Fault Management of Business Processes in the Services Cloud

PhD Thesis

Muhammad Adeel Zahid

2016-03-0053

Advisor: Dr. Basit Shafiq

School of Science and Engineering

Lahore University of Management Sciences

May 15, 2023

FAULT MANAGEMENT OF BUSINESS PROCESSES IN THE SERVICES CLOUD

by

MUHAMMAD ADEEL ZAHID

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Syed Babar Ali School of Science and Engineering at the Lahore University of Management Sciences, Lahore, Pakistan

May 15, 2023

PhD Committee
Dr. Basit Shafiq
Dr. Shafay Shamail
Dr. Jaideep Vaidya
Dr. Naveed Arshad
Dr. Naveed ul Hassan
Dr. Kashif Kifayat

Dedicated to my parents

Lahore University of Management Sciences

School of Science and Engineering

CERTIFICATE

We hereby recommend that the dissertation prepared under our supervision by ***Muhammad Adeel Zahid*** titled, ***"Fault Management of Business Processes in the Services Cloud"*** be accepted in partial fulfillment of the requirements for the degree of Ph.D.

Committee Members

Dr. Basit Shafiq (LUMS) ________________________________

Dr. Shafay Shamail (LUMS) ________________________________

Dr. Jaideep Vaidya (Rutgers University) ________________________________

Dr. Naveed Arshad (LUMS) ________________________________

Dr. Naveed ul Hassan (LUMS) ________________________________

Dr. Kashif Kifayat (Air University) ________________________________

Acknowledgments

This dissertation work is supported by the Higher Education Commission (HEC) and Planning Commission of Pakistan and LUMS FIF grant.

I would like to express my utmost gratitude to my esteemed advisors Dr. Basit Shafiq and Dr. Shafay Shamail for their invaluable guidance and support. They have been phenomenal in shaping the course of this research.

I would also like to thank members of my Ph.D. committee, Dr. Naveed Arshad, Dr. Jaideep Vaidya, and Dr. Naveed ul Hassan for their insights and suggestions.

Finally, I owe a special debt of gratitude to my mother who has always been a source of strength for me, to my late father who had always supported and encouraged me, to my brother for making it possible for me to attend the University, and to my wife and children for being kind and supportive throughout these years.

List of Publications

Journal

1. Muhammad Adeel Zahid, Basit Shafiq, Jaideep Vaidya, Ayesha Afzal, and Shafay Shamail, "Collaborative Business Process Fault Resolution in the Services Cloud". *IEEE Transactions on Services Computing*, vol. 16, no. 1, pp. 162-176, 1 Jan.-Feb. 2023.
doi: 10.1109/TSC.2021.3112525.2021.

Conference

1. Muhammad Adeel Zahid, Basit Shafiq, Shafay Shamail, Ayesha Afzal and Jaideep Vaidya,"BP-DEBUG: A Fault Debugging and Resolution Tool for Business Processes". *IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*, 2022, pp. 1306-1309.
doi: 10.1109/ICDCS54860.2022.00143.

2. Muhammad Adeel Zahid, Ahmed Akhtar, Basit Shafiq, Shafay Shamail, Ayesha Afzal and Jaideep Vaidya, "An Integrated Framework for Fault Resolution in Business Processes". *2022 IEEE International Conference on Web Services (ICWS)*, pp. 266-275.
doi: 10.1109/ICWS55610.2022.00048.

Abstract

The emergence of cloud and edge computing has enabled rapid development and deployment of Internet-centered distributed business process applications. Several platforms and tools exist that enable users to develop distributed business process (BP) applications by composing relevant service components in a plug-and-play manner. However, these platforms and tools do not ensure that the developed BP application is fault-free. BP designers may make errors due to lack of semantic understanding of these web services, or incorrect and/or incomplete workflow specifications. Such errors result in faults which may not necessarily be identified at design or development stage and may only manifest during execution. Therefore, there is a need to develop diagnostic capabilities for Internet-centered BP development.

The objective of this dissertation is to develop an integrated framework for fault resolution of Business Processes (BPs) in the services cloud environment. Given a faulty BP and a set of test cases as input, this integrated framework detects and resolves design-time faults in the BP. The framework includes a prototype implementation that graphically presents a comprehensive fault report including the list of faults and their fixes. The BP designer inspects the fixed BP according to their criteria and accepts or rejects the modifications. The framework aims to empower the BP designer to detect and resolve faults in BPs. It not only produces the fixes of discovered faults but also generates the implementation-level code of the fixed BP, which can be readily deployed for execution.

There are two main underlying approaches and a hybrid approach that perform fault resolution. The first approach is Efficient Generate-and-Validate (EGV) which is an improvement of basic generate-and-validate (G&V) automated program repair. EGV generates the candidate fixes by applying mutations on the faulty BP. It employs fault localization and slicing to keep the number of candidate fixes manageable. The second approach is Collaborative Fault Resolution (CFR) which aims to utilize information from existing fault-free BPs that use similar services to resolve faults in a user-developed BP. CFR is based on association analysis of pairwise transformations

between a faulty BP and existing BPs to identify the smallest possible set of transformations to resolve the fault(s) in the faulty BP. Finally, we propose a hybrid approach that performs fault resolution by analyzing a faulty BP in isolation as well as by comparing it with other BPs using similar services. This hybrid approach results in improved accuracy and broader coverage of fault types. We also perform an extensive experimental evaluation to compare the effectiveness of the proposed approach using a dataset of 208 faulty BPs.

Contents

List of Figures **v**

List of Tables **vii**

1 Introduction **1**

1.1 Objective 1

1.2 Research Motivation 2

1.3 Common Faults in Business Processes 5

1.3.1 Expression Fault 5

1.3.2 Branching Fault 6

1.3.3 Control Flow Fault 6

1.3.4 Variable Assignment Fault 6

1.4 Problem Statement 7

1.5 Methodology 8

1.6 Contributions 9

1.7 Outline of Dissertation 11

2 Related Work **12**

2.1 Fault Localization 12

2.1.1 Static Techniques for Fault Localization 13

2.1.2 Dynamic Fault Localization 14

2.1.3 Fault Localization in SOA Context 18

2.1.4 Recent Trends in Fault Localization 19
2.1.5 Summary 21
2.2 Fault Resolution 22
2.2.1 Mutation-based Fault Resolution 22
2.2.2 Pattern-based Fault Resolution 24
2.2.3 Machine-learning-based Fault Resolution 26
2.2.4 Summary 29
2.3 Service Composition Testing 30
2.3.1 Unit Testing 30
2.3.2 Integration Testing 33
2.3.3 Regression Testing 37
2.3.4 Summary 38

3 An Efficient Generate & Validate Approach for Fault Resolution 40
3.1 Generate and Validate (G&V) 40
3.2 BP Fault Resolution – Problem Formulation 45
3.3 Efficient G&V approach (EGV) for fault resolution 48
3.3.1 Fault Localization 51
3.3.2 BP Slicing 51
3.3.3 Candidate Fix Generation 52
3.3.4 Validation of Candidate Fixes 54
3.4 Experimental Evaluation 54
3.4.1 Dataset 54
3.4.2 Results 55
3.5 Chapter Summary 56

4 Fault Resolution with Collaborative and Hybrid Approaches 57
4.1 Introduction 57
4.2 Proposed Approach for Collaborative Fault Resolution 60

4.2.1 Fault Localization . . . 63
4.2.2 Comparison with Existing BPs . . . 64
4.2.3 Association Rule Mining on Transformations . . . 66
4.2.4 Computation complexity . . . 72
4.3 Hybrid Approach for Fault Resolution . . . 73
4.4 Experimental Evaluation (CFR) . . . 74
4.4.1 Random Fault Injection . . . 76
4.4.2 User Developed BPs . . . 84
4.4.3 Parameter sensitivity . . . 85
4.5 Experimental Evaluation (Hybrid) . . . 86
4.5.1 Discussion . . . 87
4.6 Chapter Summary . . . 88

5 Prototype Implementation 89
5.1 Introduction . . . 89
5.2 BP-DEBUG: Architecture and Implementation . . . 90
5.2.1 Specification of the Faulty BP and the Test suite . . . 91
5.2.2 Fault Resolution . . . 94
5.2.3 Code Generation and BP Deployment . . . 95
5.3 Related Tools and Research Prototypes . . . 96
5.4 Chapter Summary . . . 98

6 Conclusion and Future Work 99
6.1 Research Contributions . . . 99
6.2 Challenges . . . 101
6.3 Future Work . . . 102
6.3.1 Recommendation System . . . 103
6.3.2 Collaborative Resolution of Configuration Faults . . . 103
6.3.3 Collaborative Fault Resolution in Heterogeneous Environment . . . 103

6.3.4 Privacy-preserving Resolution of BP Faults 104

List of Figures

1.1 Services cloud environment for BP composition. 4
1.2 A sales-order BP from e-commerce domain 5
1.3 Expression fault introduced in Fig. 1.2 5
1.4 Branching fault introduced in Fig. 1.2 6
1.5 Control flow fault introduced in Fig. 1.2 6
1.6 A simple sales-order BP from e-commerce domain with data flow 7
1.7 A simple sales-order BP from e-commerce domain depicting variable assignment fault 8
1.8 High-level architecture of the fault resolution methodology 9

3.1 An incorrect divide function in C++ 41
3.2 First candidate fix for Listing 3.1 41
3.3 Second candidate fix for Listing 3.1 42
3.4 Third candidate fix for Listing 3.1 42
3.5 Fourth candidate fix for Listing 3.1 43
3.1 First candidate fix for BP in Fig. 1.5 43
3.2 Second candidate fix for BP in Fig. 1.5 44
3.3 Third candidate fix for BP in Fig. 1.5 44
3.4 Example of an e-commerce BP graph 46
3.5 An example BP graph (G_f) from e-commerce sale order processing domain. 47
3.6 Efficient G&V fault resolution approach for BPs. 49
3.7 Slice of faulty BP G_f. 52

4.1 Collaborative BP fault resolution approach 60
4.2 Faulty sales order BP (G_f) and its subgraph (shown in rectangular box) used for pair-wise comparison 66
4.3 Subgraphs of existing BPs used for comparison with the faulty BP 66
4.4 SCC graph resulting from association analysis on transformations listed in Table 4.1 72
4.5 Comparison between the percentage of BPs fixed vs. number of iterations and the iteration-wise average time taken for fault resolution per BP 78
4.6 Comparison between the percentage of BPs fixed vs. similarity and service overlap based on top 3 rules 79
4.7 Execution time comparison of EGV and CFR. 82
4.8 Execution time comparison of H1, H2, and CFR. 88

5.1 Architectural overview of the BP-DEBUG System 90
5.1 Control flow of the BP of Fig. 1.2 91
5.2 Data flow of *createOrder* service in the BP of Fig. 1.6 92
5.3 Test suite for the BP of Fig. 1.6 93
5.4 Visual representation of *fops* and control flow faults and their fixes in BP-DEBUG . 95
5.5 Identification and representation of data flow faults in BP-DEBUG fault report interface 96

List of Tables

1.1 BP-specific fault categories and types 3

3.1 Candidate fixes for BP graph in Fig. 3.5. 53
3.2 Accuracy of EGV and Basic G&V. 55

4.1 Results of pair-wise graph comparison for the BPs depicted in Fig. 4.2 and Fig. 4.3 67
4.2 Statistics of existing BPs 76
4.3 α vs. average number of rules 77
4.4 Accuracy results over synthetic dataset 78
4.5 Accuracy of EGV and CFR. 81
4.6 Accuracy comparison of CFR with G&V fault repair approach 82
4.7 Evaluation results over user-developed BP dataset 85
4.8 Accuracy of H1, H2 and CFR. 87

Chapter 1

Introduction

1.1 Objective

Our objective, in this dissertation, is to develop an integrated framework for fault resolution of business processes in the services cloud environment. The proposed framework enables an organization to debug its business processes in an automated or semi-automated manner. Specifically, an organization submits its faulty business process along with correctness criteria (usually in the form of a test suite) and the proposed framework generates the candidate fixes and validates them against the provided criteria. This relieves the BP designer of manual debugging that involves significant time and technical understanding of business process constructs, development, and debugging practices. With the proposed fault resolution capability at its disposal, the focus of an organization can shift from low-level technical details to high-level business objectives.

The proposed framework is developed for novice users with little technical expertise in debugging business processes. In particular, it is suitable for small and medium enterprises that cannot dedicate sufficient resources to debugging. Additionally, experienced users can also use our system to accelerate the debugging cycle for their business process development.

The proposed framework, not only, helps organizations fix faults in their business processes but also generates a fault report with a list of observed faults and corresponding fixes. The framework is also capable of generating the source code of the fixed business process after it has been reviewed

by the BP designer and supports the deployment of the fixed business process on the server.

1.2 Research Motivation

Cloud computing and Internetware software paradigm have enabled rapid development and deployment of Internet-centric distributed applications, including distributed workflows, business processes, and Web mashups [1]. These applications are developed using computation, data, and storage services available in the cloud data centers and enterprise networks as well as large numbers of IoT and edge computing devices providing diverse sensory and computation services. Increasingly, there are new platforms and tools [2, 3, 4, 5] available that can facilitate automated or semi-automated development of such distributed applications by composing relevant service components in a plug-and-play manner. These Internetware-based platforms and tools have not only reduced the time and cost of developing distributed applications but also changed the overall enterprise application development process.

Amazon Web Services (AWS) [6] and Bizagi [7] are two examples from the industry that enable the modeling, development, and deployment of distributed applications and Business Processes. Could-based services of AWS can help organizations to automate and optimize their processes, reduce costs, and improve operational efficiency. One such service is AWS Step Functions, which is a fully-managed workflow service that enables businesses to coordinate the components of their applications and microservices using visual workflows. With Step Functions, businesses can build and run workflows that integrate with AWS services, including AWS Lambda, Amazon SNS, Amazon SQS, and more. This allows businesses to automate complex workflows, reduce manual intervention, and improve the speed and reliability of their processes. Similarly, the Bizagi platform is designed to help businesses automate and optimize their workflows and processes, with features like drag-and-drop process modeling, process analytics and reporting, process automation, and integration with other systems and applications. The platform is used by organizations in a wide range of industries, including financial services, healthcare, manufacturing, and government.

However, BP applications developed using a plug-and-play approach are not guaranteed to be fault-proof. BP designers may make errors due to a lack of semantic understanding of these web

Table 1.1: BP-specific fault categories and types

Fault Category	**Fault Type**	**Description**	**Equivalent Mutation Operator**
Variable assignment	Variable identifier replacement	Replaces a variable identifier by another of the same type, i.e, $servce2.var1 = service1.var1$ to $servce2.var1 = service1.var3$ or $service2.var1 = c$	ISV
Expression	Arithmetic operator replacement	Replace an arithmetic operator $(+, -, \times, /, mod)$ with another of the same type	EAA
	Unary operator removal	Removes unary $-$ or $+$ operator from an expression	EEU
	Relational operator replacement	Replaces a relational operator $(<, \leq, >, \geq, \neq, =)$ by another of the same type	ERR
	Logical operator replacement	Replaces a logical operator $(\wedge, \vee)$ by another of the same type	ELL
	Path operator replacement	Replaces a path operator $(/, //)$ by another of the same type	ECC
	Numeric constant modification	Modifies a numeric constant by incrementing/decrementing its value by 1 or by adding/removing one digit	ECN
Branching	Branch path removal	Deletes an $Elseif$ element from an If activity	AIE
	Join condition removal	Removes the $joinCondition$ attribute from an activity	AJC
Control flow	Activity removal	Removes an activity	AEL
	Activities order exchange	Exchanges the order of two $sequence$ child activities	ASI
	Sequential to parallel loop replacement	Replaces a sequential loop with a parallel one	AFP
	Sequence to flow replacement	Replace a $sequence$ activity by a $flow$ activity	ASF

services, or incorrect and/or incomplete workflow specifications. Such errors result in faults that may not necessarily be identified at the design or development stage and may only manifest during execution. Faults in a BP may also occur due to service implementation errors, service failure/unavailability, or network failure. This dissertation addresses the problem of detecting and resolving faults in BPs that result in incorrect or unexpected output due to design-time faults. Other faults including service implementation faults, network/service failure, and service unavailability are not considered in this work because they have already been addressed in prior work. For example, component service implementation errors can be addressed in BP development using unit testing-based approaches [8, 9]. Similarly, run-time faults in BPs due to service failure/unavailability, deployment issues, and unexpected network failure have been extensively studied in the literature for process adaptation [10, 11, 12] and delta debugging [13, 14]. However, no prior work addresses the problem of design-time faults that occur during the development of business processes.

BP design-time faults can be grouped into four broad categories listed in Table 1.1. This categorization and the underlying fault types within each category are based on the comprehensive set of mutation operators defined by Estero-Botaro et al. [15] for fault injection in BPEL processes. Therefore, any design-time fault within a BP can be represented as a combination of these mutation operators. Note that Estero-Botaro et al. [15] identified five different categories of mutation operators. Out of these five categories, we explicitly cover four while also implicitly covering the last (which relates to exception and event mutation), since faults related to exception and event mutation can be considered as special cases of control flow or branching faults.

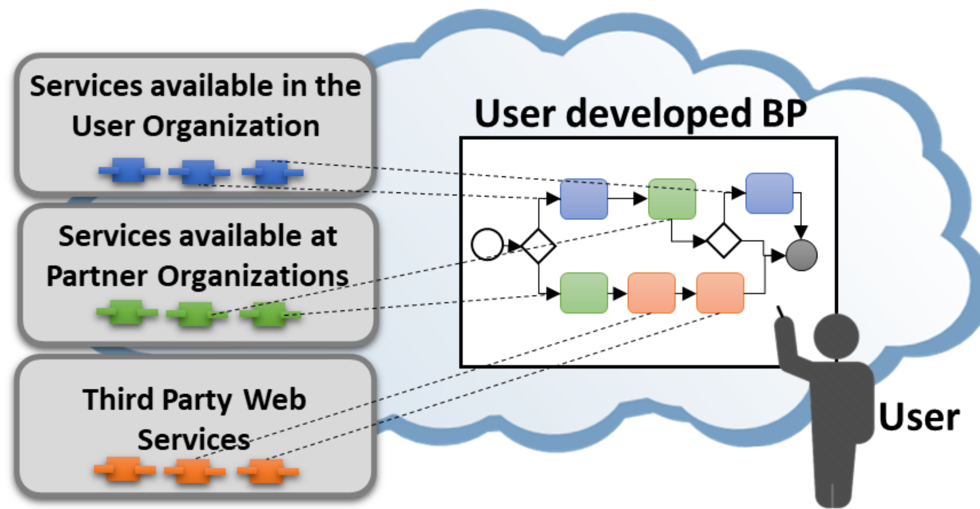


Figure 1.1: Services cloud environment for BP composition.

In this dissertation, we focus on the detection and resolution of design-time faults in BPs in the services cloud environment depicted in Fig. 1.1. The cloud environment hosts the services and BPs belonging to multiple organizations and provides a ground for collaborative BP development and fault resolution activities. Tools have been proposed [16] that support automatic or semi-automatic development of BPs by mapping the services of an organization to the BPs of other organizations available in the cloud. Such cloud-based BP development can induce faults and calls for a fault resolution framework within the services cloud.

1.3 Common Faults in Business Processes

In this section, we will discuss the faults that commonly occur during the development of Business Processes. Particularly, we present an example from each fault category listed in Table 1.1. Fig. 1.2 depicts a simple, fault-free, sales-order BP from the e-commerce domain in standard BPMN notation with annotations to show the path conditions of branches.

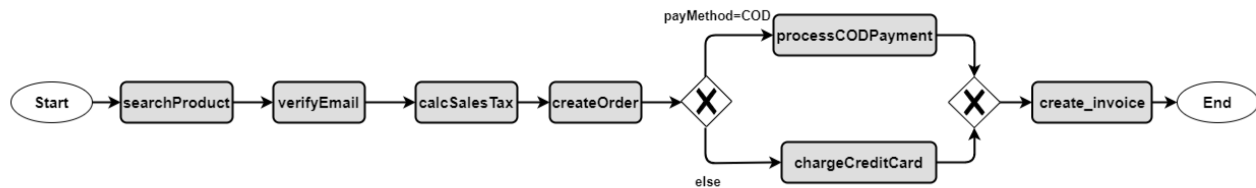


Figure 1.2: A sales-order BP from e-commerce domain

1.3.1 Expression Fault

Expression faults in BP development can occur when arithmetic or logical operators are incorrectly swapped with others from the same category. For example, swapping a $+$ with a $-$ or a $\times$ in an expression can result in an expression fault. Similarly, replacing an $=$ with a $\neq$ or $\geq$ also causes an expression fault. In Fig. 1.3, such a fault is shown where the path condition of an exclusive service

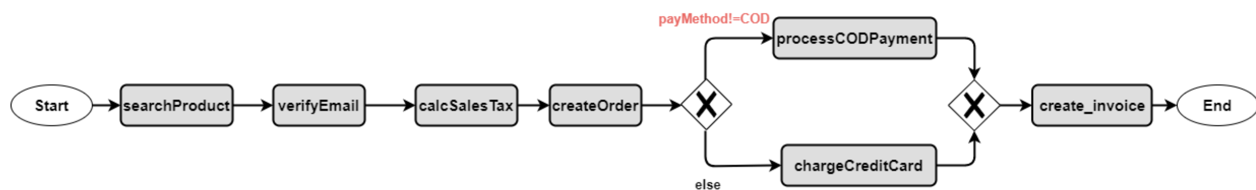


Figure 1.3: Expression fault introduced in Fig. 1.2

invocation was altered by changing the $=$ operator to $\neq$. This forces the BP to take an incorrect execution path and fail certain test cases.

1.3.2 Branching Fault

Branching faults in business process (BP) development can occur due to incorrect join conditions or the deletion of entire branch paths. An example of such a fault can be seen in Fig. 1.4, which shows a BP with a branching fault that was caused by the removal of a path from the BP in Fig. 1.2. As a result of this change, the BP is unable to process payments when the selected payment method is not cash-on-delivery.

