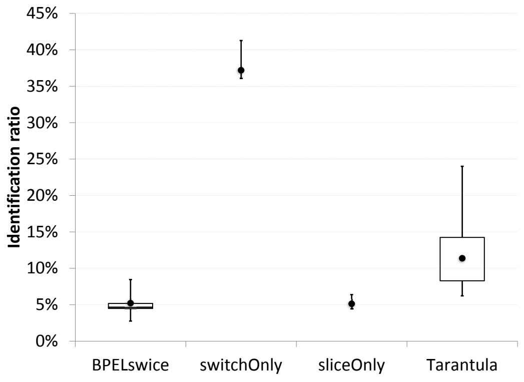
In our experiments, we executed a fault localisation technique three times on every mutant, and then recorded the runtime. The runtime results for BPELswice, switchOnly, and sliceOnly are given in Figure 9, where box plots are again used to show the distribution of the runtime (in seconds) for each object program. As discussed in Section 4.2, the runtime of Tarantula is not included here.

It can be observed from Figure 9 that, compared with BPELswice and switchOnly, sliceOnly has a very short runtime. This implies that predicate switching is much more time-consuming than slicing. Also due to the negligible runtime of slicing, the overall runtime of BPELswice is almost the same as that of switchOnly. We can also observe that the complete fault localisation procedure of BPELswice only takes tens of seconds. Such a runtime is acceptable, especially considering the high effectiveness and precision of BPELswice.

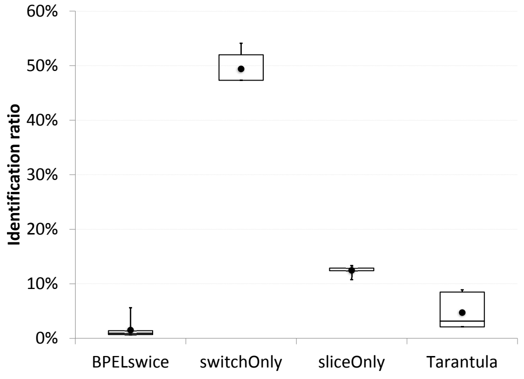
5.4. RQ4: Predicate Switches

As shown in the previous section, predicate switching is the main computation overhead in BPELswice. For each mutant where BPELswice successfully localised the fault, we also recorded the number of predicate switches (N_{ps}) for each mutant, as summarised in Table 4. As discussed in Section 4.2, we only report the N_{ps} of BPELswice, as switchOnly has exactly the same values.

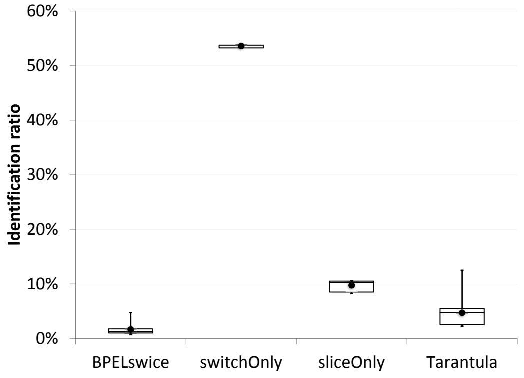
In the third column of Table 4, we report the number of mutants that are associated with the same value of N_{ps} (in the second column). In addition, the fourth column gives the range of the runtime (in seconds) for