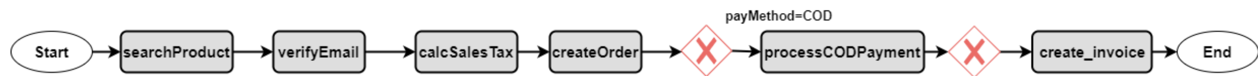


Figure 1.4: Branching fault introduced in Fig. 1.2

1.3.3 Control Flow Fault

The control flow faults originate when the order of sequential activities is exchanged, sequential activities are placed in parallel execution order or a loop with sequential dependency is configured for parallel execution. Fig. 1.5 manifests an example of a control flow fault introduced by

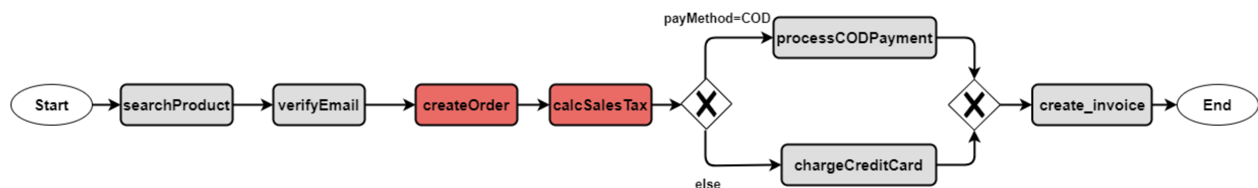


Figure 1.5: Control flow fault introduced in Fig. 1.2

exchanging the order of *createOrder* and *calcSalesTax* service operations. This introduces a data anti-dependence between both services, that is, *createOrder* service is data-dependent on a later service, *calcSalesTax*, in the execution path.

1.3.4 Variable Assignment Fault

This fault is a result of variable identifier replacement. For instance, if a correct variable assignment $a = b$ is replaced by $a = c$, this will cause a variable assignment fault. Fig. 1.6 shows the same BP

as in Fig. 1.2 but with elaborate detail of control flow and data flow between the various elements of the BP. The solid edges represent the control flow, dotted edges connect each service operation to its input and output parameters and dashed edges represent data flow or variable assignments. For instance, *searchProduct* service has one input parameter called *productName*. Input parameters are connected to the services with an arc from the parameter to the service. *searchProduct* has two output parameters namely *productId* and *taxClassId*. Output parameters are connected to the services with an arc from the service to the output parameter. Dashed edges represent the variable assignment or data flow of the BP.

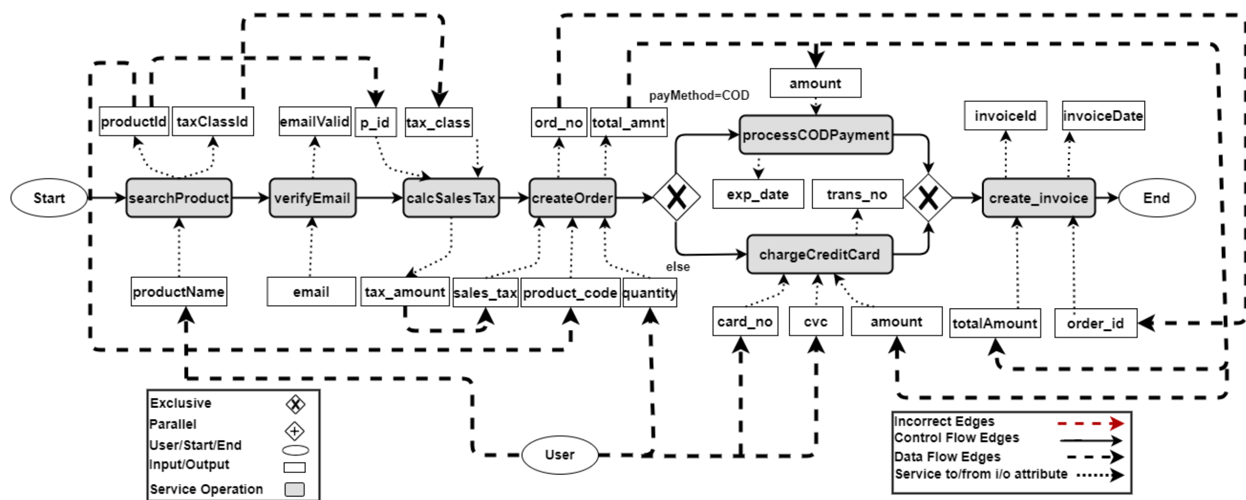


Figure 1.6: A simple sales-order BP from e-commerce domain with data flow

Fig. 1.7 shows the same BP as in Fig. 1.6 but with a replaced variable assignment shown with a red arrow. In this example, *productId* output parameter of *searchProduct* service is incorrectly assigned to *tax_class* input parameter of *calc_sales_tax* service and *taxClassId* output of *searchProduct* is mapped to *product_code* input of *createOrder*. The swapping of two identifiers results in the miscalculation of sales tax, leading to incorrect execution and inconsistent state of the BP.

1.4 Problem Statement

In this dissertation, we address the problem of automatic fault resolution in business processes. The objective is to develop a framework that can automatically locate and resolve faults in BPs in

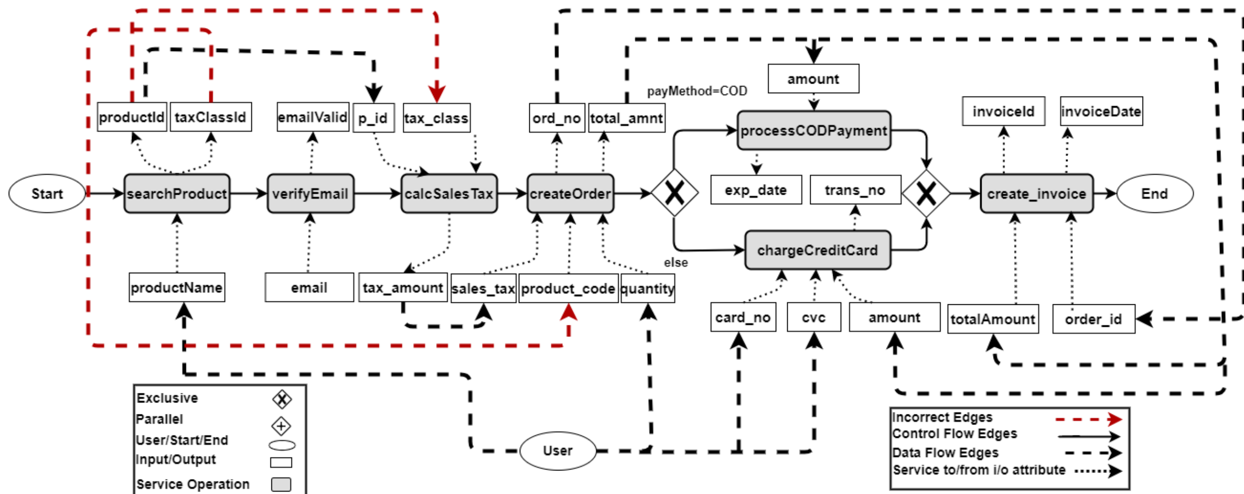


Figure 1.7: A simple sales-order BP from e-commerce domain depicting variable assignment fault

a seamless manner. Specifically, we address the following subproblems in this dissertation.

- Develop a fault resolution approach that given a faulty BP, automatically generates candidate fixes and applies those candidate fixes to resolve faults. The candidate fixes are generated by considering the possible mutations of the faulty BP while keeping the number of candidate fixes to a manageable level. Essentially, we are interested in building on the generate-and-validate methodology for fault resolution in BPs.

- Develop a collaborative fault resolution approach that given a faulty BP and a set of fault-free BPs, resolves faults in the faulty BP by exploiting the knowledge of given fault-free BPs.

- Develop a hybrid fault resolution approach that combines both generate-and-validate and collaborative approaches for fault resolution in BPs.

1.5 Methodology

Fig. 1.8 depicts a high-level view of the proposed methodology. The fault resolution process starts when the user provides a faulty BP, the test cases, and existing fault-free BPs (only for collaborative and hybrid fault resolution). After the input, candidate fixes are generated from the faulty BP and are validated against the set of provided test cases. If a candidate fix passes all the

test cases, it is presented to the BP designer. In order to avoid changing the scope and goal of the original BP, the suggested modifications and the resulting BP are reviewed by the BP designer who may selectively accept the modifications and/or make additional changes to the BP. Once the BP designer is satisfied with the changes, the resulting BP is deployed.

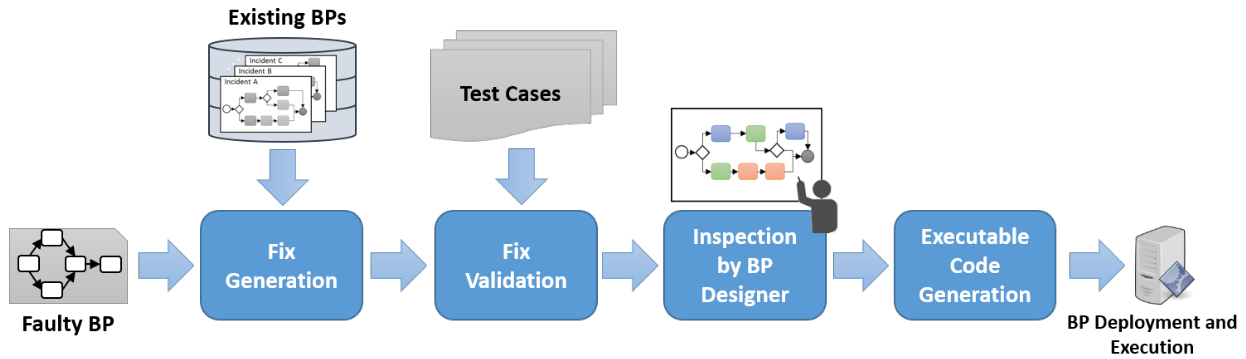


Figure 1.8: High-level architecture of the fault resolution methodology

1.6 Contributions

The main contributions of the work presented in this dissertation are summarized below.

1. *Efficient G&V (EGV) approach.* We propose an efficient fault resolution approach for BPs by extending the traditional G&V automated program repair methodology. While the proposed approach leverages mutation-based fault localization to achieve high localization accuracy, it significantly improves its efficiency by considering a relatively smaller subgraph of the BP that is obtained through statistical fault localization and predicate-based switching and slicing. Moreover, we boost the efficiency of fault resolution through static analysis and conditional generation of mutants. Note that G&V is widely used for automatic repair of Java and C programs [17, 18, 19, 20] but it has not been adapted for automatic resolution of faults in BPs encoded in BPMN or BPEL.

2. We formalize the problem of Collaborative Fault Resolution that aims to resolve the faults in a user-developed BP using information from existing fault-free BPs that use similar services

and that are known to be correct. We develop a heuristic approach based on association analysis over pair-wise transformations to identify likely transformation candidates, which are then iteratively selected so that the fault(s) in the BP are resolved while reducing the modifications to the original BP. Our proposed approach significantly improves on the existing automated program repair approaches since we make use of the knowledge of existing working BPs instead of just analyzing the faulty BP in isolation. This allows us to cover a broader range of BP fault categories and also expand the search space to find valid fixes.

3. We perform a comprehensive experimental evaluation over both synthetic and real data. The synthetic data is created by randomly injecting faults allowing comprehensive testing with all possible design time faults. The real data comes from a user study that asks real users to develop BPs as part of a class exercise, thus testing the effectiveness of the approach in resolving faults introduced by real users. We compare the proposed approach to a baseline generate-and-validate (G&V) automated program repair methodology. The results show that our approach can resolve a broader range of faults with high accuracy significantly outperforming the baseline.

4. *Hybrid Approach*. We also propose a hybrid approach combining the proposed EGV approach that performs fault resolution by analyzing a faulty BP in isolation with a collaborative fault resolution (CFR) approach for improved accuracy. Rather than examining the faulty BP in isolation, the hybrid approach enables broader coverage of fault types by utilizing the knowledge of existing BPs that are composed of similar services and are assumed to be correct.

5. We extend an existing framework for automated BP composition and management in a services cloud environment [16] by integrating automated fault resolution capabilities. In addition, we demonstrate the viability of this integrated framework by developing a prototype implementation that supports BP composition as well as the automatic resolution of faults.

1.7 Outline of Dissertation

The rest of the dissertation is organized as follows. Chapter 2 reviews the related work in the area of testing, fault localization, and automatic program repair. Chapter 3 presents the Efficient Generate and Validate (EGV) approach for fault resolution in BPs. Chapter 4 presents the Collaborative Fault Resolution (CFR) that compares the faulty BP with existing fault-free BPs for the resolution of faults. It also presents a hybrid approach which is a combination of EGV and CFR. Chapter 5 discusses the implementation of the prototype tool that is developed for fault resolution. Finally, chapter 6 concludes the dissertation and also discusses the future research directions.

Chapter 2

Related Work

This chapter briefly outlines the approaches and tools that either perform fault resolution or that are helpful in removing faults from a program. Fault resolution problem has been extensively studied for programs written in C/C++ and Java but this field has gained little attention in the SOA context. Generally, a fault-repair or fault-resolution approach has three components i) fault discovery, ii) fix generation, and iii) validation of the generated fixes. Fault localization techniques are used to identify the approximate vicinity of the faults. Fault localization is an integral part of most fault-repair techniques and its accuracy impacts the effectiveness of underlying fault-repair schemes. Fix generation is the process of generating candidate fixes by employing various techniques. Finally, the validation of candidate fixes is generally performed by running the available test suite. There are several techniques that can automatically generate the test suite from the input program. We will discuss the related work in three categories namely, fault localization, fault resolution, and service composition testing.

2.1 Fault Localization

Fault localization techniques aim to identify positions in a faulty program (a list of statements, branches, or blocks) that are likely to be responsible for the fault. The goal is to help programmers in debugging/ patching by focusing on the identified faulty statement(s) and to support automated

program repair and recovery [21, 18]. The effectiveness of any fault localization technique is measured by the accuracy with which it locates the faults. Broadly fault localization techniques can be categorized as being static or dynamic. Static techniques [22, 23, 24, 25] attempt to locate the faults by navigating through source code. Static techniques take a backward slice from incorrect outputs or they involve using a well-defined program model for source code checking. Dynamic techniques [26, 27, 28, 29], on the other hand, attempt to discover faults by contrasting the runtime behavior of successful and unsuccessful executions of the programs. The main benefit of dynamic techniques is that they do not require knowledge of program semantics. Instead, they only require labeled data for behavioral modeling of the faults. In the following text, we discuss representative work from both static and dynamic fault localization.

2.1.1 Static Techniques for Fault Localization

Static slicing: Static slicing [30, 31] is among the earliest works on fault localization. The idea of static slicing is to run the test cases and take a backward slice from the incorrect output(s) of the failed test case with the goal to reduce the search space to find bugs. Static slice includes all the statements that can potentially write to the incorrect output. Since static slicing does not use any execution information, so, it cannot determine the exact statements that actually changed the value of the incorrect output variable.

Static slicing has been found to be useful in locating the fault but it results in larger slices to examine for the faults [21]. Slice size can be reduced by either incorporating the run-time information with slicing (dynamic slicing) or by computing a dice. Dynamic slicing is discussed in Section 2.1.2. Dice is simply a set difference of program elements belonging to different slices [32]. Another challenge with static slicing is resolving the memory equivalence between different variables. Such equivalence is common in programs that use pointer variables or in situations where parameters are passed by reference or passed by pointer. Resolving memory equivalence is necessary because static slicing mainly depends upon data flow analysis which leads to the incorrect value of a variable for a particular test case(s). Liang et al. [33] proposed an approach for resolving memory equivalences for meaningful data-flow analysis in pointer-based programs.

Model-based techniques: Model-based techniques [22, 23] are among the static fault localization techniques that require a well-defined program model for effective fault localization. The availability and correctness of the model are critical to the functioning of model-based fault localization. In fact, the variations of the program from its model are used to detect faults. Techniques have been proposed [34, 35, 36, 37, 38] to learn the model directly from the faulty program. The learned model is, then, used to locate faults by comparing the behavioral difference between the model and the faulty program. Wotawa et al. [38] perform program analysis to build a dependency model from the source program. The model along with test cases and expected outputs are represented in first-order predicate logic. Failure in the source program is seen as a violation of the model and resulting conflicts between the source program and model are used to identify program components responsible for the fault.

A similar technique is presented in [34] that builds models for Java programs. It covers some features of Java language including methods, classes, conditional statements, and while loops. A dependency-based model describes the structure of the program whereas the behavior is described with logic-based languages such as first-order predicate logic. Then, the dependency-based model can be extended to handle unstructured branches like exceptions, recursion, and jump statements. Baah et al. [36] use a probabilistic dependency model to explain the behavior of program elements thus enabling a probabilistic analysis of faulty program elements. One limitation of model-based techniques is that they cannot be easily decoupled from the programming language of the subject program because it has to support the language features partially or completely.

2.1.2 Dynamic Fault Localization

Dynamic fault localization techniques have an advantage over static techniques in a way that they only require a test suite and program traces labeled as successful or failed. Fault localization is performed by comparing and contrasting the successful and failed traces using a variety of different methods. Below, we discuss some key approaches for dynamic fault resolution.

Dynamic Slicing: Unlike static slicing, discussed in Section 2.1.1, dynamic slicing [39, 40, 41, 42, 43, 44] incorporates the run time information to include only those statements in the slice/chop

that actually affect its value during execution. Wotawa et al. [39] combined dynamic slicing with model-based analysis for fault localization. Hitting sets are computed that comprise of at least one statement from each incorrect variable's slice. The probability or suspiciousness is calculated based on the number of hitting sets containing a statement. Zhang et al. [45] use the intersection of backward and forward slices to find the faulty program elements.

One limitation of dynamic slicing is that it cannot capture the execution omission errors. One solution is to use relevant slicing [46] that combines dynamic slicing with dependency analysis to localize faults. Relevant slicing, however, tends to include unrelated statements in the slice in the presence of incorrect dependencies between the statements.

Spectrum-based Techniques: Spectrum-based techniques make use of test case coverage information to associate a suspicion value to program entities computed based on statistics of passed and failed test case runs. The programmer is then expected to examine the statements ranked in order of suspicion score to locate faults. Set union, set intersection, nearest neighbor queries [47] and Tarantula [48] are representatives of spectrum-based fault localization.

Set union, set intersection and nearest neighbors consider a faulty program with one failed run and multiple successful runs. Set union reports the statements to be fault relevant if they appear in the failed spectrum but do not appear in any of the successful spectra. Set intersection, on the other hand, considers those statements that appear in the failed spectra but do not appear in the intersection of successful spectra. The nearest-neighbor approach first finds a successful spectrum closest to the failed spectrum according to some distance measure. Then, the statements in the difference set of the closest passed spectrum and failed spectrum are reported as fault relevant. In all three cases, Renieris and Reiss [47] produce a fault report that is a collection of suspicious statements for the debugger to examine. Interestingly, the individual statements in the fault report are not arranged in any order but a usefulness score is assigned to the entire fault report with 1 representing the perfect report and 0 representing the worst report.

On the contrary, Tarantula [48] is a statement-based technique that assigns a suspiciousness score to each statement based on its hit spectra both in successful and failed executions. The statements are, then, investigated by the debugger in order of suspiciousness score from high to low.

The difference between Tarantula and nearest neighbors is that Tarantula uses multiple successful and failed spectra to assign a score to each statement whereas nearest neighbors only consider one failed spectrum and one successful spectrum that is closest to the faulty spectrum. A study on the Siemens suite [49] shows that Tarantula outperforms nearest-neighbors in a number of statements that must be inspected before the fault is uncovered. Later on, different extensions and variations of Tarantula have been proposed. For instance, Derboy et al. [50] grouped the statements that appear in an equal number of failed runs, and the statements within each group are ordered according to their suspiciousness score. Tarantula has been used in conjunction with different scoring functions like *Ochiai* [51] and *ochiai2* [52] similarity coefficients to achieve better results. An empirical study [53] evaluates the proposed Dstar technique with 31 spectrum-based techniques that use similarity coefficients to rank the program elements. It is noted that Dstar, in most cases, outperforms all such techniques including Tarantula [48] and Crosstab [54].

Statistical Techniques: Statistical fault localization works by instrumenting the source program with special statements called predicates. Predicates are evaluated at runtime and their result can either be true or false. Mostly, predicates are associated with branches, function return values, and relational operators between scalar variables. Statistical formulas are used to rank the predicates based on their evaluations of successful and failed executions of the source program. Two representative techniques in this category are Libit05 [27] and SOBER [28, 29].

Libit05 [27] computes two conditional probabilities for each predicate P, that are, ***Failure(P)*** and ***Context(P)***. ***Failure(P)*** is the probability of a run being failed given that P evaluates to $true$ and ***Context(P)*** is the probability of failure given that P is executed. The difference, $Context(P) - Failure(P)$, is then used as a discriminant. Predicates for which the difference is less than or equal to zero are excluded from the list of suspicious predicates. The remaining predicates, with a positive difference, are ranked according to a statistical formula.

Unlike LIBIT05, SOBER [28, 29] considers the multiple evaluations of each predicate in each run. It computes the probability $\pi(P) = \frac{n(t)}{n(t)+n(f)}$ where $n(t)$ is the number of times a predicate evaluates to $true$ and $n(f)$ is a number of time it evaluates to $false$ in a particular execution of the source program. The idea of SOBER is that, if a predicate is predictive of a fault then its

probability distribution must diverge significantly across successful and failed executions of the source program. The divergence in a probability distribution is used to rank the fault relevance of each predicate.

SOBER proved to be more effective on Siemens software suite [28] than Libit05 [27] because it considers a broader spectrum of predicates rather than analyzing its single evaluation. Both for SOBER and Libit05, statements related to top k predicates are taken as initial fault report to be examined by the debugger. Libit05 does not provide any mechanism for ranking all the statements especially if they lie outside the scope of a predicate. For SOBER, if a faulty statement is not included in the initial fault report, the program dependency graph is traversed using breadth-first search unless the fault is found or the entire graph is traversed.

Naish et al. [55] conducted a study to evaluate the effectiveness of spectrum-based and predicate-based approaches and concluded that, in general, predicate-based techniques perform better than spectrum-based fault localization.

Memory-based Techniques: Every program has a state or context during execution at any particular point in execution. The context of a program is represented by its memory contents or memory graph. Memory-based techniques exploit the difference between memory graphs of successful and failed runs of a program to localize faults. Zeller et al. proposed delta debugging [56, 57] that replaces the state of a successful run by corresponding variables in the corresponding state of a failed run to reproduce the fault. A variable is not marked as suspicious if the change does not produce the exact same fault observed by the failed run. The extension in delta debugging [26] uses causal analysis to discover the time and location when the root cause of a failure transitions from one variable to another.

One limitation of delta debugging is its cost. There can be many states in the execution of a program and it is infeasible to track all the states and incorporate failure-inducing deltas in them. Gupta et al. [58] used delta debugging to, first, identify failure-inducing inputs. Then, a program chop at the intersection of an output's backward slice and failure-inducing input's forward slice is reported as suspicious.

Another class of memory-based techniques is predicate switching [45, 59] that forces a program

on a different execution path. Predicates are switched either according to their priority or on the last executed first switched strategy. Zhang et al. [45] switch the predicates to observe their effect on program output. if a change in predicate causes the failed run to execute successfully then the predicate is marked as fault-relevant. Li et al. [59] combine predicate switching with dependency analysis to reduce the size of the fault report.

2.1.3 Fault Localization in SOA Context

Fault localization is more challenging in service-oriented architecture (SOA) than in monoliths due to its distributed and asynchronous nature. Many component services interact in a predefined fashion to make a composition that achieves pre-defined business objectives. The component services can be developed in-house or can be owned by a third party and they can be geographically dispersed in different locations. The services interact through the network to contribute to the objective of the composition. More often than not, the source code and tracing/logging information of the component services is not available. This, along with the challenges of connecting networks and asynchrony, makes the debugging and localization of faults harder than the monoliths.

However, most of the fault localization techniques [14, 60, 61, 62, 63, 64] in SOA context are adapted from the ones proposed for the monoliths. BPELDebugger [60] is one such approach that brings many spectrum-based techniques to the SOA context. It considers the compositions (implemented in BPEL) instead of component services. The adapted techniques include set union, set intersection, and nearest-neighbors [47] and Tarantula [48]. After evaluating different formulas on the elements of composition, BPELDebugger uses two BPEL-specific guidelines to rank the suspicious program elements.

Sun et al. proposed BPELswice [61] that is designed specifically for fault localization in BPEL programs. BPELswice employs predicate switching and backward program slicing to locate the suspicious faulty code with higher precision. Unlike other fault localization approaches that return the blocks in BPEL code with possible faulty statements, BPELswice performs program slicing to reduce the number of statements within the suspicious blocks to help the BP designer in debugging.

Delta debugging [13] is another frequently employed approach for fault localization that can

identify deployment and configuration-related faults in addition to general programming errors. In the context of microservices-based systems, Zhou et al. [14] employed delta debugging to uncover faults that occur in deployment, environmental configuration, and execution sequences of microservices. Some recent approaches combine mutation testing [65] and delta-debugging for accurate fault localization in a more efficient manner. Delta debugging [14] is a state-based technique and its major limitation is its computation time due to the existence of potentially many program states. Zhou et al. [63] proposed the parallel version of delta debugging for microservices to mitigate this overhead.

Some recent studies [62, 66, 67] apply fault localization techniques to address the problem of root cause localization when performance degradation is observed in microservice systems. The approaches [66, 67] work by creating a dependency graph between services and by analyzing the graph for causal inference based on conditional independence. Building of dependency model is time-consuming, especially in the context of SOA where a microservice system evolves over time. Ye et al. [62] proposed T-Rank to rank the services in order of suspiciousness. They use spectrum-based fault localization by considering the latency of component services to complete a request. The latency information is captured from system traces in a sliding-window fashion and updated continuously. This makes T-Rank a lightweight and readily applicable approach for microservices.

Recently, Mathur [64] proposed a replay-based debugging framework. Messages routed to the microservice system are captured in a language-agnostic fashion which are, then, used to reproduce the anomaly in debug environment running in a different container. The anomaly detector of the framework captures the latency of component services and resource (CPU, memory, etc.) usage of the container as time series data. This data is used to track anomalies back to traces so that the traces can be replayed in debug container to reproduce the anomalies.

2.1.4 Recent Trends in Fault Localization

In recent years, the focus of research on fault localization has shifted towards machine learning [68, 69], the combination of different techniques [70, 69, 71], and methods for improving existing techniques [72, 73] by the extra provision of data or by introducing new parameters for existing

ranking formulas. For instance, MEPFL [68] is a machine-learning technique for fault localization and latent error prediction for microservices. Faults are injected into the microservice system and trace logs from the original program and injected version are used for training the model. UniVal [69] is a combination of predicate-based statistical and variable-based fault localization that employs causal analysis and machine learning. The predicate evaluations are converted into variable values so that a unified approach can be used for causal analysis.

Jia et al. proposed SMFL [70] which is a combination of spectrum-based and mutation-based fault localization. spectrum-based techniques suffer from assigning the same rank to multiple statements and mutation-based techniques are notoriously time-consuming. SMFL addresses the limitations of both approaches. It, first, computes the suspiciousness scores using spectrum-based techniques, and then, top n suspicious statements are chosen to generate the mutants. Finally, the mutant that changes the outcome of failed test cases more frequently and of the passed test cases less frequently is regarded as the most suspicious. To substantiate the study, two spectrum-based approaches Ochiai and Dstar[74, 53] and two mutation-based approaches Metallaxis and MUSE [75, 76] were selected and all four combinations were studied. For instance, the DStar-MUSE combination performs spectrum-based ranking using DStar, and mutants on top n statements are generated using MUSE.

IsoVar [71] combines statistical and mutation-based fault localization to identify suspicious variables. First, statistical analysis is performed to capture the execution metrics of variables to identify the set of suspicious variables. Subtle mutants are, then, generated for these variables on the byte-code level to observe their impact on program outcome. Those variables are regarded as fault-relevant that affect the outcome of failed test cases more than the outcome of passed test cases.

Zhang et al. present page rank fault localization (PRFL) [72] to boost the effectiveness of existing spectrum-based techniques. The key insight of PRFL is that if a failing test case t_1 covers fewer statements than another failing test case t_2, then, t_1 makes more contribution than t_2 for the fault localization cause. Thus, PRFL first ranks the test cases using the page rank algorithm reflecting their contribution. Then, the program spectrum is recomputed based on ranked test cases.

Finally, a variety of spectrum-based formulas are applied to the recomputed spectrum for ranking the statements in order of suspiciousness. A similar study is proposed in [73] that calculates an importance weight for each program element (statement, method, etc.) and multiplies it with the score achieved from the spectrum-based formula. Importance weight is simply the ratio of failed test cases containing the element.

2.1.5 Summary

We have comprehensively covered the related work in fault localization both for monoliths and in SOA context. Fault localization is a key component of any fault resolution approach because the faults that are not localized effectively are unlikely to be fixed. Liu et al. [77, 78] observe that automatic program systems are sensitive to the fault localization noise and their effectiveness highly correlates with the effectiveness of the underlying fault localization strategy. This calls for the selection of the best fault localization technique at the forefront of our fault resolution framework. But, there is no single best technique that outperforms others on all or most of the benchmarks. Some techniques work better than others on one benchmark and are less effective on other benchmarks. However, it is clear that static techniques cannot perform better than dynamic techniques because of their inability to incorporate execution information.

Among the dynamic techniques, machine-learning-based approaches are not feasible in our context due to the unavailability of training data. Moreover, we need a fault localization approach that assigns suspiciousness scores to program elements or predicates so that they can be ranked accordingly before the generation of candidate fixes. Both spectrum-based and statistical techniques perform this task. However, Wong et al. [21] note that statistical techniques can achieve more effectiveness than spectrum-based techniques. Consequently, In our work, we make use of statistical fault localization [28] in both Efficient Generate-and-Validate (EGV) (discussed in Chapter 3) and Collaborative Fault Resolution (CFR) (discussed in chapter 4). For EGV, we also use predicate switching and program slicing [61] to further limit the number of suspicious elements.

2.2 Fault Resolution

Automatic program repair (APR) involves resolving faults in a program without human intervention. A failure is a behavior of a program that is different from the expected behavior. An error is a state that leads to a failure, and the fault is the root cause of the error. APR systems contain mechanisms for differentiating between acceptable and unacceptable behavior, locating faults, and generating fixes or patches. The behavior of a program is often described with the help of *oracles*. An *oracle* is a term that can refer to a formal model, program specifications, or test suite. These all define the expected behavior of a program, and any behavior that deviates from the expected behavior is deemed unacceptable. An oracle can be divided into a *fault oracle* and a *regression oracle*, where the former uncovers the fault, and the latter is used to verify that no new faults have appeared during the fault resolution process.

Once unacceptable behavior has been identified in a program, the next step is to locate the fault in order to find a solution. Fault localization is therefore critical to the success of any fault resolution strategy. If a fault is not localized, it is unlikely to be repaired. Therefore, the effectiveness of a fault resolution technique is heavily dependent on the fault localization scheme used. Section 2.1 discusses key fault localization techniques from the literature. After faults have been discovered, the final step is to generate candidate fixes or patches that pass the fault oracle and do not violate any of the regression oracles. Candidate generation often involves applying mutation operators to the program, using predefined fault templates, or using machine/deep learning techniques. There has been significant work on the automated repair of programs written in Java and C/C++, but none of this work has addressed program repair in Business Process (BP) programs developed through web service orchestration. In the following text, we discuss some key approaches to program repair from the literature categorized by their patch generation strategy.

2.2.1 Mutation-based Fault Resolution

Genprog [79, 80, 81] is a genetic-programming-based repair system. It converts the faulty programs into an Abstract Syntax Tree (AST) representation and uses three repair operators that mutate ASTs: deletion, addition, and replacement. For addition and replacement, the nodes used are

taken from elsewhere in the code base. The selection of nodes is based on a technique known as the redundancy assumption. Genprog has been successfully applied to large-scale C code and has been shown to fix 55 out of 105 bugs. A similar study [82, 83] defines seven mutation operators for AST representations of faulty programs. A prototype called *Jaff* was also developed to handle a subset of Java constructs and was evaluated on small programs.

Some studies [84, 85, 86] apply mutation operators directly on the faulty programs rather than its intermediate representation like AST. Common mutation operators include replacing a relational or arithmetic operator with another of the same type, negating a boolean expression, etc. The faults are located using spectrum-based techniques. Nica et al. [85] explore mutation space more comprehensively than [84]. Kern and Esparaza [86], on the other hand, generate a meta-program that includes all the possible mutants according to a mutation operator. Only the selected mutants from the meta-program are executed based on the values of meta-variables which are evaluated using symbolic execution.

SemFix [87] is a program repair tool that uses angelic debugging [88] for fault localization. It considers the faulty programs that can be fixed by modifying one expression. Angelic debugging finds plausible candidates which are, in fact, the mutants of the faulty program. The search space of mutants is reduced by using symbolic execution. Finally, SemFix patches the candidates produced by angelic debugging through code synthesis. One limitation of SemFix is that its symbolic execution phase is quite expansive and cannot scale for large programs. To address this limitation, Angelix [89] significantly optimizes the symbolic execution for large programs.

NOPOL [90] only focuses on repairing the faults in existing conditional statements and patching missing conditionals. It examines the spectrum of a failed test case to find suitable candidates for repair. Then, instrumented traces from failing and passing test cases are transformed into Satisfiability Modulo Theory (SMT) problem. The result of SMT problem is then converted to generate a patch for an existing or a missing conditional statement. NOPOL is also used in fixing the infinite loop problem [91] as they are caused by an incorrect or missing conditional statement.

One widely used methodology for automated program repair is *Generate-and-Validate* (G&V) [92, 93, 94, 95, 18, 96, 97]. G&V takes as input a faulty program and a group of passing and

failing tests and heuristically searches the program space to generate candidate fixes. The validity of a candidate fix is then checked by running all available tests. Xu et al. have proposed an efficient G&V approach for repairing Java programs [18]. Their approach employs fault localization to identify a list of suspicious snapshots, including program states that are indicative of faults. For each suspicious snapshot, a number of candidate fixes are generated by considering different program mutations. However, instead of validating each candidate fix for fault resolution, only selected candidate fixes based on their suspicious score are validated. It is possible that none of the selected candidate fixes pass all the test cases. The candidate fixes that pass some of the test cases are used to generate variants of the original program and the entire process of fault localization, fix generation, and validation (called retrospective fault localization) is repeated on the program variant. This retrospective fault localization continues until valid fixes are found. Retrospective fault localization provides an efficient search of the fix space by reusing the outcome of failed fix validation to support mutation-based dynamic analysis without exhaustively validating all candidate fixes.

2.2.2 Pattern-based Fault Resolution

Pattern-based automatic repair (PAR) [98] employs a template-based patch generation for Java programs. PAR devises 10 patches after a careful manual inspection of $60,000$ patches written by developers. Templates aim to target the most common programming faults. For instance, there is one template that targets the null pointer exception. The template is parameterized by variable name and places a null check on the variable before it is actually accessed. PAR is empirically evaluated and reported to outperform GenProg [79].

Relifix [99] takes a template-based approach for patch generation and it considers the regression bugs, that is, the faults arising from the code changes. Relifix comes with eight repair templates manually derived from 73 real code regressions. The templates leverage the changes in previous commits of the source versioning system to suggest repairs. The goal is to keep as much functionality with the latest version as possible while reconciling code changes among different commits. The functionality of the latest version is specified by regression tests.

Logozzo and Ball [100] propose a repair approach based on static analysis of .NET code. They consider programs off by one fault. Faults are categorized into different classes and a repair operator is chosen on the basis of the fault class. Some common fault operators include placement of precondition, changing the array size, etc. The result of fault operators is also verified by static analysis. Logozzo and Martel [101] also used static analysis to repair integer arithmetic faults. For instance, re-ordering of arithmetic operations is performed to ensure that arithmetic overflow does not occur.

Gao et al. [102] employs static analysis to discover and fix memory leaks in C programs. The suggested repair is the inclusion of a deallocation statement at the appropriate place in the faulty program. DeepFix [103] combines static analysis with deep learning to repair common compile time errors of C programs. Muntean et al. [104] incorporate static analysis for discovering buffer overflow which is, then, repaired using parameterized templates.

Liu et al. [77] present an extensive assessment of fix patterns for APR, including an investigation of their diversity, repair performance, and sensitivity to fault localization noise. The authors implement an APR system called TBar that uses a curated set of fix patterns and evaluate it on the Defects4J benchmark. They find that TBar achieves a new record level of repair performance, with 74/101 bugs fixed correctly and 43/81 bugs fixed plausibly with realistic fault localization output. They also find that most bugs are fixed by a single fix pattern, but some patterns are more effective at repairing certain types of bugs. Finally, they find that fix patterns are sensitive to fault localization noise, with certain patterns being more robust than others. Liu et al., in another article [78], argue that the performance of APR systems can be impacted by the accuracy of the FL step, which can either boost or degrade the performance of the APR system. Therefore, the number of bugs fixed is a misleading metric for reporting the effectiveness of program repair systems because unlocalized faults are unlikely to be fixed.

FixMiner [105] focuses solely on the patch generation process rather than considering automated program repair as a whole. The fix patterns are mined from thousands of atomic changes and patches from existing open-source repositories. The code is represented using *Rich Edit Script* at three levels: abstract syntax tree, edit action trees, and code context trees. All three represen-

tations are used in different phases of clustering to group similar and actionable changes that can be readily applied and used by any patch generation system. Quantitatively, FixMiner is evaluated by integrating it to PAR [98], and the resulting automatic repair system is called $PAR_{FixMiner}$. $PAR_{FixMiner}$ showed promising effectiveness on Defects4J benchmark system by producing 81% correct patches.

2.2.3 Machine-learning-based Fault Resolution

Chen et al. [19] propose using sequence-to-sequence learning, a branch of statistical machine learning, for APR in a language-agnostic, generic manner. They propose a program repair approach called SEQUENCER that uses sequence-to-sequence learning to repair real bugs in Java programs. SEQUENCER is trained on a dataset of one-line commits and uses a specific network architecture to address the challenges of using sequence-to-sequence learning on code, such as handling dependencies and strict language rules. The authors evaluate SEQUENCER on a number of benchmark programs and show that it is able to repair a wide range of bugs with a high degree of reliability. Overall, the authors argue that using sequence-to-sequence learning for APR has the potential to provide a foundation for connecting program repair and machine learning, allowing the program repair community to benefit from training with more complete bug datasets and continued improvements to machine learning algorithms and libraries. They also suggest that this approach could lead to the development of more advanced APR techniques that are able to handle larger and more complex repairs.

CoCoNuT [106] addresses the limitation of existing G&V techniques that have limited search spaces and require extensive customization to work across programming languages. Neural Machine Translation (NMT) is a deep learning approach that uses neural networks to generate likely sequences of tokens given an input sequence and has been applied to natural language translation tasks. NMT has the potential to be used in APR as a way to translate buggy code into correct code, but there are challenges to applying it to this task, such as representing context and handling large search spaces. CoCoNuT proposes a context-aware NMT architecture with separate encoders for buggy lines and context and a new decoding strategy that uses a sliding window to handle large

search spaces and improve the efficiency of the model. Convolutional Neural Network (CNN) is used to capture the hierarchical features from the code and ensemble learning is used to combine multiple trained models with tuned hyper-parameters. The proposed approach is evaluated on a dataset of real-world Java bugs and is shown to generate more correct patches than a baseline G&V technique.

DLFix [107] addresses the limitations of NMT-based approaches for program repair. Specifically, NMT-based techniques are unable to encode knowledge about what parts of the buggy code have changed. Additionally, the use of a sequence-based representation for source code is not suitable for capturing code structures, and it lacks consideration for the context surrounding the fixing locations. To address these limitations, DLFix proposes a graph-based representation for source code and a graph neural network (GNN)-based approach to APR. The proposed network consists of a tree-based RNN layer that captures the code context and its dependencies. The weighted output of this layer is subsequently used to produce bug-fixing transformations. DLFix is evaluated on a dataset of real-world Java bugs and is shown to outperform NMT-based approaches and most of the template-based approaches in terms of the number of correct patches generated and the quality of the patches.

CURE [108] is another NMT-based approach for program repair. Prior to learning the translation task, it trains a Programming Language (PL) model on a fairly large code base to learn the programmer-like coding style. This process helps understand the strict syntax of source code, a property that is not observed in human languages and, thus, is alien to neural machine translation. Based upon the PL model, CURE employs a code-aware strategy and subword tokenization for patch generation resulting in a significantly smaller search space than contemporary NMT-based repair systems. CURE is evaluated on Defects4J and QuixBugs and has outperformed all the existing fault repair approaches.

Namavar et al. [109] assess the effect of different code representations in Deep Learning (DL) based program repair systems. Essentially, the learning process requires the source code to be transformed into vectors. Prior to this transformation, the code can be expressed with a variety of representations including AST and tokens. Different DL-based approaches use different repre-

sentations and, thus, their effect was unknown. Namavar et al. [109] fill this gap by conducting a study that switches the code representations of existing DL-based techniques and reports its effect in general and with respect to fault categories. The study concludes that no single representation is best for all categories.

CACHE [110] is a deep learning-based approach for assessing the correctness of patches. CACHE is context-aware, that is, it takes into account not only the changed statements in a patch but also the other correlated statements within the same method. CACHE is also aware of the program structure, using Abstract Syntax Trees (ASTs) to represent the code for embeddings. CACHE was evaluated on two patch datasets, one containing 1,183 patches and the other containing 50,794 patches. The results showed that CACHE outperformed previous representation learning techniques and other patch correctness assessment techniques in terms of F1 score and precision. An ablation study also demonstrated that the context information and program structures significantly contribute to CACHE's performance. It is claimed to be the first approach that incorporates context information in the embedding of code changes.

Tian et al. [111] propose a novel heuristic that suggests a relationship between patches and their failing test cases, and use this heuristic to develop an approach called BATS (Behavior Against Failing Test Specification) for predicting patch correctness. BATS works by statically checking the similarity of generated patches against past correct patches that correspond to failing test cases that are similar to the failing tests of the bug being fixed. The authors validate the heuristic by performing hierarchical clustering based on the embeddings of test cases and patches in the Defects4J dataset and then evaluate BATS by applying it to a large dataset of patches generated by 32 APR (Automatic Program Repair) tools or extracted from defects benchmarks. The results show that BATS has an AUC (Area Under Curve) of approximately 0.56 to 0.72 and a recall of approximately 56% to 84% in identifying correct patches. BATS is also found to be complementary to other approaches, such as a recently supervised learning classifier and PATCH-SIM [112], in improving the recall for detecting correct patches.

Yang et al. [113] discuss the importance of reliable evaluation in automated program repair (APR) and propose a systematic approach for understanding and discovering biases in APR eval-

uation. The authors conduct a systematic literature review to identify known biases in APR evaluation and build a taxonomy to categorize these biases. As a result, they identify 17 known biases and uncover a new bias related to the usage of patch validation strategies. To validate this new bias, they develop an executable framework called APRConfig and use it to evaluate three patch validation strategies with four APR techniques on three bug datasets. The authors' systematic exploration and the APRConfig framework provide insights and infrastructure for understanding and mitigating biases in APR evaluation.

SeAPR (Self-Boosted Automated Program Repair) [114] is a technique for leveraging patch execution information during automated program repair (APR) to directly boost existing APR techniques on the fly. SeAPR promotes the ranking of patches that are similar to high-quality patches that have been executed and degrades the ranking of patches that are similar to low-quality patches. The patch similarities are based on the modified elements in the patches. The authors evaluate SeAPR on 13 state-of-the-art APR systems and investigate the impact of various configurations on its performance. They also evaluate SeAPR's performance using historical patch execution information from other APR tools. The results show that SeAPR can substantially speed up APR techniques by up to 79% with minimal overhead, has stable performance with different patch priority formulas and patch execution matrices, and can effectively use historical patch execution information from other APR tools.

2.2.4 Summary

Fault resolution is a complex task that involves fault identification (localization) followed by the generation of its fixes and then, the verification of generated fixes. We have comprehensively covered the state-of-the-art fault resolution (aka automatic program repair or APR) approaches for monoliths from the literature. However, no fault resolution system has been proposed that fixes design time composition faults in the Business Processes. The proposed fault resolution framework is a unique and novel endeavor in this regard.

Fault resolution systems generally vary in the way they generate candidate fixes for the incorrect program. In our framework, we propose two approaches; EGV (discussed in chapter 3)

and CFR (discussed in chapter 4). EGV is a mutation-based approach that builds upon the basic Generate-and-Validate (G&V) approach by considering the mutation operators specific to the BPs. CFR, on the other hand, is a data-mining-based approach that mines the fix patterns automatically from existing fault-free BPs.

2.3 Service Composition Testing

In this section, we discuss the existing work in the area of testing web service compositions. The work in this area can be broadly categorized into unit testing, integration testing, and regression testing. We discuss work related to each of them in separate subsections.

2.3.1 Unit Testing

Unit testing is a technique of testing software systems where a program is decomposed into smaller modules or functions. Each module or function is then tested independently of the rest of the program [115]. There has been much work on unit testing the individual web services [9, 116, 8, 117, 118] but most of this work assumes the presence of expected outputs that can be used to verify the results of generated test cases.

Bartolini et al. [8] address the testing of web services, which is a critical issue for the IT industry. The study recommends that at least 25% of the effort spent on an SOA project should be dedicated to testing, as web services must offer strict guarantees of reliability and security. There are several industrial testing tools available, such as soapUI, PushToTest, and SOATest. The research proposes a framework for the turn-key generation of web service test suites, which combines the coverage of web service operations with well-established strategies for data-driven test input generation. The prototype is obtained by integrating two existing software, soapUI and TAXI, and named WS-TAXI. The original idea of WS-TAXI is the inclusion of a systematic strategy for test generation based on basic principles of the testing discipline, such as equivalence partitioning, boundary analysis, and combinatorial testing.