(a) SmartShelf

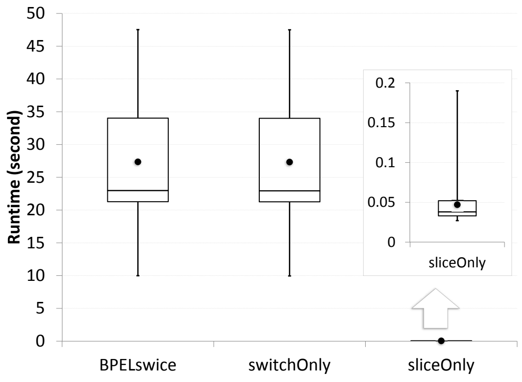


(b) TravelAgency

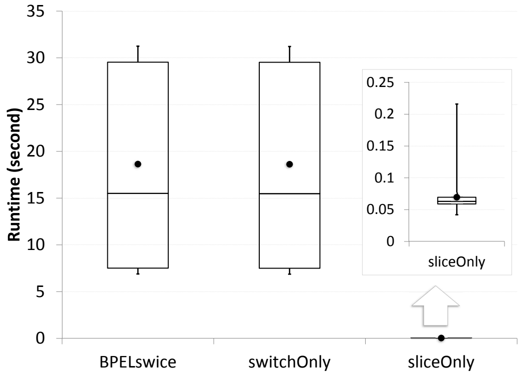


(c) QuoteProcess

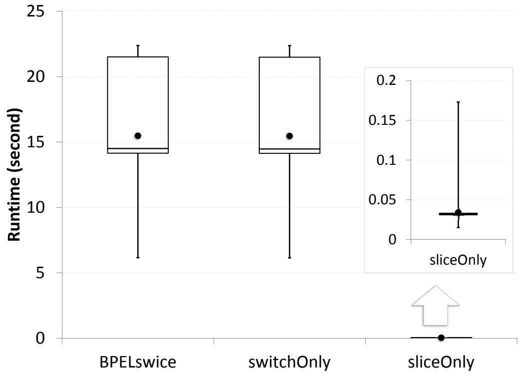
Figure 8: Comparison of identification ratios for BPELswice, switchOnly, sliceOnly, and Tarantula



(a) SmartShelf



(b) TravelAgency



(c) QuoteProcess

Figure 9: Comparison of runtime for BPELswice, switchOnly, and sliceOnly

executing BPELswice with the same N_{ps}. It can be observed that BPELswice only needed to switch predicates several times, with a maximum number of four times. In addition, there is a strong correlation between the value of N_{ps} and the runtime, which is intuitively as expected — the more predicates switched, the longer BPELswice executed.

6. Related work

How to effectively locate faults reported by testing is a crucial activity in debugging. A lot of effort on this topic has been made and a number of fault localisation techniques have been proposed [16][28][17]. These techniques explore the fault localisation problem in different ways. The reported approaches include those based on program analysis, on program execution, and also using data mining or machine learning. Next, we discuss related work in terms of slicing-based fault localisation techniques, predicate-based fault localisation techniques, and fault localisation techniques for WS-BPEL programs.

6.1. Slicing-based fault localisation techniques

One category of fault localisation techniques is based on program analysis techniques such as program slicing [29], symbolic execution [30], and formal methods. *Program slicing* is the most widely used one for debugging aids. The principle of program slicing is to strip a program of statements without influence on a given variable at a given statement [29]. The idea of program slicing-based fault localisation is: given a program p and a variable v at a statement i where a fault appears, the suspicious slice is the statements that directly affect the value of v at i — this eliminates those that have no impact on the value of v at i. A pioneering study was reported by Weiser [31] and showed the evidence that programmers slice when debugging.

Table 4: N_{ps} of BPELswice

Program	N_{ps}	#Mutants	Runtime range (second)
SmartShelf	1	8	[9.98, 11.95]
	2	15	[19.05, 23.50]
	3	10	[31.57, 35.29]
	4	9	[41.21, 47.53]
TravelAgency	1	18	[6.89, 7.66]
	2	9	[14.60, 17.64]
	3	7	[21.94, 23.01]
	4	19	[28.84, 31.25]
QuoteProcess	1	9	[6.16, 8.27]
	2	19	[13.36, 15.13]
	3	15	[21.41, 22.39]

Generally, program slicing can be either static or dynamic: the former is only based on the source code, while the latter works on a specific execution of the program (for a given execution trace). Fault localisation based on static program slicing analyzes the data flow and control flow of the program statically to reduce the search scope of faults [32], and its fault localisation precision is low since no other information than source code is used. Fault localisation based on dynamic program slicing introduces more precise slicing criteria for a particular execution and the search scope of faults can be further reduced [33]. Many efficient slicing algorithms have been proposed, and these algorithms may be used to further improve the efficiency of program slicing-based fault localisation techniques. For instance, *roBDD* is an efficient forward dynamic slicing algorithm using reduced ordered binary decision diagrams [34]. Recently, Wen [35] proposed program slicing spectrum to improve the effectiveness of statistical fault localisation methods, where the program slice is first used to extract dependencies between program elements and refine execution history, and then the suspiciousness of each slice is calculated to locate the fault based on statistical indices.