In [9], Chang-ai et al. present a framework for testing individual web services in the absence

of oracles (expected correct outputs of a test case). The approach exploits metamorphic relations between services to generate metamorphic test cases and verify their correctness. This provides an effective mechanism for testing external web services without source code access. The framework comprises four components namely Test Case Generator (TCG), Executor, Evaluator, and Configuration. TCG parses the web service WSDL and is responsible for the generation of test cases given Metamorphic Relations (MRs). Test cases are generated randomly or fetched from the previous test executions. TCG can generate the test cases both in batch mode or one-by-one mode depending upon the option chosen in the Configuration component. The executor executes the test cases generated by TCG via SOAP messages and intercepts the output. The evaluator, on the other hand, compares the output intercepted by Executor and the output conforming to the MR. If the outputs do not match then the fault is perceived.

Hong et al. [119] proposed an approach for collaborative testing of web services. The framework consists of special T-services, General Testers, Test Brokers, a UDDI registry, and ontology management. T-services are basically mock services for actual services called F-services with additional functionality. For instance, T-services support the formal description of semantics, configuration, internal design, and the information of underlying software and hardware platforms. To support the internal behavior of an F-service, T-service can even consist of the actual code for the tests like code coverage. Furthermore, a T-service is equivalent to an F-service in functionality but different in its real-world impact. For instance, F-services that do not maintain an internal state can also act as T-services and there is no need for specific T-services for such F-services. General Testers are also the services but they are not related to any actual service. They, rather, are testing tools that can plan and generate test cases and check testing results. Test Brokers are the components that take testing requests from the user, decompose them into sub-activities and find the appropriate General Testers for carrying out those sub-tasks. The broker also monitors the performance of involved T-services for optimization and replacement. A registry is a place where all the service descriptions reside so that they can be discovered and invoked at run-time. This supports the testing on the fly when services are bound dynamically. Finally, the ontology manager provides the vocabulary for passing information among different parts of the framework.

Barros et al. [120] present a systematic approach for static analysis of API specifications in order to improve modularity and address the complexity of APIs. The approach is based on the concept of business entity and applies interface analysis methods and techniques to elicit knowledge of business entities and their attributes, derive the temporal order of calling operations across multiple business entities, and learn and extract various ways of invoking a service via APIs. It is implemented and applied to a group of widely-deployed services for validation. The research aims to identify key aspects of both the structure and behavior of APIs to help users understand complex interfaces and facilitate efficient and effective service integration.

RESTest [121] is an open-source automated testing framework for RESTful web APIs. RESTest utilizes a model-based approach, making it easy to integrate with different test case generators and testing frameworks. The framework's key feature is its support for the specification and automated analysis of inter-parameter dependencies using the IDL tool suite. This allows for constraint solving to be used as part of the test generation process, which is known as constraint-based testing. This approach enables better coverage of the program under test by systematically generating valid and invalid input combinations and using novel output assertions. RESTest was evaluated by testing 9 operations on 6 commercial APIs, including Tumblr, GitHub, and YouTube. The results showed that RESTest's constraint-based testing was able to generate 100% valid test cases in milliseconds, and detected more failures than random testing.

RESTTESTGEN [122] is another approach for automatically generating test cases for REST APIs. The approach is based on defining the interface for interacting with a REST API, which includes the list of available operations, and the format of input/output data for requests and responses. The authors propose the Operation Dependency Graph as a way to model data dependencies among operations, which can be inferred from the REST API interface and updated when generating test cases. RESTTESTGEN aims to test REST APIs from two perspectives: nominal execution scenarios, which test the system using input data as documented in the interface, and error execution scenarios, which exploit input data that violate the interface to expose implementation defects and unhandled exceptional flows. The approach was evaluated through an extensive study involving 87 real REST APIs and was able to reveal a significant number of implementation

defects.

ARTE [123] is an automated method for extracting realistic test data for web APIs from knowledge bases like DBpedia. The approach uses natural language processing, search-based, and knowledge extraction techniques to automatically search for realistic test inputs using the API parameters specification. ARTE has been integrated into RESTest, an open-source testing framework for RESTful APIs, which fully automates the test case generation process. Evaluation results on 140 operations from 48 real-world web APIs showed that ARTE can efficiently generate realistic test inputs for 64.9% of the target parameters, outperforming the current state-of-the-art approach SAIGEN (31.8%). Additionally, ARTE supported the generation of over twice as many valid API calls (57.3%) as random generation (20%) and SAIGEN (26%), leading to higher failure detection capabilities and uncovering several real-world bugs. These results demonstrate the potential of ARTE to enhance existing web API testing tools and achieve an unprecedented level of automation.

2.3.2 Integration Testing

Integration testing is used to verify the combined functionality of components after their integration. Integration testing can test not only services but messages their interactions and the overall composition as well [124]. There is a whole body of work attributed to integration testing in SOA environment [125, 126, 127, 128, 129].

Chang-ai et al. [130] presented a novel testing framework that generates automated test cases given a service composition in WS-BPEL. The WS-BPEL composition of a process is first converted to a graph model called WS-BPEL Graph Model (BGM) by recursively applying predefined mapping rules. BGM is, in turn, used to generate test scenarios where a scenario corresponds to the execution path of a business process. Test scenarios are generated by applying different rules to different node types in BGM. For instance, in the presence of a fork, condition, or loop the BGM is broken into two segments where the first part corresponds to the relevant fork condition or loop and the second part corresponds to the rest of the process from join, merge or cycle onward to the end node. In the end, both segments are combined to produce the test scenario.

Subsequently, test cases are generated for the scenarios produced in the previous step. For each program path or scenario, a path condition is generated which is actually the conjunction of input constraints for that particular scenario. Moreover, test scenarios are driven by automatically generated test data. A constraint-based technique is used for test data generation by analyzing the input and output requirements of WS-BPEL composite services. Test data is generated for each scenario and the set of test data for all scenarios is called the test suite. Finally, a prototype has been developed for the automatic execution of the test cases and the effectiveness of the framework [130] is validated using an empirical study.

Leal et al. [131] argue that orchestrations can present challenges in ensuring that the intended application is working as expected. Testing is often used as a strategy to mitigate these challenges and validate the system, but the unpredictability of third-party services updates and their possible unavailability, as well as the ever-changing application requirements and quick cycles of application deployment, can create problems for the use of runtime testing on Service-Oriented Architecture (SOA) applications. They propose using Model-Driven Engineering (MDE) to address these problems by applying it to a test framework or process. MDE simplifies the management and development of complex systems by using model abstractions and techniques such as model transformation and model generation, which can be used to improve the interoperability of distributed systems.

The work in [132] presents a method for generating test cases for composite web services implemented in WS-BPEL by utilizing a static call graph. The approach focuses on cases where there is a calling sub-process between WS-BPEL files, and the generated test cases comply with test paths based on branch coverage. There are similar approaches [133, 134] that use path conditions and static call graphs to generate test cases for WS-BPEL compositions. However, these approaches do not consider calling sub-processes in their test generation mechanism.

SAMBA framework [135] is a self-adaptive, model-based online testing framework for runtime functional and regression testing of orchestration-based SOA (Service-Oriented Architecture) applications. SAMBA aims to detect defects introduced by the dynamic and evolutionary behavior of SOA applications by performing testing at runtime, applying both functional and regression test-

ing, and using automatic test case generation from a model of the SOA application. The technique of self-adaptation is used to overcome the unpredictability of evolution in SOA. The effectiveness of the framework was evaluated in an experimental campaign using three different test objectives on three different orchestration-based SOA applications.

EvoMaster [136] is a tool that implements a novel approach for automatically generating integration tests for RESTful web services. The technique aims to maximize code coverage and find faults using HTTP return statuses. It is designed for testing RESTful services in isolation, which is the typical type of testing done by engineers during development. The authors use an evolutionary algorithm called the Many Independent Objectives (MIO) algorithm and aim to improve performance by exploiting typical characteristics of RESTful APIs. They also propose a method to automatically analyze and export white-box information of web services to improve test data generation. The authors present an empirical study on 5 open-source RESTful web services and show that their technique was able to automatically find 80 real faults.

Zhang et al. [137] propose a novel approach to enhance the automated generation of systems tests for RESTful web services using search-based methods. An evolutionary algorithm, MIO, is employed to generate tests to maximize code coverage and fault finding. The MIO algorithm is inspired by the (1+1) evolutionary algorithm. The approach is designed according to REST constraints on the handling of HTTP resources. The paper presents a set of effective templates to structure test actions on one resource, a sampling strategy for the selection of test cases, and specialized mutation operators to improve performance. The approach was implemented as an extension of an existing test case generation tool, EvoMaster [136]. The results of an empirical study show that the proposed approach can significantly improve the performance of test generation by up to 42% on systems that use independent or clearly connected resources.

Hammal et al. [138] proposed a formal method for verification of Business Processes implemented in WS-BPEL and OWL-S. Orchestrations are converted into corresponding automata using rules directed by the respective language constructs. The goal is to extract and merge the single allowed orderings of exchanged messages of these services into a global observable behavioral model in order to analyze it and unveil any errors. The authors proposed a technique for com-

patibility checking of the orchestration of composite web services, which can contribute to the automation of useful tasks in service-oriented architectures. The methodology is also augmented by adding a step for consistency checking of the orchestration against external properties depicted by a WS-CDL choreography. The methodology aims to perform consistency checks between the choreography and its implementation, which is the composition of behavioral interfaces of cooperating (basic or composite) services and checks whether each of these services conforms to the contract.

Arcuri et al. [139] extend search-based software testing by incorporating SQL heuristics to generate more effective test data for web and enterprise applications that interact with external systems, such as SQL databases. The goal is to intercept every SELECT query made by the system under test (SUT) and optimize the WHERE clauses of these queries to return non-empty sets of data. This heuristic is integrated with bytecode heuristics to guide the search and generate the right sequence of inputs on the SUT. However, populating the database with data that enables interesting application behavior might not be simple and the approach also enables the insertion of SQL data directly from the test cases, which introduces new research challenges. The technique is implemented as an extension of EvoMaster [136].

De Jager and De Gouw [140] propose tool-supported approaches for formal verification and test case generation of Straight-Through Processes (STPs). An STP is a subset of BPEL and is specifically designed for short-lived processes. STPs are commonly used to automatically process a large number of requests without human intervention. Both approaches are based on grammar, one for static verification through parsing and another for automatic test-case generation using Prolog. The tool suite supports both protocol and data-oriented properties of STP event traces and is validated and compared to existing approaches in an industrial case study with the Dutch Tax and Customs Administration.

Leal et al. [141] focus on service orchestrations and discuss the challenges and limitations of the validation process especially the variety in Model-Based Testing (MBT) tools and diversity in web services and their descriptions. In order to overcome these challenges, Leal et al. advocate for the use of Model-Driven Engineering (MDE) techniques of model transformation and

artifact generation to improve the runtime MBT process on SOA applications. The study presents two metamodels, three transformations to guarantee interoperability with different orchestration languages and testing tools, and a running implementation based on an existing framework. The proposed approach was evaluated in a case study with orchestrations expressed in two different languages.

2.3.3 Regression Testing

Regression testing is used to ensure the quality of software when parts of the software evolve [142]. Test suites are re-executed to assure the correctness of evolved software. One concern in the context of regression testing is that the entire suite may be infeasible to execute due to constraints on testing resources. Therefore test cases are prioritized to gain as much coverage as possible. There is a whole body of work in regression testing [143, 144, 145, 146, 147] and test case prioritization in the context of web services composition. None of them, however, consider the case of a web service being evolved during the execution of the test cycle.

In [148], Lijun et al. proposed a preemptive approach to addressing the challenges of regression testing in a dynamically changing SOA environment, that is, when external services evolve within a test session. The preemptive regression testing (PRT) [148] preempts the current testing session and creates a new sub-session if a change is detected. Upon completion of the sub-session, the preempted session is resumed.

The preemption is made according to fix, reschedule, and fix-reschedule strategies. Mainly, fix strategy identifies the missing coverage that has been covered in the previous execution of the tests. Reschedule strategy, on the contrary, finds the new items that are covered but were missing in the previous execution. Test prioritization is then adjusted accordingly. Fix-reschedule is a hybrid approach that first runs fix strategy followed by the rescheduling strategy. The approach also compares existing prioritization schemes from the literature and proposes some new schemes for the execution of test strategies. The work [148] also conducted an empirical study to prove the effectiveness of the proposed technique.

Ji et al. [149] proposed an approach for test case selection in regression testing of BPEL

composite services. The approach addresses the issues of data flow testing criteria and the impact of three types of changes (Process Change, Binding Change, and Interface Change) on data flow. It uses a two-level model (XCFG and WSDM) to model the behavior of the BPEL process and the interface information of composite service and partner services, respectively. Change impact analysis is used to identify affected definition-use pairs by comparing the two-level models of the baseline and evolved versions. The approach generates testing paths to cover the affected definition-use pairs, selecting test cases based on path condition analysis.

Godefroid et al. [150] introduce regression testing for REST APIs. The technique considers two types of regressions: those in the API specification, and those in the service itself. The study uses differential testing to detect these regressions by comparing the behavior of different versions of client-service pairs and identifying differences. Several technical challenges are also addressed, such as how to compare the outputs of a cloud service that contains non-determinism, and how to handle out-of-order or newly appearing requests when updating the specification. The study then presents a historical analysis of 17 versions of Microsoft Azure networking APIs to evaluate the effectiveness of the differential regression testing technique, and it was able to detect 5 regressions in the official specifications and 9 regressions in the services themselves.

Chen et al. [151] evaluate the effectiveness of the PageRank algorithm in prioritizing test cases for microservices-based systems. The proposed approach is called TCP-SR and ranks services based on API gateway logs and uses these rankings to calculate the weights of test cases. These weights can then be used to order test cases using single-objective and multi-objective strategies. To evaluate the effectiveness of TCP-SR, an empirical study was conducted using four microservice systems. The results indicate that TCP-SR has a much higher fault detection rate compared to the random prioritization technique and performed similarly to the prioritization technique based on WS-BPEL, but with a much lower cost in terms of prioritization time.

2.3.4 Summary

Testing is the most common mechanism for verifying the correctness of the generated candidate fixes. A test suite is a collection of test cases that, generally, define the correctness of a program.

So, the fundamental characteristic of a test suite is that the correct programs must pass all the test cases in the test suite and the faulty programs must fail at least one test case in the test suite. This characteristic calls for a comprehensive test suite. But, on the other hand, larger test suites force the fault resolution approach to spend much of the time in the verification of the generated candidate fixes and adversely impact its efficiency.

We also use a test suite for the verification in both EGV (discussed in chapter 3) and CFR (discussed in chapter 4). In both approaches, we define the correctness of a BP based on its ability to pass all the test cases in the test suite. Similarly, the existing fault-free BPs in CFR are fault-free because they pass all the test cases. So, the test suite has a central importance in the verification process of the generated candidate fixes and significant impact on the efficiency of the fault resolution approach. However, we leave the choice of test suite to the user and they provide the test cases along with the faulty BP to the fault resolution approach. The user can decide to manually craft the test cases or choose any of the test case generation schemes discussed in this section.

Chapter 3

An Efficient Generate & Validate Approach for Fault Resolution

This chapter presents the Efficient Generate-and-Validate (EGV) approach for fault resolution in BPs. The proposed approach is an improvement of the basic Generate-and-Validate (G&V) approach. So, first, we revisit the methodology of basic G&V for fixing faults in software followed by a detailed presentation of EGV. We also introduce the formalism and notations used to explain EGV in this chapter and collaborative and hybrid fault resolutions in chapter 4.

3.1 Generate and Validate (G&V)

G&V is an automated program repair technique that uses a faulty program and a set of passing and failing test cases as input to generate candidate fixes by heuristically searching the program space. The generated fix candidates are validated by running all available test cases [18, 17]. G&V requires a fault localization mechanism for identifying suspected code blocks in a faulty program. Once the suspicious code block is identified, it is followed by the generation of candidate fixes and their validation.

Generation: In this step, a repair tool generates a set of candidate patches or fixes for the bug. These candidates are typically generated using mutation-based, pattern-based, or machine-learning-

based techniques as discussed in Section 2.2.

Validation: In this step, the repair tool tests each of the candidate fixes to determine whether they actually fix the bug. This is typically done using an oracle that can be in the form of a formal model or a test suite. If a test suite is chosen as an oracle, all the test cases can be executed to verify the correctness of the generated fix, or test cases can be prioritized using regression testing as discussed in Section 2.3.3. Below we present the working of basic G&V in C++ and in the domain of Business Processes.

G&V in C++

Listing 3.1 shows a divide function that takes two integer parameters *a* and *b* and returns the result of their division as an integer.

Listing 3.1: An incorrect divide function in C++

```
1 int divide(int a, int b) {
2     return a / b;
3 }
```

There are two bugs in Listing 3.1. The first one is that the program attempts to divide by zero whenever a 0 is assigned to parameter *b* when calling the function. Division by zero is not allowed and will cause the program to crash at line 2. The second bug is that this piece of code assumes that the result of an integer division will always be an integer. This will result in a loss of value when the result of division is not an integer but a real number.

Listing 3.2: First candidate fix for Listing 3.1

```
1 int divide(int a, int b) {
2     return a + b;
3 }
```

Listing 3.3: Second candidate fix for Listing 3.1

```
1 int divide(int a, int b) {
2     return a * b;
3 }
```

Listings 3.2 and 3.3 show two candidate fixes typically generated using mutation-based candidate generation by replacing the arithmetic operator $/$ with $+$ and $*$ respectively on line 2. Obviously, both candidate fixes are incorrect and do not fix the bug. They, instead, cause the already passing test cases to fail.

Listing 3.4: Third candidate fix for Listing 3.1

```
1 int divide(int a, int b) {
2     if (b == 0) {
3         return 0;
4     }
5     return a / b;
6 }
```

Listing 3.4 presents a fix that is closer to the real solution. It checks for the value of parameter *b* and if it is zero, zero is returned from the function instead of performing the division. This prevents the program from crashing program path. This fix, however, does not save the loss of value that can occur whenever the result of integer division is not an integer, that is when we call the *divide* function with *a = 3* and *b = 2*. The role of the test suite (or any oracle in general) is fundamental in determining the correctness of a program and the generated candidate fixes. For instance, if there is no test case in the test suite that results in a floating point outcome of the division, the candidate fix of Listing 3.4 will be marked as a correct fix to the *divide* function.

Listing 3.5: Fourth candidate fix for Listing 3.1

```
1 double divide(int a, int b) {
2     if (b == 0) {
3         return 0.0;
4     }
5     return (double) a / b;
6 }
```

Listing 3.5, on the other hand, not only caters to the division-by-zero problem but prevents the loss of value when the result of integer division is not an integer. This is achieved by changing the return type of the divide function and returning a *double* value instead of an *int*. The candidate fixes in Listings 3.4 and 3.5 are hard to generate using mutation-based fix generation. They contain more sophisticated changes in the source program like the addition of a conditional path for checking division-by-zero problems. These candidate fixes are generally generated by pattern-based or machine-learning-based techniques that inspect a large amount of code and find fault/bug patterns and their corresponding fixes.

G&V in Business Processes

Fig. 1.5 represents a faulty BP with control flow fault by exchanging the order of *createOrder* and *calc_sales_tax* service. Although there are many candidate fixes that can be generated by G&V approach, we will only focus on three representative candidates obtained by order exchange mutation.

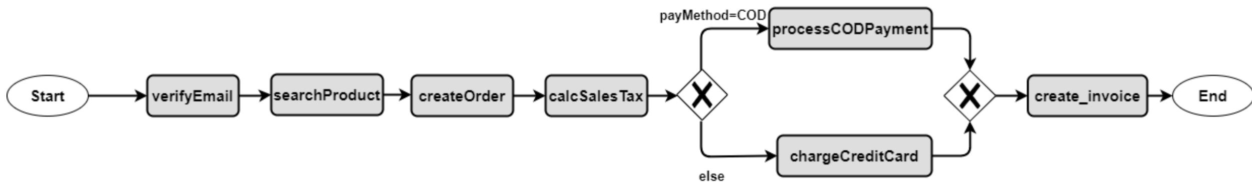


Figure 3.1: First candidate fix for BP in Fig. 1.5

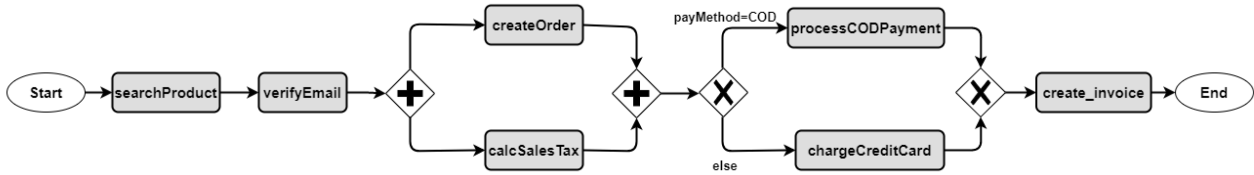


Figure 3.2: Second candidate fix for BP in Fig. 1.5

Fig. 3.1 proposes a candidate fix by replacing the order of *searchProduct* and *verifyEmail* service operations. Although the proposed candidate fix does not induce new faults in the BP yet it fails to address the existing fault in BP which is a data anti-dependence between *createOrder* and *calc_sales_tax* service operations.

Fig. 3.2 is an interesting candidate fix in the sense that it places *createOrder* and *calc_sales_tax* in parallel execution order. This partially fixes the fault and causes the BP to execute successfully only when the result of *calc_sales_tax* is available before making a request to *createOrder* service.

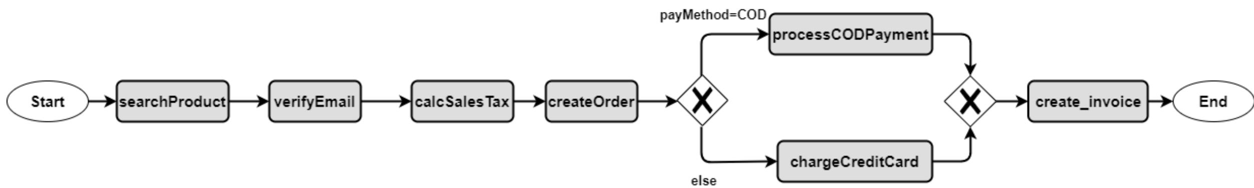


Figure 3.3: Third candidate fix for BP in Fig. 1.5

Fig. 3.3 shows a candidate fix that places *createOrder* after *calc_sales_tax* in a sequential execution flow. The suggested change causes the *createOrder* service to execute only when the result of *calc_sales_tax* service is available, thus, removing the data anti-dependence between the two services.

The basic G&V approach has high computation overhead because it generates candidate fixes in a brute-force manner by considering all possible mutations of elements in suspected regions. We improve the efficiency of G&V by applying only a small selective set of candidate fixes instead of brute force application of all fixes. The proposed approach called *efficient G&V (EGV)* leverages mutation-based fault localization in combination with program slicing to improve localization

accuracy and considers a minimal set of suspicious code blocks for generating candidate fixes.

3.2 BP Fault Resolution – Problem Formulation

Definition 1. **Business Process:** *A business process BP is denoted by a typed Graph $G = (L, T, V, E, \mathcal{E}, v^{start}, v^{end}, v^{user})$ where:*

- *L is the universal set of labels.*
- *$T = \{$Service Operation, Xor Split, Xor Join, And Split, And Join, Attribute$\}$ is a set of types for each $v \in V$. The type is accessible through the operator $v.type$.*
- *$V \subseteq L \times T$ is the set of vertices. Each vertex has a label assigned from L and a type assigned from T.*
- *$E \subseteq V \times V$ is the set of edges in G.*
- *$\mathcal{E} = \{$Boolean Expression, true, false$\}$, for each edge.*
- *The vertex v^{start} corresponds to the start activity of the BP.*
- *The vertex v^{end} corresponds to the terminal activity of the BP.*
- *The vertex v^{user} denotes a user and it is connected to all those attribute vertices whose value are provided by the user.*

Fig. 3.4 shows a simple e-commerce BP graph. Each vertex in this graph is annotated with its label. In addition to v^{start}, v^{end}, and v^{user}, there are three types of vertices in this e-commerce BP graph: i) service operations represented as shaded gray boxes (e.g., *createOrder*, *completeOrder*, etc.); ii) attributes corresponding to input/output parameters of service operations represented as white rectangular boxes (e.g., *cart_id*, *orderId*, etc.); and iv) XOR splits and joins represented as a diamond with X.

The edges in the typed BP graph depicted in Fig. 3.4 are characterized as: i) control flow edges represented as solid arrows (e.g., the edge from *createOrder* to *xor split*); and ii) dataflow edges.

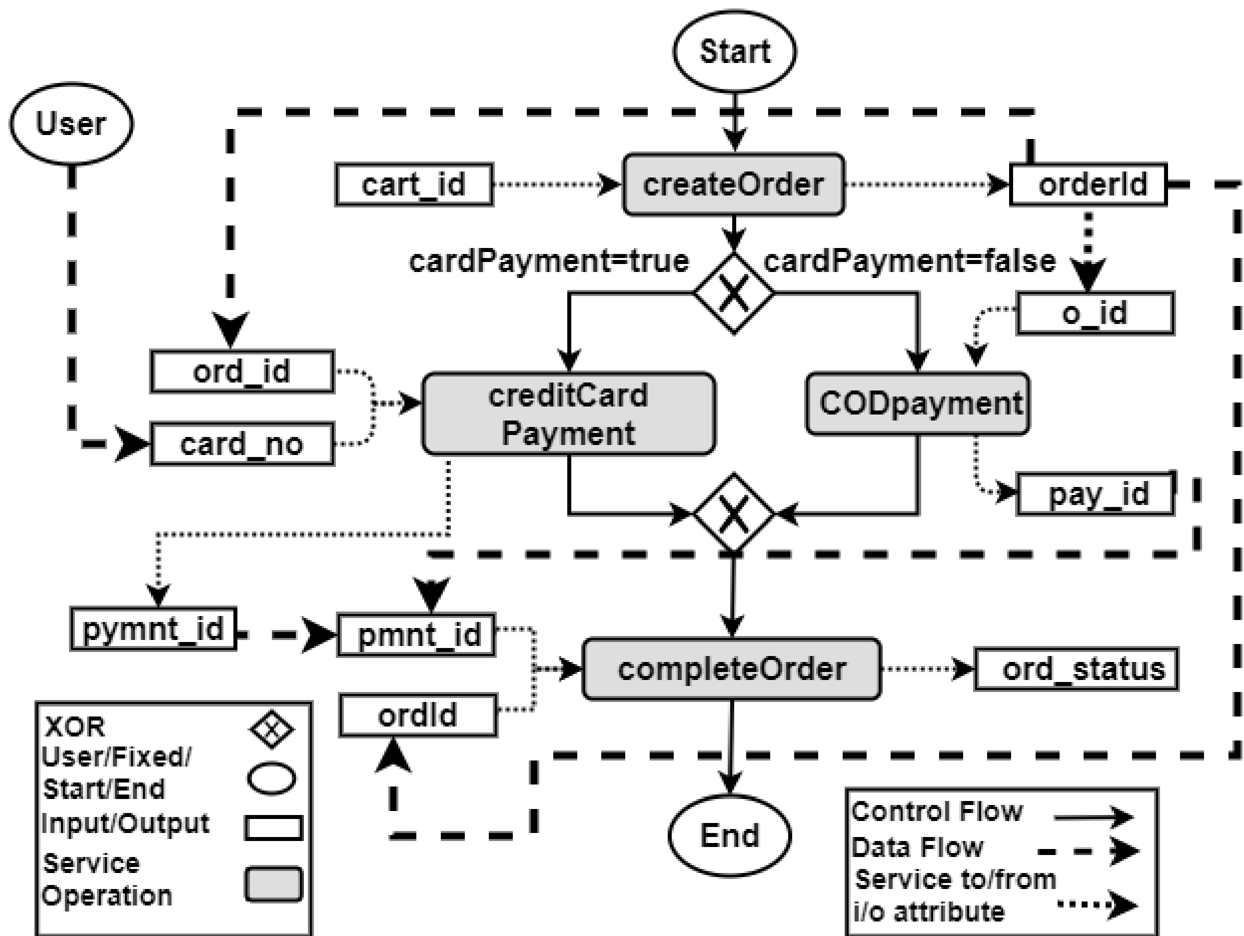


Figure 3.4: Example of an e-commerce BP graph

There are two types of dataflow edges. The first type, represented as an arrow with a dotted line, is drawn between a service operation and its input or output attribute (e.g., the edge from *createOrder* service operation to its output attribute *orderId*) or between input attribute *ord_id* and its service operation *creditCardPayment*. The second type of dataflow edge is represented as an arrow with a dashed line that originates from the output attribute of a service operation and terminates at the input attribute of another service operation. These edges model the variable assignments (e.g., the edge from *orderId* to *o_id* where *orderId* is the output attribute of *createOrder* service operation and its value is assigned to *o_id*, which is the input attribute of service operation *CODPayment*).

Illustrative Example of a Faulty BP: Fig. 3.5 shows a small BP graph representing a faulty version of an elementary BP from the e-commerce domain. Shaded gray boxes represent activities

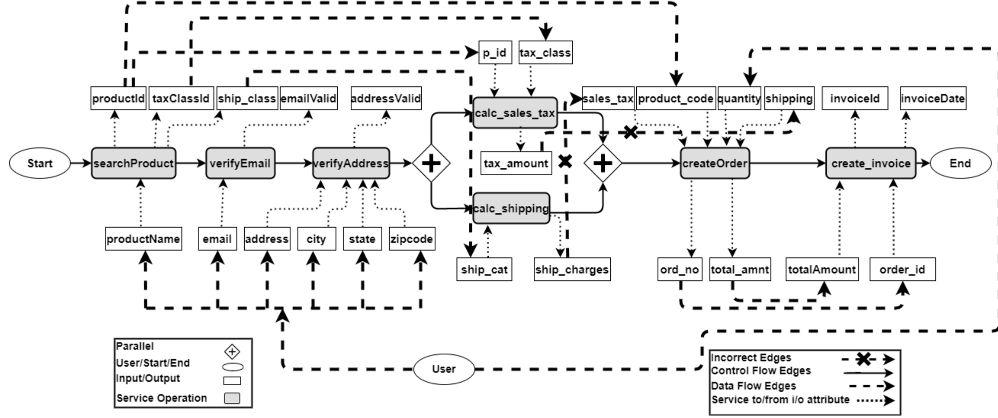


Figure 3.5: An example BP graph (G_f) from e-commerce sale order processing domain.

corresponding to the invocation of web service operations e.g., *searchProduct*, *verifyEmail*, etc. White rectangular boxes represent the input and output attributes of the service operations e.g., *productId*, *taxClassId*, etc. Control flow edges in the BP are denoted by solid arrows. Arrows with dotted lines denote dataflow edges that link each service operation with its input/output attributes. Arrows with dashed lines model the variable assignment which is essentially a dataflow from the output of one service operation to the input of another service operation (e.g., the edge from *productId* to *product_code* where *productId* is produced by *searchProduct* and it is assigned to *product_code*, which represents an input attribute of the *createOrder* service operation).

Faulty BP: Given a set of test cases specified for a BP, we consider the BP as a faulty BP, if it fails one or more of the test cases. The BP graph G_f, shown in Fig. 3.5, is a faulty BP where the edge from $ship_charges$ to $sales_tax$ and the edge from tax_amount to $shipping$ (marked with $\times$) correspond to the incorrect variable assignments.

For fault resolution, we need to discover fixes (mutations) that can remove the faults and allow for the correct execution of the faulty BP. The BP fault resolution problem addressed in this work is formally stated below.

Definition 2. BP Fault Resolution Problem: *Given a faulty BP, G_f, and a set of test cases, $T = \{t_1, \ldots, t_m\}$, compute a minimal set of candidate fixes, $\mathcal{C} = \{G_{c1}, G_{c2}, \ldots, G_{ck}\}$, that when applied to G_f produces a BP that successfully passes all the test cases in T.*

Note that we do not address the test case generation problem for BPs in this dissertation. We assume that the test cases are either provided by the user or existing test case generation techniques [116, 9, 152, 136, 153] can be applied for this purpose as discussed in Chapter 2.

3.3 Efficient G&V approach (EGV) for fault resolution

For fault detection and resolution, we propose an integrated approach that builds on the generate-and-validate (G&V) methodology and improves its efficiency by generating a small set of candidate fixes for BP repair. Fig. 3.6 shows the main steps of the proposed EGV approach. EGV takes as input a faulty BP graph (G_f) and a set of test cases and attempts to resolve faults in G_f. There are four key steps that are executed repeatedly until all faults are resolved (as per the execution on the test suite). First, the test cases are executed to check the correctness of the given BP, G_f. If it results in an unexpected/incorrect output, fault localization is performed (discussed in Section 3.3.1) to identify the location(s) where faults are observed in the faulty BP. These locations are referred to as fault observation points (fop), which may not necessarily correspond to the actual source of the fault in the BP. The source of the actual fault might be present at a location prior to fop.

Specifically, we perform statistical fault localization [29] to identify fop. For a given fop, we perform program slicing [61] to obtain suspicious code blocks. These suspicious code blocks are referred to as BP slices. We use the BP slices that lie between the starting vertex (v_f^{start}) of G_f and the given fop for generating candidate fixes. Fault localization and BP slicing help us keep the generated number of candidate fixes manageable but still relevant. Once the BP slices are identified, we apply the different mutation operators and their combinations to obtain mutants of G_f which are called candidate fixes. Then, for the given fop, we run the test cases against each candidate fix to check if the faults are resolved without introducing any new faults. For executing test cases, first, the BP code is generated and deployed. If a candidate fix removes all the faults then our approach terminates and returns the candidate fix that passes all the test cases. In case the faults up to the given fop are fixed, but the execution of test cases results in faults at a later point in the BP, we repeat the entire process to identify subsequent fop and candidate fixes. This

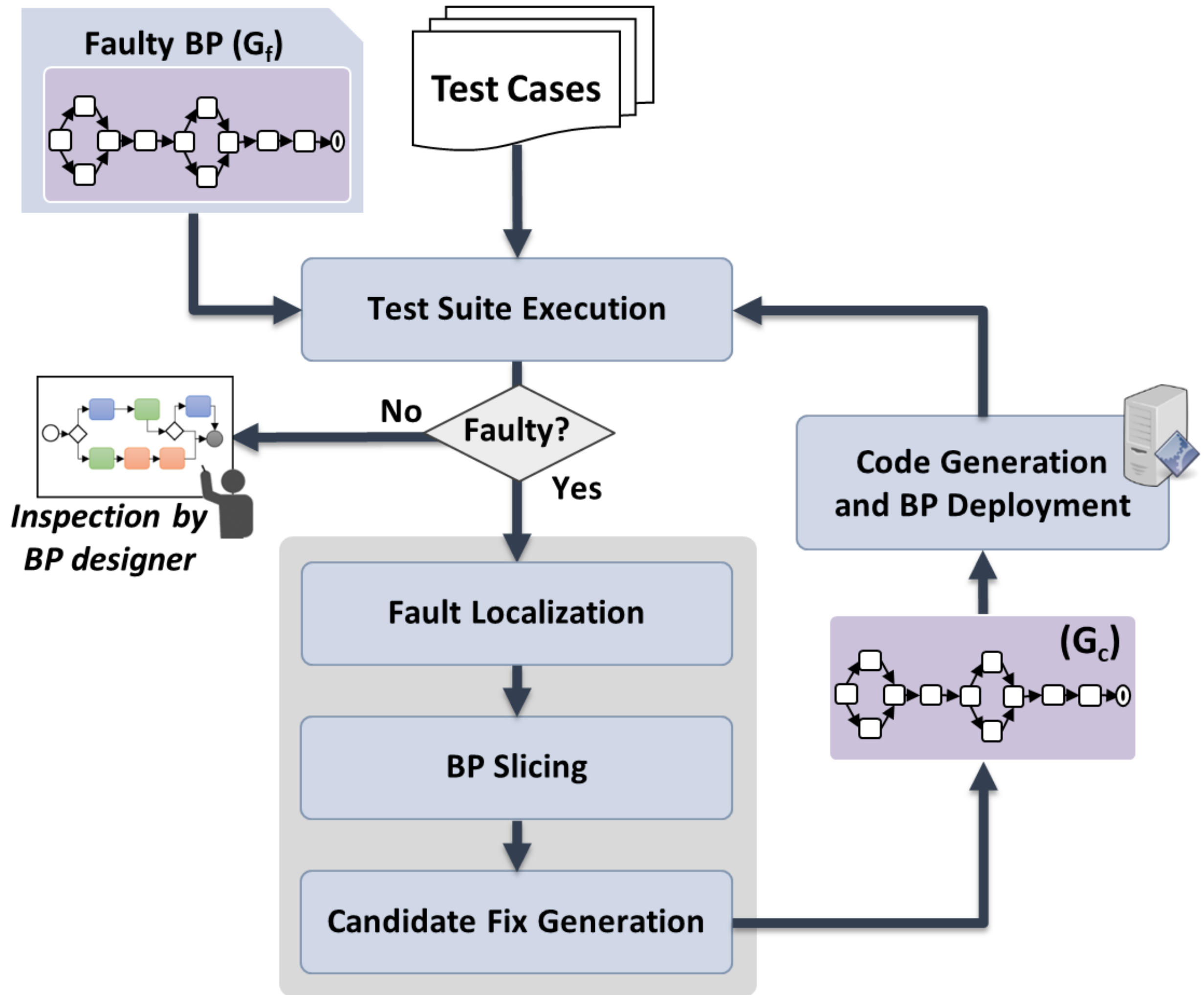


Figure 3.6: Efficient G&V fault resolution approach for BPs.

process is repeated in an iterative manner until all the faults are resolved or all the candidate fixes are exhausted.

Algorithm 3.1 outlines the steps required to fix a faulty BP. Lines 1 and 2 find the fop and BP slice respectively using statistical fault localization and BP slicing as discussed in Sections 3.3.1 and 3.3.2. Next the relevant BP slices between the starting vertex (v_f^{start}) of G_f and the fop is extracted for generating candidate fixes (Lines 3 – 10). Candidate fixes are generated by successively applying mutation operators and their combinations on the input BP G_f. For each candidate fix, G_c is validated to check if it passes the failed test cases up to the computed fop

ALGORITHM 3.1: EGV_FaultResolution

Input: $G_f = (V_f, E_f, \mathcal{E}_f, v_f^{start}, v_f^{end}, v_f^{user})$ - faulty BP

Input: T - Set of test cases

Output: G_c - Corrected BP

1: $fop \leftarrow localize_fault(G_f)$
2: $slice \leftarrow get_process_slice(G_f, fop)$
3: $S \leftarrow \phi$
4: **for** each vertex $v \in slice$ **do**
5:     **if** $distance(v, v_f^{start}) \leq distance(fop, v_f^{start})$ **then**
6:         $S \leftarrow S \cup v$
7: **for** each edge $(u, v) \in slice$ **do**
8:     **if** $distance(u, v_f^{start}) \leq distance(fop, v_f^{start})$ **or** $distance(v, v_f^{start}) \leq distance(fop, v_f^{start})$ **then**
9:         $S \leftarrow S \cup (u, v)$
10: $\mathcal{C} \leftarrow generate_candidate_fixes(S)$
11: **for** each $G_c \in \mathcal{C}$ **do**
12:     Apply each test case $t \in T$ on G_c
13:     **if** G_c fixes faults up to fop **then**
14:         **if** G_c passes all the test cases for the entire BP **then**
15:             **return** G_c
16:         **else**
17:             **return** EGV_FaultResolution(G_c, T)
18: **return** NULL

(Line 13). If any such G_c is found, it is tested against all the test cases for the entire BP (Line 14). If it passes all these test cases, then all the faults with respect to the given test cases have been removed. The resulting BP is returned to the user (Line 15). Otherwise, we recursively call the EGV fault resolution procedure (Line 17) until all the faults are fixed or the entire space of mutants is exhausted. We provide a detailed description of each component of Fig. 4.1 below.

3.3.1 Fault Localization

Fault localization aims to locate and isolate faulty software components or bugs to determine the likely causes of errors or software failures [154, 21]. For fault localization in a BP, we employ a statistical analysis-based debugging approach [29]. This approach considers predicate evaluation against program elements in correct as well as incorrect program executions. A predicate is assumed to be fault-relevant if the pattern of evaluation in an incorrect run significantly deviates from the correct ones. Predicates are ranked in order of their computed fault-relevance scores.

In the context of BPs, we establish predicates for each of the branching conditions as well as for each service invocation, their execution status, and their impact on the test case result. Each predicate is assigned a fault-relevance score depending upon its execution status in passing and failing runs of a BP for the given test suite. The location having the highest fault-relevance predicate score is chosen as a fault observation point (fop) for the subsequent steps in our approach. For instance, in Fig. 3.5, the $createOrder$ service fails with an exception due to incorrect mapping of $sales_tax$ and $shipping$ input attributes. This failure results in a high predicate score for $createOrder$ than any other BP element and it is selected as an fop for BP slicing and candidate fix generation.

3.3.2 BP Slicing

Once the fop is identified, we employ program slicing to identify suspicious code blocks (BP slices). For this, we employ the BPELSwice approach by Sun et. al [61]. BPELSwice uses predicate switching and program slicing to obtain BP slices from BPEL programs. Specifically, it switches the conditional statements and verifies the modified BP against the test cases. If all test cases are passed then it takes the backward slice from the conditional statement and takes the elements of the BP that write to the variables used in the conditional statement. If the predicate switching does not result in the passing of all the test cases, BPELSwice takes the backward slice from the incorrect/ unexpected BP outputs.

For instance, in Fig. 3.5 the BP fails on invocation of $createOrder$ service operation due to incorrect inputs and produces unexpected output. Hence, the BP slice will contain the edges and vertices of the BP graph that are connected to $createOrder$ in the control flow and the service

operations and their attributes that provide input to $createOrder$. Fig. 3.7 depicts the slice of BP shown in the illustrative example (Fig. 3.5). Fig. 3.7 does not include $verifyEmail$ and $verifyAddress$ service operations because they do not provide any input to $createOrder$ service directly or indirectly nor are they adjacent to it in the control flow. Furthermore, the slice also does not contain the input attributes of $searchProduct$ service operation because all of its inputs are provided by the user and hence by the test cases that are assumed to be valid. After the slice has been identified, we select the part of the slice that lies before fop. In this example, the whole slice will be selected because all elements of the slice occur before $createOrder$ which is the fop in this case.

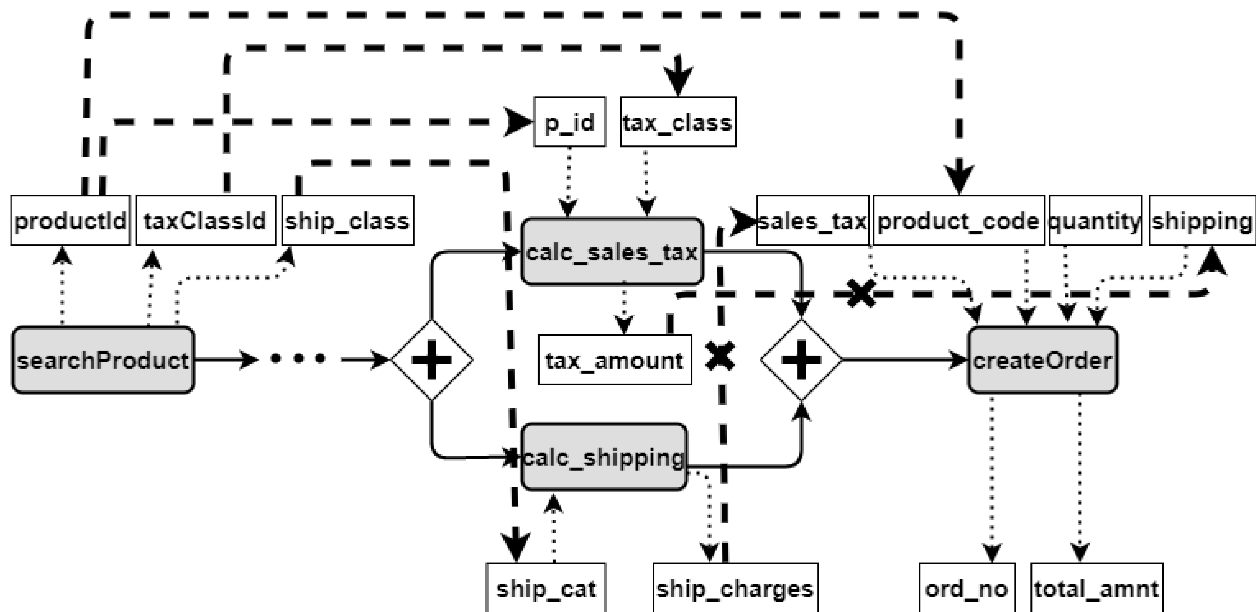


Figure 3.7: Slice of faulty BP G_f.

3.3.3 Candidate Fix Generation

After identifying the filtered slice before fop, we generate candidate fixes by applying different mutation operators shown in Table 1.1. We only apply semantically meaningful mutation operators for generating candidate fixes. This also reduces the number of candidate fixes. Specifically, we do not change the order of data-independent services in the control flow for generating candidate

fixes. Moreover, we do not use ECN operator because it generates a large number of candidate fixes. Furthermore, we do not consider path or activity removal operators (AIE, AEL) because they may change BP scope.

Table 3.1: Candidate fixes for BP graph in Fig. 3.5.