In our study, a backward dynamic program slicing technique was used to further improve the efficiency of locating faults in WS-BPEL programs. Our approach first analyzes the execution trace from an WS-BPEL engine and then extract suspicious statements via data flow analysis. Only those statements that have a direct impact on the value of elementary variables in the critical predicate are chosen. Our approach addressed the challenges due to the fact that the syntax, data structure, and execution mode of WS-BPEL programs are different from that of traditional programs.

6.2. Predicate-based fault localisation techniques

The other category of fault localisation techniques is based on program execution. Typically, such techniques make use of a program execution spectrum obtained in software testing to locate the suspicious elements. These techniques count the executions of program elements in different executions, and use the ratio of a program element being exercised in a failed execution and that in a passed execution to calculate the suspiciousness of the program element. Naish *et al.* [16] surveyed 33 different formulas for the suspiciousness calculation. The existing approaches work either at the level of statements or based on predicates.

Fault localisation techniques at the level of statements, such as Tarantula [5] and Code Coverage [7], often rely on statistics and need both successful and failing test cases to work. However, because they depend more on the pass or fail status of the test cases, and do not consider the static structure of the program, these methods may face other challenges. Renieris and Reiss [6] proposed a Set-Union technique based on neighboring queries which separated the failing program slices from the successful slice sets, deleting slices that appeared in both successful and failed execution paths, thereby generating a suspicious statement set.

Fault localisation techniques based on predicates first instrument predicates in programs, and then capture and/or sample execution behaviours to efficiently identify fault-relevant program elements. Among these techniques, some are based on statistics, and others are based on predicate switching. Typical predicate-based statistical fault localisation techniques include: Liblit *et al.* [36] ranked the predicates according to the probability that the program under study will fail when those predicates are observed to be true; Nainar *et al.* [37] used compound Boolean predicates to locate faults; Zhang *et al.* [38] investigated the impact of short-circuit evaluations on the effectiveness of existing predicate-based techniques; Chilimbi *et al.* [39] used path profiles as fault predictors to locate faults; Hao *et al.* [40] proposed a self-adaptive fault localisation algorithm which dynamically selects the intensity of each predicate based on predicate execution information analysis.

Predicate-switching based fault localisation was first proposed by Zhang *et al.* [11]: it focuses on a failed run corresponding to single input for fault localisation. Unlike existing statistical techniques, the idea of this technique is to forcibly switch a predicate's outcome at runtime and alter the control flow until the program produces the desired output. By examining the switched predicate, the cause of the fault can then be identified. Although predicate-switching based fault localisation significantly reduces the search space of potential state changes, the overhead for locating a program with scaled predicates may still be high. Wang and Liu [41] proposed a hierarchical multiple predicate switching method which restricts the search for critical predicates to the scope of highly suspect functions identified by employing spectrum-based fault localisation techniques. The predicate switching technique has demonstrated good efficiency for locating faults in *C* programs.

In our study, the predicate switching technique was employed to narrow the search scope of blocks within WS-BPEL programs. In particular, we implemented the predicate switching technique through mutating predicates rather than instrumentation, which is very different from previous studies [11, 41].

6.3. Fault localisation techniques for WS-BPEL programs

As mentioned before, WS-BPEL programs demonstrate new features that are not common in traditional programs, and accordingly suffer from new fault types. In our previous work [4], we explored the fault localisation issue of WS-BPEL programs and proposed a block-based fault localisation framework. We synthesized three well-known spectrum-based fault localisation techniques within the framework (Tarantula [5], Set-Union [6], and Code Coverage [7]), and evaluated the effectiveness of the synthesized techniques using two WS-BPEL programs. Although such techniques were empirically evaluated to be effective in previous studies [18], however, their effectiveness was not as good as expected when they were used for the fault localisation of WS-BPEL programs.