Candidate Fix	Mutations
m_1	$(productId, pid)^-$, $(taxClassId, tax_class)^-$ $(productId, tax_class)^+$, $(taxClassId, pid)^+$
m_2	$(productId, product_code)^-$, $(taxClassId, tax_class)^-$ $(productId, tax_class)^+$, $(taxClassId, product_code)^+$
m_3	$(taxClassId, tax_class)^-$, $(ship_class, ship_cat)^-$ $(taxClassId, ship_cat)^+$, $(ship_class, tax_class)^+$
m_4	$(productId, product_code)^-$, $(ship_class, ship_cat)^-$ $(productId, ship_cat)^+$, $(ship_class, product_code)^+$
m_5	$(productId, pid)^-$, $(ship_class, ship_cat)^-$ $(productId, ship_cat)^+$, $(ship_class, pid)^+$
m_6	$(ship_charges, sales_tax)^-$, $(tax_amount, shipping)^-$ $(ship_charges, shipping)^+$, $(tax_amount, sales_tax)^+$

In the slice shown in Fig. 3.7 there are no conditional statements and expressions. Additionally, the order of service operations will not be changed because no service operation depends upon data produced by a service operation that appears later in the control flow. Therefore, the only applicable mutation operator is ISV, which is equivalent to replacing the data flow edges between pairs of attribute-type vertices having the same data type. Table 3.1 shows the generated candidate fixes. In candidate fix m_1, $pid = productId$ mapping is replaced with $pid = taxClassId$ and $tax_class = taxClassId$ mapping is replaced with $tax_class = productId$. Note that m_6 is the candidate fix that actually resolves the fault by removing incorrect mapping with the correct mapping of $shipping$ and $sales_tax$ attributes. During the generation of candidate fixes, $sales_tax$ or $shipping$ is not mapped to any of $productId, taxClassId$ or $ship_cat$ because they belong to

different data types.

3.3.4 Validation of Candidate Fixes

Finally, we translate each candidate fix into executable BP code and deploy it for testing. The entire test suite is executed against each candidate fix to check that the resulting BP is fault-free with respect to the given test suite. If such a fix is found, the fault-free BP after the application of the candidate fix is returned to the user. If a candidate fix passes some, but not all, of the test cases that originally failed, we use that candidate fix for further discovery and repair of faults (Algorithm 4.1, Line 17).

3.4 Experimental Evaluation

We have performed extensive experiments to evaluate the performance of the proposed EGV fault resolution approach.

3.4.1 Dataset

For evaluation, we used a random fault injection technique to generate faulty BPs from a fault-free BP. we started with a correct BP from the insurance domain. The selected BP was composed of 26 Web service operations with 3 branches and 2 parallel structures. To generate faulty BPs, we applied random combinations of the mutation operators listed in Table 1.1. This resulted in 208 faulty BPs that were used for validation. The number of mutation operators applied to generate a faulty BP varied between 1 and 4 with a mean of 2.24 and a standard deviation of 0.81. Faults from each fault category and their random combinations were tested.

We created a suite of test cases based on the expected output covering all branching paths of the correct BP. Now, a BP is considered as correct if it passes all the test cases in the suite. We applied fault localization on each faulty BP to determine their fop(s). The number of fop(s) varied between 1 and 4 for the generated faulty BPs.

After the identification of fop, candidate fixes are generated by selectively applying different mutation operators in the BP fragment around the fop. This produces candidate fixes which are, then, evaluated by executing the entire test suite.

3.4.2 Results

For comparison of EGV with basic G&V, we performed experiments on 40 BPs as shown in Table 3.2. Basic G&V outperforms EGV with 0.9 accuracy as opposed to 0.65 of EGV. But the higher accuracy is achieved at the cost of generating a lot more candidate fixes. Basic G&V, on average, validates 4139 candidate fixes for each faulty BP whereas EGV only uses 10 candidate fixes. The accuracy of EGV is quite low (0.42) for variable assignment fault types compared to G&V with an accuracy of 0.84. The accuracy of both EGV and G&V is almost the same for other fault types, though EGV uses a much smaller number of candidate fixes.

Table 3.2: Accuracy of EGV and Basic G&V.

Category	No. of BPs	Accuracy		Candidate Fixes	
		G&V	EGV	G&V	EGV
Variable assignment faults	24	0.83	0.42	2192	6
Expression	5	1	1	464	29.6
Control Flow excluding Element removal	11	1	1	9351	6
Overall	40	0.9	0.65	4139.47	10

We will present more results for EGV and compare them with collaborative fault resolution (CFR) and hybrid approaches in chapter 4.

3.5 Chapter Summary

In this chapter, we proposed Efficient G&V (EGV) approach for fault resolution in Business Processes. EGV significantly outperforms basic G&V in the number of generated candidate fixes. However, the accuracy of EGV is much lower than basic G&V. Additionally, there are types of faults that cannot be resolved using EGV or by any approach based on basic G&V. We will address this problem in chapter 4 by using an existing repository of similar BPs for fault resolution.

Chapter 4

Fault Resolution with Collaborative and Hybrid Approaches

In this chapter, we will present a collaborative fault resolution approach for BPs that exploits the knowledge of existing fault-free BPs to fix a faulty process. Later in the chapter, we will present a hybrid approach for fault resolution that combines collaborative fault resolution with Efficient Generate and Validate (EGV) approach.

4.1 Introduction

As discussed in Section 1.2 and illustrated in Fig. 1.1, a cloud-based environment offers a wide range of resources for developing and deploying web services and business processes interactively. The cloud's resources are not limited to computational services; they also include software components that can be exposed to users and configured for reuse. Afzal et al. [2] utilized existing business processes hosted on the cloud to implement their approach for automatically composing structured and knowledge-driven business processes. In this dissertation, we aim to take advantage of these existing business processes to go one step further and use the knowledge they provide to identify and resolve faults in faulty business processes. Essentially, we can consider the business processes and services hosted on the cloud and serving user requests as fault-free. Their structure

can serve as a reference not only for composing new business processes but also for identifying and fixing faults in faulty business processes.

In the services cloud environment of Fig. 1.1, we refer to a BP composed by a BP designer that contains fault(s) as *faulty BP*. A faulty BP may include web services that are also used in the BPs of other users. We refer to these other users' BPs that have one or more services common with the faulty BP as *existing BPs* and they are considered to be fault-free. CFR exploits the knowledge of existing BPs with common services to detect and resolve faults in a faulty BP. This is the unique and novel aspect of the CFR as compared to the existing BP fault detection and debugging approaches, such as [60, 130, 14, 61, 18] that examine the faulty BP in isolation. Therefore, we refer to our proposed approach as a collaborative fault resolution approach. Essentially, CFR performs a pairwise comparison of a faulty BP with related BPs of other users to identify their structural and semantic differences with the faulty BP. All of the pair-wise differences are then holistically analyzed to compute BP transformation rules that modify the faulty BP by adding and/or removing some of its structural components. Such modifications may resolve the fault but change the BP workflow to such an extent that it does not meet its original requirements or goal. Therefore, our objective is to find a set of modifications such that: (i) these modifications, when applied, remove the fault in the BP; and (ii) the set of modifications is as small as possible, to reduce the likelihood that the goal and output of the BP changes.

The proposed approach assumes the following:

1. All of the *existing BPs* considered for resolving faults in a given *user BP* are fault-free. Note that we consider a BP as fault-free if it passes all the relevant test cases. Moreover, we have complete knowledge of all the existing BPs in terms of their control flow and data flow.

2. There is no syntactic or semantic heterogeneity among the functionally similar web service operations across BPs. For example, if two or more e-commerce BPs require computation of sales tax, then either they use the same web service operation or the corresponding web service operations have the same name, attributes, preconditions, and post-conditions.

These assumptions are quite natural. Specifically in a service cloud environment for BP development and deployment, the cloud service provider hosts and manages BPs of different organiza-

tions and has complete knowledge of such BPs [2]. Moreover, BPs running in a production environment for a sufficiently long time are likely to be correct and fault-free. Also, if heterogeneity exists between operation and/or attribute names across BPs, we can employ the existing attribute-based matching approaches [2, 155, 156] to resolve differences in operation/attribute names before continuing with the fault resolution approach.

We say that a BP is a structurally valid BP if all the service operation vertices, control flow vertices (e.g., XOR join/split, parallel join split), and attribute vertices have valid predecessors and successors. This is formally stated in the following definition.

Definition 3. Structurally valid BP: A BP, $G = (L, T, V, E, \mathcal{E}, v^{start}, v^{end}, v^{user})$, is structurally valid, if and only if:

- $\forall v \in V - \{v^{start}, v^{user}\}$ and $v.type \neq Attribute$, $\exists u \in V - \{v^{end}, v^{user}\}$ such that $(u, v) \in E$ and $u.type \neq Attribute$
- $\forall v \in V - \{v^{end}, v^{user}\}$ and $v.type \neq Attribute$, $\exists w \in V - \{v^{start}, v^{user}\}$ such that $(v, w) \in E$ and $w.type \neq Attribute$
- $\forall v \in V - \{v^{user}\}$ and $v.type = Attribute, \exists u \in V - \{v^{start}, v^{end}\}$ such that $(u, v) \in E$ and $(u \in \{v^{user}\}$ or $u.type \in \{Service\ Operation, attribute\})$

We now formally specify the collaborative fault resolution problem. Note that given a set of test cases for a BP, we only denote the BP as a faulty BP if it fails one or more of the test cases. Then, given a faulty BP and a set of correct BPs, the collaborative fault resolution problem aims to determine a set of transformations that, when applied to the faulty BP, remove the fault(s) and enable its correct execution with respect to the given set of test cases.

Definition 4. Collaborative Fault Resolution Problem:
Given

- *a set of BPs, $\mathcal{B} = \{G_1, G_2, ...G_n\}$ where each G_i corresponds to some existing BP that is assumed to be fault-free,*

- *a faulty BP,* G_f
- *a set of test cases,* $T = \{t_1, \dots t_m\}$

Compute the minimal set of transformation operations $\tau_F = \{\tau_1, \dots \tau_k\}$ *that, when applied to* G_f*, result in a BP that successfully executes all the test cases in* T*.*

4.2 Proposed Approach for Collaborative Fault Resolution

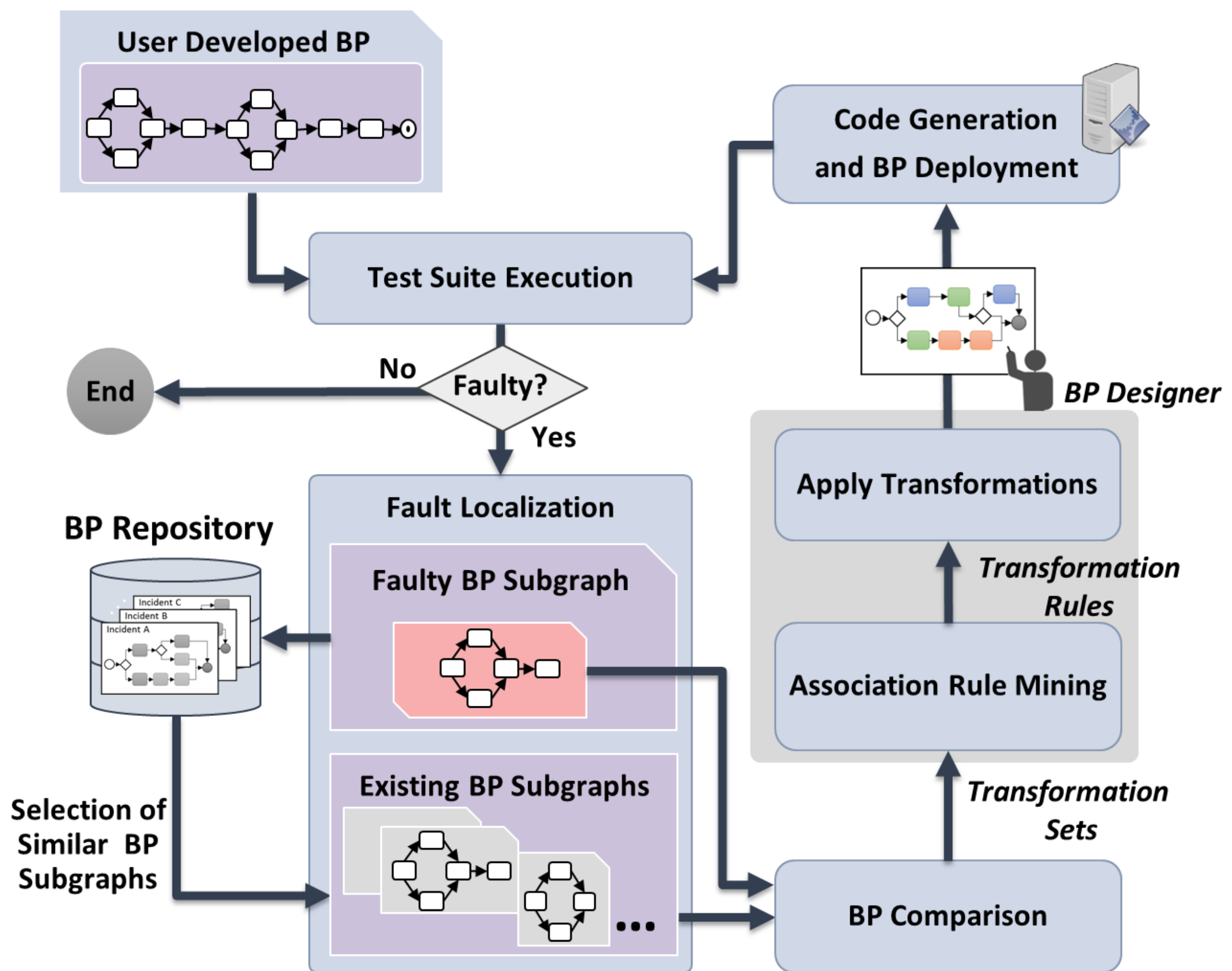


Figure 4.1: Collaborative BP fault resolution approach

Collaborative Fault Resolution (CFR) exploits the knowledge of existing BPs for the identification and resolution of faults in a faulty BP. Fig. 4.1 shows the different steps of the proposed approach. There are four main steps that are executed repeatedly until all faults are resolved (as per the execution on the test suite). If faults are detected, in the first step, fault localization is performed (discussed in Section 4.2.1) to identify one or more locations in the BP where the fault is observed. We refer to each of these locations as a *fault observation point* (fop). Note that an fop may not necessarily correspond to the source of the fault in the BP. The actual fault might lie at any location prior to the fault observation point. In the second step (discussed in Section 4.2.2), we perform a pairwise comparison of a faulty BP with existing BPs of other users to identify their structural and semantic differences with the faulty BP. In step 3, all of the pair-wise differences are holistically analyzed to compute BP transformation rules that modify the faulty BP by adding and/or removing some of its structural components. Our objective is to select the set of transformations that resolve the fault with minimal changes in the faulty BP. For this purpose, we develop a heuristic-based approach (discussed in Section 4.2.3) that employs association analysis over all the transformations to filter out unnecessary transformations. These transformations are then applied to the faulty BP. In step 4, the resulting BP is deployed so that the test cases can be re-executed to identify if faults remain.

Algorithm 4.1 outlines the steps of the proposed fault resolution approach. This algorithm is invoked for each fop. This algorithm tries to resolve the fault by iteratively expanding the search region considered backward from the given fop. Let S_f denote the subgraph of the faulty BP graph corresponding to the current search region. S_f is compared with the subgraphs of existing BPs that include a certain minimum degree of overlapping services with S_f. This comparison is performed in a pair-wise manner and computes the structural differences between S_f and the subgraph S_i of each existing BP that meets the overlapping service criterion. These differences essentially correspond to all the elements present in S_f but not in S_i and vice versa. Essentially, the pair-wise differences between S_f and S_i can be used to transform the subgraph S_f of the faulty BP to the subgraph S_i of an existing BP. Therefore, we refer to these structural differences as transformations.

ALGORITHM 4.1: Fault Resolution

Input: $G_f = (L_f, T_f, V_f, E_f, \mathcal{E}_f, v_f^{start}, v_f^{end}, v_f^{user})$ - faulty BP

Input: fop - fault observation point returned by fault localization procedure

Input: $\mathcal{B} = \{G_1, G_2, \ldots G_n\}$ - existing BPs

Input: α_0 - initial distance threshold

Input: δ - step size to increase the distance threshold

Output: G_c - Corrected BP that passes all the test cases

1: $\alpha \leftarrow \alpha_0$
2: $\mathcal{T} \leftarrow \Phi$
3: **while** $\alpha \leq distance(v_f^{start}, fop)$ **do**
4: $\quad S_f \leftarrow subgraphIncorrect(G_f, fop, \alpha)$
5: $\quad$ **for** each $G_i \in \mathcal{B}$ **do**
6: $\quad\quad S_i \leftarrow subgraphCorrect(G_i, S_f)$
7: $\quad\quad \mathcal{T}_i \leftarrow GraphComparison(S_f, S_i)$
8: $\quad\quad \mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}_i$
9: $\quad tRuleQ \leftarrow association_analysis(\mathcal{T}, conf, sup)$
10: $\quad$ **while** $tRuleQ \neq \phi$ **do**
11: $\quad\quad r \leftarrow dequeue(tRuleQ)$
12: $\quad\quad G_c \leftarrow applyTranformations(r, G_f)$
13: $\quad\quad$ **if** G_c exhibits no more faults in testing upto the given fop **then**
14: $\quad\quad\quad$ **return** G_c
15: $\quad \alpha \leftarrow \alpha + \delta$
16: **return** $NULL$

Depending on the size of the subgraph S_f, the entire faulty BP may be transformed into some existing BP. Since we assume that all existing BPs are correct and fault-free, transforming the faulty BP to any of the existing BP would remove the faults. However, this may change the scope or domain of the BP, for example, an e-commerce sales BP may get transformed into an insurance BP. Therefore, we need to identify a minimal set of transformations that removes the fault without changing the scope/domain of the faulty BP. As discussed above, we use an association analysis-

based approach that generates a set of transformation rules with changes that are common across multiple existing BPs. If the existing BPs are not limited to a single domain then considering the commonality of the transformation rule between a certain minimum number of BPs would decrease the likelihood of entirely changing the domain/scope of the BP since the transformed structure occurs across multiple different domains/scopes.

CFR may generate multiple transformation rules. We apply these transformation rules iteratively and run the test cases after each iteration to check if the resulting process passes all the test cases. In case the fault is not removed, we expand the search region and repeat. This process is depicted in Figure 4.1, and described below.

4.2.1 Fault Localization

Fault localization is the process of locating and isolating bugs or faulty software components to determine the likely causes of software failures or errors [154, 21]. For localizing faults in a BP, we use the statistical debugging technique of Liu et al. [29]. This technique considers predicate evaluation in both correct and incorrect executions of the software and considers a predicate to be fault-relevant if the evaluation pattern in the incorrect execution significantly diverges from the correct one. Each predicate is assigned a fault-relevance score and is ranked based on this score.

For fault localization in BPs, we define predicates characterizing service invocation, successful execution, or service failures for each service in the BP as well as for each branching condition. We evaluate the fault relevance score of each of these predicates using the execution logs of test cases. The location in the BP corresponding to the predicate with the highest fault relevance score is considered as *fault observation point (fop)*.

In order to efficiently locate and resolve the fault, we limit our search to the subgraph S_f, which includes the services and interconnections that precede the fop and it is parameterized by the distance threshold α.

Given a faulty BP, $G_f = (L_f, T_f, V_f, E_f, \mathcal{E}_f, v_f^{start}, v_f^{end}, v_f^{user})$ and a fault observation point fop, the subgraph $S_f = (V_{Sf}, E_{Sf})$ is computed as:

- $V_{Sf} = \{v \in V_f | distance(v, fop) \leq \alpha \wedge distance(v_f^{start}, v) < distance(v_f^{start}, fop)\}$, and

- $E_{Sf} = \{(u, v) | u, v \in V_{Sf}$ and $(u, v) \in E_f\}$

4.2.2 Comparison with Existing BPs

We compare the subgraph S_f of the faulty BP with all the subgraphs of existing BPs that include a certain minimum number of overlapping services with S_f. Let γ denote the threshold for the minimum number of overlapping services between S_f and an existing BP. For an existing BP $G_i = (V_i, S_i)$, its subgraph S_i is computed for comparison by considering the set of the common vertices between G_i and S_f. Let U_i denote this set, i.e., $U_i = \{u | u \in V_i \cap V_{Sf}\}$. G_i will be considered for comparison if $|U_i| \geq \gamma$.

Suppose $u_{min}, u_{max} \in U_i$ are nodes that are at a minimum and maximum distance from the starting node of the BP G_i, respectively. Then $S_i = (V_{Si}, E_{Si})$ can be computed as:

- $V_{Si} = \{v | v \in U_i \vee distance(v, u_{max}) \leq distance(u_{min}, u_{max})\}$, and
- $E_{Si} = \{(u, v) | u, v \in V_{Si}$ and $(u, v) \in E_i\}$.

We perform a pair-wise comparison between the subgraph S_f of the faulty BP and the relevant subgraph S_i of each of the existing BP. The steps for this pair-wise comparison between S_f and S_i are given in Algorithm 4.2. Specifically, this algorithm computes the difference between S_f and S_i in terms of the vertices, edges, and edge expressions. All those vertices and edges that are present in S_f but not in S_i are returned as sets V_i^r and E_i^r, respectively. Similarly, all those vertices and edges that are present in S_i but not in S_f are returned as set V_i^a and E_i^a, respectively. Note that an edge (u, v) that is present in both S_i and S_f, but the corresponding edge expressions are different in S_i and S_f is considered as a non-matching edge. This edge is placed in both E_i^r and E_i^a. In the E_i^r it is associated with the edge expression of S_f, and in the E_i^a it is associated with the edge expression of S_i.

As an example, consider a fragment of a faulty sales order BP G_f shown in the rectangular box of Figure 4.2. G_f contains two faulty data flow edges that are marked with $\times$. Specifically, the value of *tax_amount* is incorrectly assigned to *shipping* and the value of *ship_charges* is incorrectly assigned to *sales_tax*. The subgraph S_f of this faulty BP that needs to be compared with existing

ALGORITHM 4.2: Graph Comparison

Input: $S_f = (V_{s_f}, E_{s_f})$ - subgraph of faulty BP

Input: $S_i = (V_{s_i}, E_{s_i})$ - subgraph of correct BP

Output: $\mathcal{T}_i$ - the set of transformations required to convert S_f to S_i

1: $V_i^r \leftarrow \emptyset, V_i^a \leftarrow \emptyset, E_i^r \leftarrow \emptyset, E_i^a \leftarrow \emptyset$
2: **for** $\forall v \in V_{s_f}$ **do**
3:     **if** $v \notin V_{s_i}$ **then**
4:         $V_i^r \leftarrow V_i^r \cup v$
5: **for** $\forall v \in V_{s_i}$ **do**
6:     **if** $v \notin V_{s_f}$ **then**
7:         $V_i^a \leftarrow V_i^a \cup v$
8: **for** $\forall (u, v) \in E_{s_f}$ **do**
9:     **if** $((u, v) \notin E_{s_i})$ or (edge expressions of (u, v) in E_{s_f} and E_{s_i} do not match) **then**
10:         $E_i^r \leftarrow E_i^r \cup (u, v)$
11: **for** $\forall (u, v) \in E_{s_i}$ **do**
12:     **if** $((u, v) \notin E_{s_f})$ or (edge expressions of (u, v) in E_{s_f} and E_{s_i} do not match) **then**
13:         $E_i^a \leftarrow E_i^a \cup (u, v)$
14: $\mathcal{T}_i \leftarrow V_i^r \cup V_i^a \cup E_i^r \cup E_i^a$
15: **return** $\mathcal{T}_i$

BPs is depicted in the rectangular box in Fig. 4.2. This subgraph is compared with relevant subgraphs of three existing BPs that have at least two overlapping services with S_f as shown in the rectangular boxes in Figs. 4.3(a), 4.3(b), and 4.3(c). The results of this comparison are shown in Table 4.1. Each row in this table corresponds to a transformation operation. For example, the transformation $t_6 : (tax_amount, shipping)^-$ implies that the variable assignment from *tax_amount* to *shipping* needs to be removed from the faulty BP if it is to be transformed into BPs of G_1, G_2, or G_3. Similarly, the transformation $t_8 : (tax_amount, sales_tax)^+$ needs to be added.

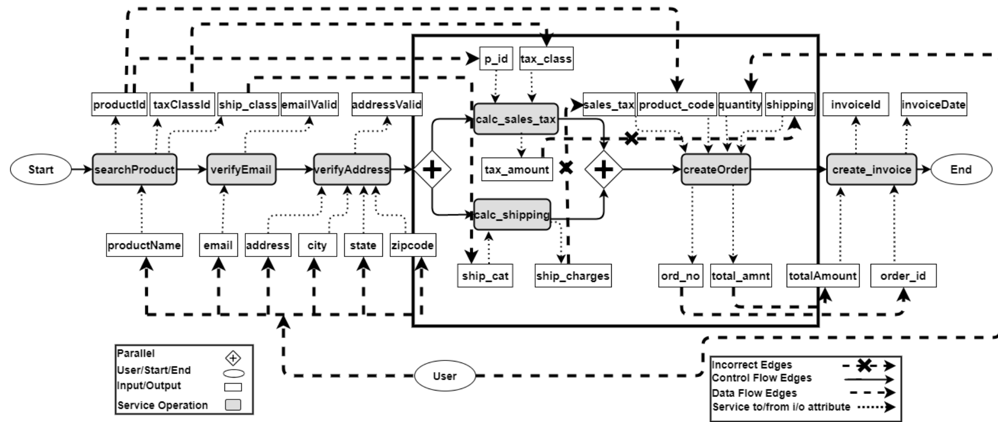
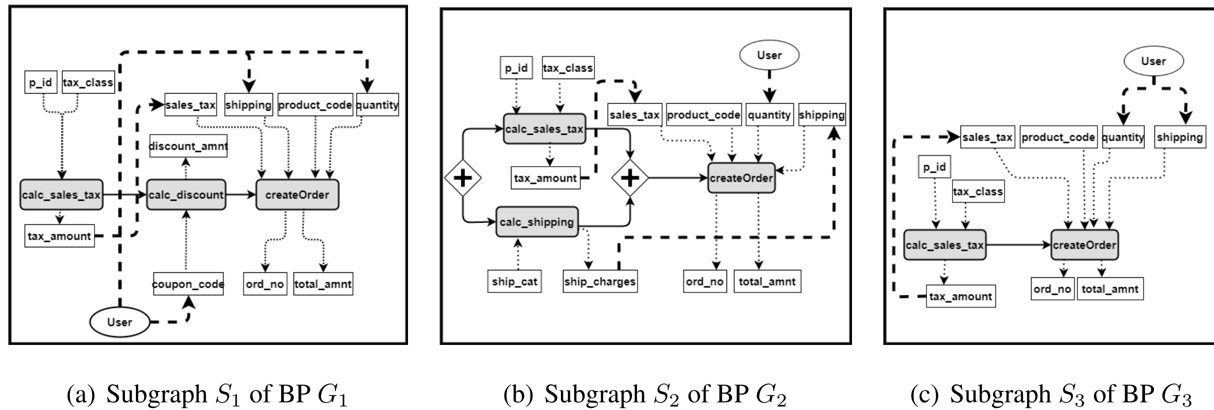


Figure 4.2: Faulty sales order BP (G_f) and its subgraph (shown in rectangular box) used for pair-wise comparison



(a) Subgraph S_1 of BP G_1 (b) Subgraph S_2 of BP G_2 (c) Subgraph S_3 of BP G_3

Figure 4.3: Subgraphs of existing BPs used for comparison with the faulty BP

4.2.3 Association Rule Mining on Transformations

As discussed above, graph comparison yields transformations between faulty BP G_f and existing BPs G_is. Since all the G_is are assumed to be correct, some of these transformations essentially correspond to the actual fault and its resolution. In other words, applying all the transformations within $\mathcal{T}_i$ changes the faulty BP G_f to G_i, thus fixing the fault but the scope and domain of the resulting BP may change as discussed in the introduction of Section 4.2.

The correct BPs may form groups based on their structural differences with respect to the faulty BP and therefore they have common or overlapping transformation sets. The differences

Table 4.1: Results of pair-wise graph comparison for the BPs depicted in Fig. 4.2 and Fig. 4.3

Symbol	Transformation	BP Graphs
t_1	$calc_dsicount^+$	G_1
t_2	$(User, coupon_code)^+$	G_1
t_3	$(andJoin, createOrder)^-$	G_1, G_3
t_4	$(calc_discount, createOrder)^+$	G_1
t_5	$(calc_sales_tax, calc_discount)^+$	G_1
t_6	$(tax_amount, shipping)^-$	G_1, G_2, G_3
t_7	$(ship_charges, sales_tax)^-$	G_1, G_2, G_3
t_8	$(tax_amount, sales_tax)^+$	G_1, G_2, G_3
t_9	$(ship_charges, shipping)^+$	G_2
t_{10}	$calc_shipping^-$	G_1, G_3
t_{11}	$(andSplit, calc_shipping)^-$	G_1, G_3
t_{12}	$(calc_shipping, andJoin)^-$	G_1, G_3
t_{13}	$(andSplit, calc_sales_tax)^-$	G_1, G_3
t_{14}	$(calc_sales_tax, andJoin)^-$	G_1, G_3
t_{15}	$(calc_sales_tax, createOrder)^+$	G_3
t_{16}	$(User, shipping)^+$	G_1, G_3
t_{17}	$andSplit^-$	G_1, G_3
t_{18}	$andJoin^-$	G_1, G_3

across these groups may range from the domains of the underlying BPs (characterized by the use of domain-specific web services not present in the faulty BP) to variations in the ordering of common web services. Our objective is to select the set of transformations that resolves the fault with minimal changes in the faulty BP. Note that we do not explicitly know the groups beforehand. However, if we can automatically identify the relationships/associations between sets of transformations, it may be possible to identify the groups and use this information to select a minimal set of transformations that can resolve the fault. Towards this, we first define the fault covering transformation set.

Definition 5. Fault covering transformation set. With respect to a given fop in a faulty BP (G_f), a fault covering transformation set ($\mathcal{F}_\mathcal{C}$) is a minimal set of transformations that resolves all faults in G_f up to the given fop. In other words, after applying all the transformations in the set $\mathcal{F}_\mathcal{C}$ to G_f, the resulting BP passes all the test cases that validate conditions up to the given fop.

CFR employs association rule mining to first identify the relationship/associations between

the transformations across different groups of BPs with respect to the faulty BP. The resulting association rules are then used to systematically discover a minimal set of transformations that contains the fault-covering transformation set.

An association rule is an implication of the form $X \implies Y$, where X and Y are disjoint sets. In our context, if the faulty BP differs with some group of BPs in terms of the transformations in the antecedent set X, then the faulty BP also differs with the same group in terms of the transformations in the consequent set Y.

The following theorem establishes an important result for identifying the fault covering transformation set based on the implication relationship between the transformation sets in association rules.

Theorem 1. Given a faulty (but structurally valid) BP G_f and a set of correct and structurally valid BPs G_i, if $X \implies Y$ is an association rule with 100% confidence, but $Y \implies X$ is not, then for any fault covering transformation set $\mathcal{F}_\mathcal{C}$ of G_f, if $X \cup Y$ contains $\mathcal{F}_\mathcal{C}$ then X cannot contain $\mathcal{F}_\mathcal{C}$, i.e., $\mathcal{F}_\mathcal{C} \subseteq X \cup Y \implies \mathcal{F}_\mathcal{C} \nsubseteq X$.

Proof: As discussed in the Introduction, we consider faults of types branching faults, data flow faults (variable assignment), control flow faults, and expression faults, listed in Table 1.1. The faulty BP, G_f, may include any of these fault types or their combination.

Since the faulty BP, G_f is structurally valid (i.e., all service operation vertices, control flow vertices, and attribute vertices in G_f have valid predecessors and successors – Definition 3), the fault is manifested in one or more vertices or edges in G_f. This implies that the edges or vertices corresponding to the fault are present in G_f, but absent in any correct BP. Note that the absence of a vertex or edge in G_f, which is present in some correct BP G_i, may also correspond to a fault. However, the transformation to add such vertex or edge in G_f in order to fix the fault requires removing at least one edge from G_f because it is structurally valid. For example, if the vertex to be added corresponds to some service operation or control flow element, a control flow edge needs to be removed from G_f and new edge(s) from/to appropriate predecessor/successor vertices need to be added in G_f. Similarly, if the vertex to be added is of the type attribute, some data flow edge needs to be removed from G_f and new data flow edges need to be added to keep the resulting BP

structurally valid. Also, if a fault corresponds to some control flow or data flow edge that is present in G_i but not in G_f, the addition of such edge requires the removal of some other edge from G_f for the same reason.

Based on the above discussion and following the graph comparison algorithm (Algorithm 4.2), we can deduce that:

$$\bigcap_i (V_i^r \cup E_i^r) \neq \phi \tag{4.1}$$

Where, V_i^r and E_i^r denote the set of vertices and edges that are present in G_f but not in G_i, respectively.

Equation (4.1) implies that the intersection of all $\mathcal{T}_i$s will not be a null set, i.e.,

$$\bigcap_i (V_i^r \cup E_i^r) \subseteq \bigcap_i \mathcal{T}_i \neq \phi \tag{4.2}$$

Since all the $G_i s$ are correct, we can prove that any minimal fault covering set $\mathcal{F}_\mathcal{C}$ contains all the transformations that are common across all $\mathcal{T}_i$s, i.e.,

$$\bigcap_i \mathcal{T}_i \subseteq \mathcal{F}_\mathcal{C} \tag{4.3}$$

For any association rule $X \implies Y$ with 100% confidence, $Y \implies X$ does not also hold with 100% confidence $\iff$ there is some correct BP G_j such that $Y \subseteq \mathcal{T}_j$ and $X \nsubseteq \mathcal{T}_j$.

Based on this and the fact that $\mathcal{F}_\mathcal{C} \subseteq X \cup Y$, we can show that:

$$\bigcap_i \mathcal{T}_i \cap Y \neq \phi \tag{4.4}$$

and

$$\bigcap_i \mathcal{T}_i \cap Y \subseteq \mathcal{F}_\mathcal{C} \tag{4.5}$$

and

Considering (4.3), (4.4), (4.5) and given that $\mathcal{F}_\mathcal{C} \subseteq X \cup Y$ and $X \cap Y = \phi$, we can deduce that $\mathcal{F}_\mathcal{C} \nsubseteq X$. $\square$

Based on the above theorem, if we have identified two transformation sets X and Y such that applying all the transformations in those sets to a faulty BP removes its fault and an association relationship exists between X and Y, then in order to find the minimal transformation set that contains $\mathcal{F}_C$, we should start with the transformation set Y first. If applying transformations in Y does not remove the fault, then we should consider the transformation sets X and Y jointly, but not X separately.

One simple approach that can be developed to detect and fix faults is to find all the association rules considering the faulty BP and correct BPs and then go through the appropriate X and Y that jointly contain the fault covering transformation set and result in minimal transformations to the faulty BP. However, we may need to analyze a large number of association rules given the exponential number of rules that can be generated from a given itemset. CFR discovers and searches the association rules for the fault-covering transformation set in a systematic and efficient manner. The specific steps of CFR are listed in Algorithm 4.3: Association Analysis. As shown in this algorithm, we use *apriori* algorithm (line 2) to find association rules of length 2, i.e., both antecedent and consequent are single items in the discovered rules. The reason that we start with rules of length 2 is to reduce the number of association rules. From the resulting set of association rules, we generate a directed graph $G_{ar} = (V_{ar}, E_{ar})$, where a vertex in V_{ar} is either a consequent or an antecedent of some association rule (line 3). The edges in G_{ar} represent the antecedent $\rightarrow$ consequent relationship. Next, we find all strongly connected components in G_{ar}. A strongly connected component (SCC) in the graph G_{ar} essentially represents a non-trivial maximal length antecedent or consequent of some valid association rule that can be computed with the given transformation dataset and support/confidence thresholds. Fig. 4.4 shows a graph with 5 SCCs. This graph is generated using the transformations listed in Table 4.1 and running association rules with minimum support = 33% and confidence = 100%. The edge between the SCCs in the graph represents a non-trivial maximal length association rule. For example in Fig. 4.4 the edge from the SCC C_3 to SCC C_1 represent the association rule $C_3 : \{t_9\} \implies C_1 : \{t_6, t_7, t_8\}$.

The SCC can be categorized as the source, sink, or internal. For example in Figure 4.4, C_3, C_4, and C_5 are source SCCs, C_1 is sink SCC, and C_2 is internal SCC. From this SCC graph, we find the

ALGORITHM 4.3: Association Analysis

Input: $\mathcal{T} = \bigcup \mathcal{T}_i$, where $\mathcal{T}_i$ is the set of transformations required to convert S_f to S_i

Input: Specified Confidence $conf$ and Support sup for Association Analysis

Output: $tRuleQ$ - list of transformation rules sorted in path length order

1: Create a binary matrix $M_{|\mathcal{T}|\times n}$ where $M_{ki} = 1$ if $t_k \in \mathcal{T}_i$, and $M_{ki} = 0$ otherwise
2: $rules \leftarrow Apriori(M, sup, conf, length = 2)$
3: Construct the directed graph $G_{ar} = (V_{ar}, E_{ar})$ where vertices in V_{ar} correspond to sets of transformations that are either a consequent or antecedent for some rule, and edges in E_{ar} represent the antecedent $\longrightarrow$ consequent relationship.
4: Find all strongly connected components in the graph G_{ar}
5: $tRuleQ \leftarrow \phi$
6: **for** all components SCC_i that are not connected to any other component SCC_j **do**
7:     $tlist \leftarrow$ the set of all transformations in SCC_i
8:     $tRuleQ \leftarrow tRuleQ \cup tlist$
9: **for** each source component SCC_{src} **do**
10:     **for** each sink component SCC_{snk} **do**
11:         **if** a path exists from SCC_{src} to SCC_{snk} **then**
12:             Find the longest path l from SCC_{src} to SCC_{snk}
13:             $tlist \leftarrow$ the set of all transformations in l
14:             $tRuleQ \leftarrow tRuleQ \cup tlist$
15: Sort tRuleQ by ascending order of length
16: **return** $tRuleQ$

longest path between each source and sink SCC pairs (lines 6 - 14). In case the SCC is both source and sink (i.e., not connected to any other SCC), the path includes only that SCC. The reason for considering the longest path is to encompass all the association rules that can be computed with the given transformation dataset and support/confidence thresholds.

As discussed above, we search for the fault covering transformation set in each of these paths. For example, for the path $C_4 \implies C_2 \implies C_1$ in Fig. 4.4, we first look for the fault covering set in the sink SCC C_1 by applying all the transformations in C_1 to the faulty BP. If the fault is not

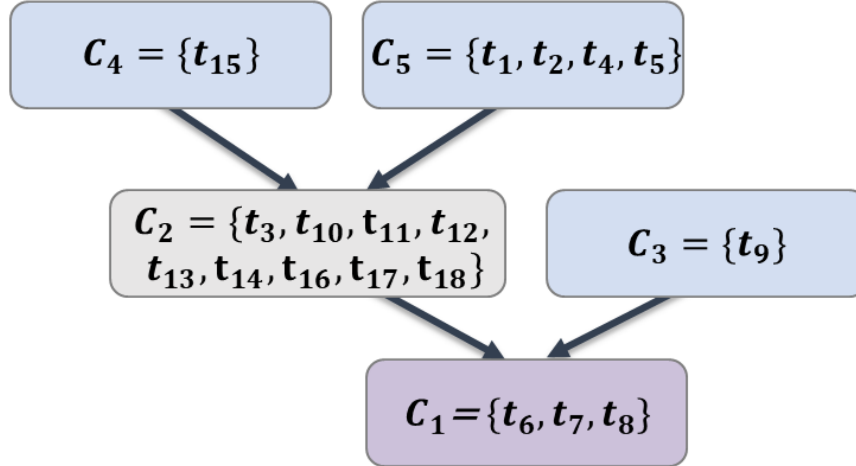


Figure 4.4: SCC graph resulting from association analysis on transformations listed in Table 4.1

resolved, we add C_2 to this search and finally C_4. To keep the number of structural changes to the faulty BP to a minimum, we sort all these paths in the ascending order of length (i.e., number of transformations and path length) and preferentially apply the smaller length transformations to the faulty BP (line 15 of Algorithm 4.3 and lines 10 - 14 of Algorithm 4.1).

The path ($C_4 \implies C_2 \implies C_1$) in Fig. 4.4 transforms the subgraph of faulty BP S_f into the subgraph S_3 of existing BP G_3 depicted in Fig. 4.3(c). The second path ($C_5 \implies C_2 \implies C_1$) resolves the faults by transforming S_f to S_1 depicted in Fig. 4.3(a). The last path ($C_3 \implies C_1$) applies minimum transformations by removing two incorrect edges $\{t_6 : (tax_amount, shipping)^-, t_7 : (ship_charges, sales_tax)^-\}$ and adding two replacement edges $\{t_8 : (tax_amount, sales_tax)^+, t_9 : (ship_charges, shipping)^+\}$ to convert S_f into S_2 depicted in Fig. 4.3(b), thus resolving G_f with minimal changes.

4.2.4 Computation complexity

Graph Comparison (Algorithm 4.2). This algorithm compares the faulty BP graph and an existing BP graph to compute the transformation set. Therefore, its complexity is linear in the size of the input graphs, i.e., $O(|V| + |E|)$.

Association Analysis (Algorithm 4.3). On line 2 of this algorithm, *Apriori association rule mining* is called to compute association rules of length 2. These association rules are computed

over $\mathcal{T} = \bigcup \mathcal{T}_i$, where $|\mathcal{T}_i| = O(|V| + |E|)$. Therefore, $|\mathcal{T}| = O(n(|V| + |E|))$, where n is the number of existing BPs used for comparison. Let $m = n(|V| + |E|)$. We can have a maximum of $2 \times \binom{m}{2} = m(m-1)$ rules of length 2. In line 3 graph $G_{ar} = (V_{ar}, E_{ar})$ is constructed from the resulting association rules, where $|V_{ar}| \leq m$ and $|E_{ar}| \leq m(m-1)$. On line 4, we compute strongly connected components (SCCs) in G_{ar} with a computational complexity of $O(|V_{ar}| + |E_{ar}|) = O(m^2)$. In the worst case the number of strongly connected components in G_{ar} is m. Let $G_{SCC} = (V_{SCC}, E_{SCC})$ denote the SCC graph of G_{ar}. In lines 9-14, the longest path from each SCC_{src} to SCC_{snk} pair is computed. Since G_{SCC} is a directed acyclic graph, the computation complexity of finding the longest path between any SCC_{src} to SCC_{snk} pair is $O(|V_{SCC}| + |E_{SCC}|) = O(m^2)$. Therefore, the computation complexity of Algorithm 4.3 is $O(m^2) = O(n^2(|V| + |E|)^2)$.

Fault Resolution (Algorithm 4.1). The algorithm calls Algorithm 4.2 n times and Algorithm 4.3 once in each iteration of the search region parameterized by α. The search region is incrementally expanded until the faulty BP is fixed or the entire faulty BP is covered. Therefore, the overall computation complexity of the fault resolution algorithm for a fixed search region is $O(n^2(|V| + |E|)^2)$.

4.3 Hybrid Approach for Fault Resolution

The EGV approach for fault localization examines a BP in isolation therefore it lacks the capability to identify any control flow and branching faults that occur due to any activity/element removal in the BP e.g., a missing service operation or a branch path in an XOR block. CFR is a collaborative fault resolution (CFR) approach [157] that is capable of resolving such faults.

CFR relies on a set of existing BPs that are composed of similar services and are assumed to be correct. The knowledge of these BPs is utilized to discover and fix faults in a faulty BP. However, CFR requires certain minimum overlapping services between the faulty BP and existing BPs for providing accurate results. Rather than examining the faulty BP in isolation (as EGV does), we propose a hybrid of EGV and CFR approaches that allow for broader coverage of fault types by leveraging knowledge from existing BPs.

The hybrid approach combines both EGV and CFR in an interleaved fashion. We consider two versions of the hybrid approach:

H1: EGV is executed first to resolve faults for a given fop. If EGV fails, only then CFR is invoked for that fop. This interleaved execution continues until either the BP is fixed or there are no more candidate fixes to apply.

H2: CFR is executed first to resolve faults for a given fop. If CFR fails, only then EGV is executed for that fop. Similar to H1, H2 is executed in an interleaved manner.

Algorithm 4.4 outlines the hybrid fault resolution approach, H1. The inputs to this algorithm are a faulty BP graph, G_f, a set of test cases, T, and the set of existing BPs, $\mathcal{B}$. H1 first calls fault localization to find the fop (Line 1). Next, a subgraph of G_f from v_f^{start} to fop is extracted for generating candidate fixes (Line 2). EGV is called on this subgraph for the resolution of faults up to the fop (Line 3). If a candidate fix (G_c) is found for the subgraph S, it is combined with the remaining part of G_f after the fop (Line 5). If G_c not only fixes the fault up to the given fop but also of the entire BP then the resulting BP is returned to the user as a fault-free BP Lines (6 – 8). Otherwise, if the candidate fix partially fixes the fault (i.e., it passes some, but not all, of the test cases that originally failed), we use that candidate fix for further discovery and repair of faults by recursively calling H1 (Line 10). In case no partial fix is found, we call CFR for collaborative fault resolution (Line 12). Similar to the steps given in lines 4 – 10, we repeat these steps for CFR in lines 11 – 19. We recursively call H1 fault resolution procedure until all the faults are fixed or the entire space of candidate fixes is exhausted. The algorithm of hybrid fault resolution approach H2 is similar to Algorithm 4.4 except that it invokes CFR first followed by EGV.