In this study, we addressed the above problem with a new fault localisation technique for WS-BPEL programs which combines predicate switching with program slicing. We empirically evaluated and compared the effectiveness and precision of the proposed technique with the Tarantula technique, which showed the best performance in the synthesized techniques for WS-BPEL programs in our previous work.

7. Conclusion

WS-BPEL program have many new features and also suffer from new types of faults when compared with traditional programs that are written in C, C++, or Java. In this paper, we have presented a novel fault localisation technique, BPELswice, for WS-BPEL programs. The proposed technique is composed of two main components: the predicate switching method, which is used to greatly reduce the state search space of faulty codes through looking for so-called critical predicates, and the dynamic backward slicing method, which is used to improve the fault localisation precision through dataflow analysis of execution traces of WS-BPEL programs. Three case studies were conducted to evaluate the fault localisation performance of the proposed technique in terms of correctness and precision, and compare its performance with that of predicate switching only, slicing only, and Tarantula, which was considered to be the most effective one for WS-BPEL programs. The experimental results show that the proposed BPELswice technique had a higher fault localisation effectiveness and precision than predicate switching only, slicing only, and Tarantula. In other words, this study proposes a more effective fault localisation technique for WS-BPEL programs.

This study advances the state of the art for the fault localisation of WS-BPEL programs in the following ways: (i) we propose a new fault localisation framework to further improve the fault localisation effectiveness of WS-BPEL programs, considering new features of WS-BPEL programs (i.e. a new style of programs); (ii) we address the challenging issues related to when predicate switching is used for WS-BPEL programs, where the predicate switching mechanism is very different from that which was developed for C programs [11]; (iii) we report on the technical treatment of the backward dynamic slicing technique for WS-BPEL programs, which is significantly different from that for traditional programs; (IV) we provide a comprehensive evaluation and comparison of the proposed technique with existing techniques in this field.

In our future work, we are interested in the following directions: (i) extending the proposed framework to cover other sections of WS-BPEL programs (the current one only consider the faults in the interaction section of WS-BPEL programs); (ii) developing techniques to enable isolation of the faults in the level of WS-BPEL programs or invoked services; and (iii) investigating the differentiation of types of locating fault among the different fault localisation techniques.

Acknowledgements

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 Aeronautical Science Foundation of China (Grant No. 2016ZD74004). Dave Towey acknowledges the financial support from the Artificial Intelligence and Optimisation Research Group of the University of Nottingham Ningbo China, the International Doctoral Innovation Centre, the Ningbo Education Bureau, the Ningbo Science and Technology Bureau, and the University of Nottingham. The authors are grateful to An Fu and Cuiyang Fan from University of Science and Technology Beijing for their help in conducting extra experiments reported in this work, and to the anonymous referees for their helpful comments on an earlier version of this paper.

References

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

[2] Eviware, Web Services Business Process Execution Language Version 2.0, `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html` (2012).

[3] W3C, Extensible Markup Language (XML), `http://www.w3.org/XML/` (2008).

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

[5] J. A. Jones, Fault localization using visualization of test information, in: Proceedings of 26th International Conference on Software Engineering, IEEE Computer Society, 2004, pp. 54–56.

[6] M. Renieres, S. P. Reiss, Fault localization with nearest neighbor queries, in: Proceedings of 18th IEEE International Conference onAutomated Software Engineering, IEEE, 2003, pp. 30–39.

[7] W. E. Wong, Y. Qi, L. Zhao, K.-Y. Cai, Effective fault localization using code coverage, in: Proceedings of 31st Annual InternationalComputer Software and Applications Conference (COMPSAC 2007), Vol. 1, IEEE, 2007, pp. 449–456.

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

[9] Rose India, Activebpel designer v1.0, http://www.roseindia.net/eclipse/plugins/webservices/ActiveBPEL-Designer.shtml.

[10] Eclipse, Bpel designer v1.0, http://www.eclipse.org/bpel/.

[11] X. Zhang, N. Gupta, R. Gupta, Locating faults through automated predicate switching, in: Proceedings of 28th international conference on Software engineering, ACM, 2006, pp. 272–281.

[12] B. Xu, J. Qian, X. Zhang, Z. Wu, L. Chen, A brief survey of program slicing, ACM SIGSOFT Software Engineering Notes 30 (2) (2005) 1–36.

[13] Apache, Apache ODE, `http://ode.apache.org/` (2006).

[14] H. Haas and A. Brown, W3C, Web Services Glossary, `http://www.w3.org/TR/ws-gloss/` (2004).

[15] M. N. Huhns, M. P. Singh, Service-oriented computing: Key concepts and principles, IEEE Internet Computing 9 (1) (2005) 75–81.

[16] L. Naish, H. Lee, K. Ramamohanarao, A model for spectra-based software diagnosis, ACM Transactions on Software Engineering and Methodology 20 (3) (2011) 11:1–11:32.

[17] W. Wong, R. Gao, Y. Li, R. Abreu, F. Wotawa, A survey on software fault localization, IEEE Transactions on Software Engineering (2016) in press.

[18] 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.

[19] Java Platform, Standard Edition 7 API Specification, `http://docs.oracle.com/javase/7/docs/api/javax/swing/JTree.html` (2016).

[20] Apache, Apache Axis2, `http://axis.apache.org/axis2/java/core/` (2012).

[21] U. Khedker, A. Sanyal, B. Sathe, in: Data Flow Analysis: Theory and Practice, CRC Press, 2009.

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

[23] Linagora, Travel Agency, `https://research.linagora.com/display/easiestdemo/Travel+Agency` (2016).

[24] ActiveVOS, ActiveVOS sample applications, `http://www.activevos.com/developers/sample-apps` (2017).

[25] A. Estero-Botaro, F. Palomo-Lozano, I. Medina-Bulo, J. Dominguez-Jimenez, A. Garcia-Dominguez, Quality metrics for mutation testing with applications to WS-BPEL compositions, Software Testing, Verification and Reliability 25 (5-7) (2014) 536–571.

[26] 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, 2008.

[27] C.-A. Sun, L. Pan, Q. Wang, H. Liu, X. Zhang, An empirical study on mutation testing of WS-BPEL programs, The Computer Journal 60 (1) (2017) 143–158.

[28] X. Xie, T. Y. Chen, F.-C. Kuo, B. Xua, A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization, ACM Transactions on Software Engineering and Methodology 22 (4) (2013) 31:1–31:40.

[29] M. Weiser, Program slicing, IEEE Transactions on Software Engineering 10 (4) (1984) 352–357.

[30] J. C. King, A symbolic execution and program testing, Communications of the ACM 19 (7) (1976) 385–394.

[31] M. Weiser, Programmers use slices when debugging, Communications of the ACM 25 (7) (1982) 446–452.

[32] J. R. Lyle, M.Weiser, Automatic program bug location by program slicing, in: Proceedings of the Second International Conference on Computers and Applications, 1987, pp. 877–883.

[33] H. Agrawal, R. A. DeMillo, E. H. Spafford, Debugging with dynamic slicing and backtracking, Software - Practice and Experience 23 (6) (1993) 589–616.

[34] X. Zhang, R. Gupta, Y. Zhang, Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams, in: Proceedings of 26th International Conference on Software Engineering, IEEE Computer Society, 2004, pp. 502–511.

[35] W. Wen, Software fault localization based on program slicing spectrum, in: Proceedings of 34th International Conference on Software Engineering, IEEE Press, 2012, pp. 1511–1514.

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

[37] P. 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), 2007, pp. 5–15.

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

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

[40] P. Hao, Z. Zheng, Z. Zhang, Self-adaptive fault localization algorithm based on predicate execution information analysis, Chinese Journal of Computers (2014) 500–510.

[41] X. Wang, Y. Liu, Automated fault localization via hierarchical multiple predicate switching, Journal of Systems and Software 104 (2015) 69–81.