4.4 Experimental Evaluation (CFR)

We have performed an extensive experimental evaluation of CFR. Two independent strategies were followed to evaluate the performance of our approach. In the first case, we randomly injected faults (of different types) into a correct BP and then employed our approach to resolving the faults injected. In the other case, we asked actual users to develop BPs using a BP composition tool. We

ALGORITHM 4.4: Hybrid_H1

Input: $G_f = (V_f, E_f, \mathcal{E}_f, v_f^{start}, v_f^{end}, v_f^{user})$ - Faulty BP

Input: T - Set of test cases

Input: $\mathcal{B} = \{G_1, G_2, \ldots G_n\}$ - Set of existing BPs

Output: G_c - Corrected BP

1: $fop \leftarrow localize_fault(G_f)$
2: $S \leftarrow subgraph_till_fop(G_f, fop)$
3: $G_c \leftarrow$ EGV_FaultResolution(S, T)
4: **if** $G_c \neq NULL$ **then**
5: $\quad G_c \leftarrow combine_graph(G_f, S, G_c)$
6: $\quad$ Apply each test case $t \in T$ on G_c
7: $\quad$ **if** G_c passes all the test cases **then**
8: $\quad\quad$ **return** G_c
9: $\quad$ **else**
10: $\quad\quad$ **return** Hybrid_H1(G_c, T)
11: **else**
12: $\quad G_c \leftarrow CFR(S, T, \mathcal{B})$
13: $\quad$ **if** $G_c \neq NULL$ **then**
14: $\quad\quad G_c \leftarrow combine_graph(G_f, S, G_c)$
15: $\quad\quad$ Apply each test case $t \in T$ on G_c
16: $\quad\quad$ **if** G_c passes all the test cases **then**
17: $\quad\quad\quad$ **return** G_c
18: $\quad\quad$ **else**
19: $\quad\quad\quad$ **return** Hybrid_H1(G_c, T)
20: $\quad$ **else**
21: $\quad\quad$ **return** NULL

then examined only the BPs that were faulty and tried to resolve the faults using our approach. As such, the first case can be considered equivalent to evaluation with synthetic data, while the second can be considered equivalent to evaluation with real data. In both cases, a repository of 48

existing BPs was used (24 BPs from flight reservation and 12 each from insurance and e-commerce sales domains). All these BPs are derived from available open-source insurance systems (e.g., OpenUnderWriter, Open Insurance, etc.), enterprise resource planning (ERP) systems (e.g., Odoo, Apache OFBiz, inoERP, and Tryton, etc.), and online flight reservation systems (e.g., Pakistan International Airline, Qatar Airways, and Emirates Airline, etc.). We generated the BPs from the execution logs of the installed ERP and insurance systems or from the reference documentation. For flight reservation systems, we created the BPs from the workflow structure derived from the websites. A similar BP dataset collection methodology was used in our prior work and is described in detail in [2]. Table 4.2 shows the detailed statistics of the developed BPs.

Table 4.2: Statistics of existing BPs

Domain	No. of BPs	Avg. no. of services	Avg. no. of branches
E-commerce	12	25.66	2.91
Insurance	12	26.00	3.00
Flight Reservation	24	22.00	3.00
Total	48	23.91	2.97

The experiments were performed on an Intel Xeon server machine with 24 2.3 GHz cores running Ubuntu Linux 16.04 with an overall memory of 256 GB. Note, however, no parallelization was used (i.e., only one core was used and no significant amount of memory was used).

4.4.1 Random Fault Injection

As mentioned above, in this case, we started with a correct BP from the insurance domain (not included in the existing BP repository). The selected BP was composed of 26 Web service operations with 3 branches and 2 parallel structures. To generate faulty BPs, we applied random combinations of the mutation operators listed in Table 1.1. This resulted in 208 faulty BPs that were used for validation. The number of mutation operators applied to generate a faulty BP varied between 1

Table 4.3: α vs. average number of rules

α	# of rules	α	# of rules	α	# of rules
2	88.95	7	1828.50	12	4881.55
3	186.54	8	2562.11	13	6365.96
4	362.34	9	2965.20	14	10593.71
5	596.85	10	3666.97	15	13878.29
6	843.72	11	3760.26	16	15499.64

and 4 with a mean of 2.24 and a standard deviation of 0.81. Faults from each fault category and their random combinations were tested.

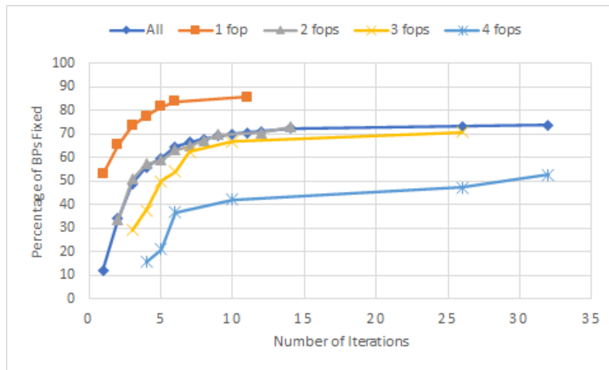
We created a suite of test cases based on the expected output covering all branching paths of the correct BP. Now, a BP is considered as correct if it passes all the test cases in the suite. We applied fault localization on each faulty BP to determine their fop(s). The number of fop(s) varied between 1 and 4 for the faulty BPs.

After determining the fops, we ran Algorithm 4.1 on each faulty BP. The average number of rules returned by the algorithm depends upon the value of α. The average number of rules for different values of α are given in Table 4.3. We considered those rules that produce structurally valid BP after their application to the faulty BPs. We sorted these rules in ascending order of their size and applied them one by one to the faulty BP. After the application of each rule, we tested the resulting BP using the test suite created earlier. If the resulting BP passed all the test cases, we marked it as correct and stopped. Note that a rule may resolve a fault specific to the given fop, but there could be more faults that are manifested later during BP execution. Therefore, in our experimental evaluation, we clipped the BP up to the given fop and run the relevant test cases to check if the BP is fixed with respect to the given fop. Only after all fop-specific test cases are passed, we run the complete suite of test cases to look for any further faults. If any of the test cases failed, we reinvoked the fault localization procedure to find a new fop and repeat the entire process. The number of transformations varied from 2 to 32 with a mean of 10.95 and a standard deviation of 6.07.

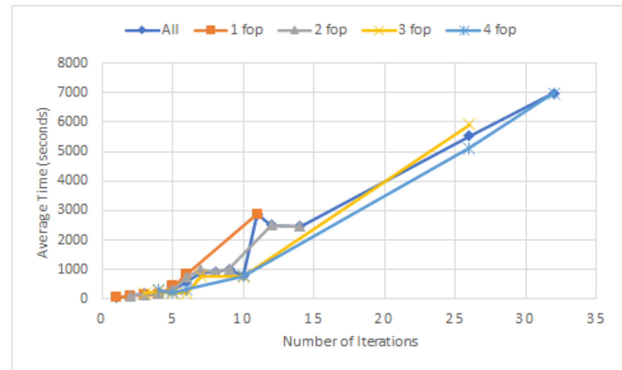
Table 4.4 shows the fop-wise distribution of the faulty BPs along with the accuracy of the proposed fault resolution approach. As the results show, we were able to fix faults in 73.83% of all the BPs. Note that the accuracy decreases as the number of fops increases. This is due to the fact that the faults corresponding to fops that are further apart are likely to be independent of one another.

Table 4.4: Accuracy results over synthetic dataset

No. of $fops$	Total	Fixed	Failed	Accuracy
1	46	39	7	0.85
2	119	87	32	0.73
3	24	17	7	0.71
4	19	10	9	0.53
Total	208	153	55	0.74



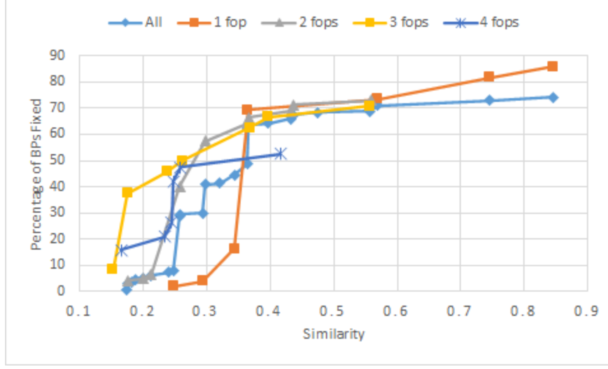
(a) Percentage of BPs fixed vs. number of iterations



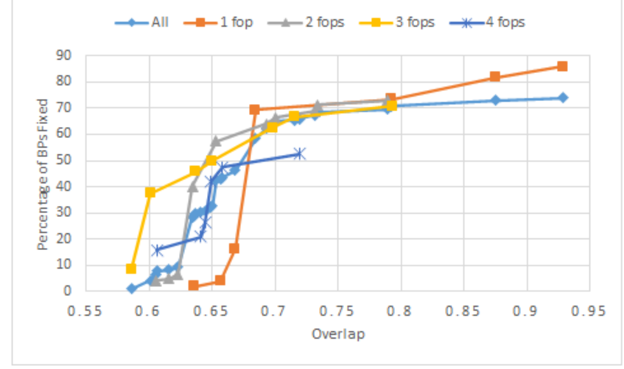
(b) Average time taken to fix a BP vs. number of iterations

Figure 4.5: Comparison between the percentage of BPs fixed vs. number of iterations and the iteration-wise average time taken for fault resolution per BP

In terms of the computation time overhead associated with the proposed fault resolution approach, Fig. 4.5(a) shows the percentage of BPs fixed vs. the number of iterations. Overall, 65% of the total BPs were fixed in 7 or fewer iterations. The BPs with 1 fop were all fixed in 11 or fewer iterations. The BPs with 2 fops took at most 14 iterations to complete. However, BPs with 3 and



(a) Percentage of fixed BPs vs. similarity



(b) Percentage of fixed BPs vs. service overlap

Figure 4.6: Comparison between the percentage of BPs fixed vs. similarity and service overlap based on top 3 rules

4 $fops$ took more iterations with a maximum of 26 and 32 iterations, respectively. The increase in the number of iterations with the corresponding increase in the number of $fops$ is due to the fact that we resolve the faults incrementally as explained above.

Fig. 4.5(b) shows the iteration-wise average time for fault resolution per BP. As depicted in this figure, the average time taken to fix a BP increases linearly in the iteration intervals [1 - 5], [6 - 10], and [10 - 15]. Moreover, the slope of this linear trend also increases across these iteration intervals. The reason is that in each iteration, α is increased which expands the search region in the faulty BP, therefore the computation time of association mining rules increases. Based on Fig. 4.5(a) and Fig. 4.5(b), the faults of the 50% of the total 153 BPs (that were fixed) were resolved in 3 or fewer iterations. Therefore, the median average time taken to resolve faults in a BP is 166 seconds. We also evaluated the impact of structural similarity and service overlap between the faulty BP and correct BPs on fault resolution. Both similarity and service overlap was computed w.r.t. the correct BPs that were part of the top 3 rules obtained after applying the association rule mining step of Section 4.2.3. Let $M \subseteq \mathcal{B}$ denote the set of correct BPs that are part of *top 3* rules. As discussed above, $\mathcal{T}_i$ denotes the set of transformations required to convert a faulty BP, $G_f = (V_f, E_f)$ into an existing BP $G_i = (V_i, E_i)$. The structural similarity between a faulty BP and BPs in M can be computed as the average of the number of transformations required to transform

G_f to G_i, normalized for each G_i:

$$Similarity = 1 - \left(\frac{1}{|M|} \sum_{G_i \in M} \frac{|\mathcal{T}_i|}{\max(|E_f| + |V_f|, |E_i| + |V_i|)} \right) \tag{4.6}$$

Similarly, the service overlap can be computed as the average of the number of common services between the faulty BP G_f and each $G_i \in M$, normalized for each G_i:

$$Overlap = \frac{1}{|M|} \sum_{G_i \in M} \frac{|V_f \cap V_i|}{\max(|V_f|, |V_i|)} \tag{4.7}$$

As per these definitions, it is highly unlikely to have a high service overlap but low similarity. Fig. 4.6(a) and Fig. 4.6(b) show the number of BPs fixed vs. similarity and service overlap, respectively. 68% of all the BPs were fixed at an average similarity of 0.43 or less. Moreover, less than 10% of the BPs were fixed when the service overlap was less than 0.62. From these results, we can infer that the likelihood of resolving faults in a BP increases when the service overlap is 0.7 or more and similarity is 0.43 or more.

Comparison with EGV

For comparison of EGV with CFR, we performed experiments on 112 BPs out of 208 that do not contain faults related to branch removal (AIE), activity removal (AEL), or constant modification (ECN). EGV is not designed for resolving fault types in which one or more BP elements are removed. Table 4.5 shows the accuracy results of EGV in comparison to CFR approach.

The results show that EGV has a higher accuracy than CFR on BPs with 1 and 2 $fops$ and lower accuracy with 3 and 4 $fops$. Overall, EGV has an accuracy of 0.83 while CFR has an accuracy of 0.76. Apart from the gain in accuracy, EGV has the advantage of being able to resolve the faults without relying on existing BPs that have overlapping services with respect to the faulty BP.

Fig. 4.7 shows that EGV takes less time than CFR for resolving faults in BPs with 3 or less $fops$ while CFR performs slightly better with 4 $fops$. This is because the number of mutants increases significantly with the increase in $fops$. Additionally, the time taken by CFR to resolve faults decreases from 2 to 3 and 4 $fops$. This reduction can be attributed to the decrease in the number of candidate fixes for BPs with 3 and 4 $fops$, as illustrated in Table 4.5. This decrease

occurs because we exclude candidate fixes that do not result in structurally valid BPs. Moreover, for this experiment, we excluded BPs containing faults related to activity removal, branch removal, and constant modification. Consequently, we were left with only 10 and 9 BPs with 3 and 4 $fops$, respectively, for this experiment. It is worth noting that the accuracy of CFR drops for BPs with 4 $fops$, indicating that only BPs containing simpler faults were fixed. As a result, CFR exhibited a significantly lower average time for resolving faults in BPs with 4 $fops$. Whereas, the time reduction from 2 to 3 $fops$ is not as significant and solely results from the lower number of structurally valid candidate fixes.

Although EGV outperforms CFR in terms of computation time with comparable accuracy yet CFR provides a broader fault coverage. For example, CFR can resolve faults belonging to branches and activity removal. These faults cannot be resolved by the basic G&V and EGV. However, CFR requires a repository of existing BPs.

Table 4.5: Accuracy of EGV and CFR.

$fops$	No. of BPs	Accuracy		Candidate Fixes	
		EGV	CFR	EGV	CFR
1	30	0.93	0.90	7.39	13.32
2	63	0.85	0.71	11.59	35.46
3	10	0.7	0.8	24.85	32.57
4	9	0.44	0.67	19.25	29.5
Overall	112	0.83	0.76	11.65	28.32

Comparison with basic G&V

We would also like to establish the effectiveness of our CFR with respect to an existing baseline. However, there is no existing solution for BP fault resolution that we can directly compare with. Therefore, we compare the proposed collaborative BP fault resolution approach with the Generate-and-Validate (G&V) automated program repair methodology. G&V takes as input a faulty program and a group of passing and failing tests and heuristically searches the program space to generate

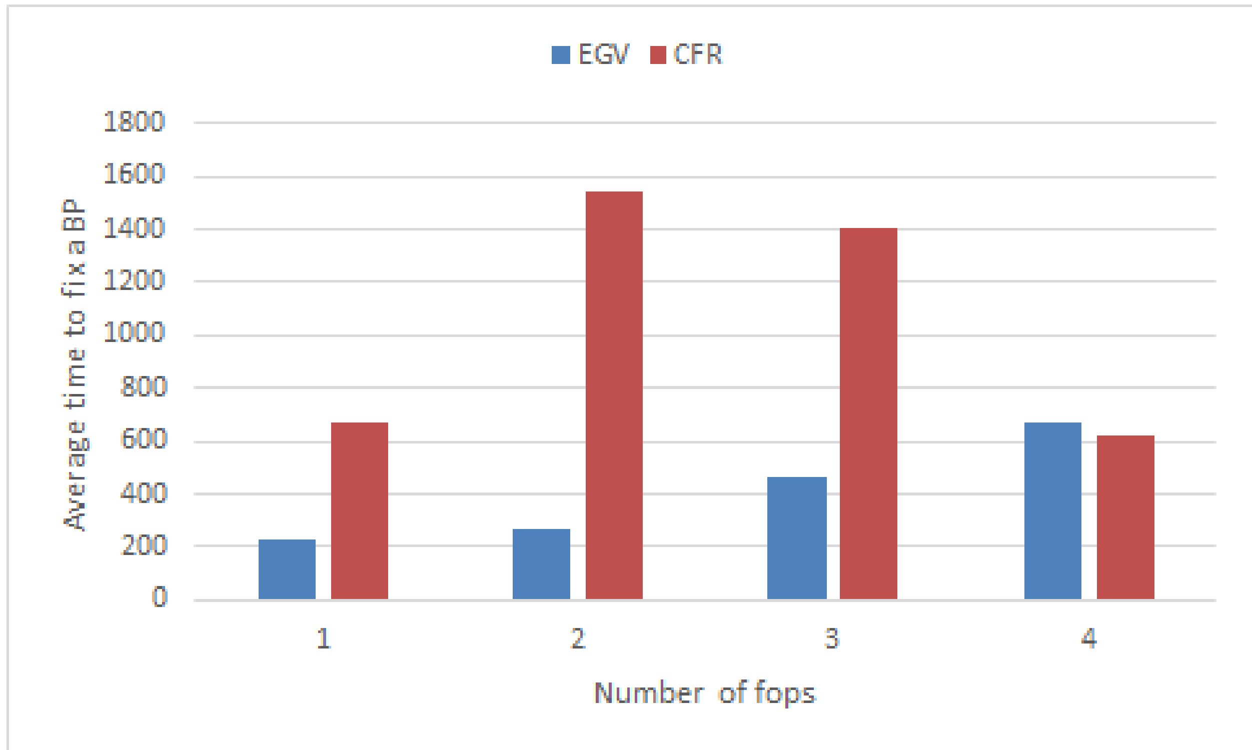


Figure 4.7: Execution time comparison of EGV and CFR.

Table 4.6: Accuracy comparison of CFR with G&V fault repair approach

Fault Category	No. of BPs	G&V with BPELswice		CFR	
		Accuracy	Avg. no. of candidate fixes applied for fault resolution	Accuracy	Avg. no. of rules (candidate fixes) applied for fault resolution
Variable assignment faults	24	20/24=0.83	2192	12/24=0.5	57
Expression faults	5	5/5 = 1.0	464	5/5 = 1.0	5
Control flow faults (excl. element removal)	11	11/11=1.0	9351	9/11=0.81	16
Control flow faults (incl. element removal)	24	-	-	11/24=0.45	41
Branching faults (incl. element removal)	3	-	-	3/3 = 1.0	9
Multiple faults in different categories	141	-	-	113/141=0.8	38

candidate fixes. The validity of the candidate fixes is then checked by running all available tests [18]. G&V employs fault localization to identify suspicious code blocks that may contain the fault. The candidate fixes are generated by considering different mutations of the statements within the

suspicious code blocks. Currently, there is no implementation of G&V automated program repair for BPEL programs, although there are several implementations available for Java and C Programs [18].

For comparison, we implemented the G&V automated program repair approach for BPEL programs by employing BPELswice fault localization technique [61] to identify suspicious BPEL statements. For generating the candidate fixes, we applied the mutation operators (listed in Table 1.1) to the suspicious statements.

As mentioned in [61], BPELswice is not designed for faults caused by the removal of activities/elements. Therefore, we consider only those fault categories that are relevant to BPELswice. This comparison is shown in the first three rows of Table 4.6. For variable assignment and control flow fault categories (excluding activity/element removal), G&V achieves higher accuracy than CFR. For expression faults both G&V and CFR resolve all the faults. However, the average number of candidate fixes applied for G&V is orders of magnitude higher than the average number of rules (candidate fixes) applied by CFR for all three categories. This clearly shows the efficiency of CFR over G&V.

Note that the higher accuracy of baseline (G&V) is expected since the faulty BPS are created by applying mutation operators on a correct BP. For instance, if the suspicious code block includes all the BPEL statements then we can exhaustively generate all mutants of the faulty BP and at least one of these mutants will be correct. Therefore, the baseline approach would be able to resolve the fault by exhaustively searching all possible mutants of the faulty BP with a very high computation time overhead. On the other hand, CFR resolves faults by applying very few transformation rules (candidate fixes).

The last three rows of Table 4.6 show the accuracy results of CFR for those fault categories that cannot be resolved by G&V + BPELswice. These include control flow and branching faults caused by activity/element removal as well as a combination of multiple faults from different categories. CFR achieves high accuracy in resolving branching and multiple faults. Overall, out of the 208 faulty BPs, CFR was able to fix 153 BPs resulting in an accuracy of 0.73.

These results clearly show that CFR is highly efficient and effective with respect to the number

of transformation rules (candidate fixes) applied for resolving BP faults. Moreover, it provides a broader coverage of fault categories as compared to the baseline. However, we note that the accuracy of CFR w.r.t. variable assignment faults and control flow faults (caused by activity removal) is low as compared to other fault categories. Variable assignment faults are typically manifested at multiple locations in the BP. For example, an incorrect variable assignment in an assignment block can result in subsequent incorrect variable assignments. Control flow faults are difficult to detect and resolve due to the varying control flow structure of activities in existing BPs. For example, one BP may compose independent activities in a sequence, while another may compose them in a parallel flow, thus, we may not find sufficient BPs in the repository for comparison.

4.4.2 User Developed BPs

For this evaluation, we applied CFR on BPs that were developed by actual users. These users were students of a graduate class (Service-oriented Computing, CS-585), who developed BPEL processes using the ASSEMBLE tool [2] as part of their class assignment.

These BPs were related to e-commerce sales, insurance sales, and flight reservations. We selected those BPs that did not execute correctly. The users were not aware that their developed BPs would be used for the evaluation of the fault resolution approach. Thus, there was no additional incentive to either increase or decrease the number of faults in any way beyond the normal goal of developing a correct BP. Overall, there were 4 e-commerce sales BPs, 5 insurance sales BPs, and 6 flight reservation BPs that were used for the evaluation. We were able to fix 12 out of the 15 BPs. Table 4.7 lists all user-developed BPs, with the type of faults that were made by the users and the results of CFR.

Note that the accuracy of our approach is higher for user-developed BPs as compared to random fault-injected BPs. The reason is that user-developed BPs typically have a smaller number of faults because of the users' understanding of the BP and the underlying service semantics, as well as the extensive debugging that they perform.

Table 4.7: Evaluation results over user-developed BP dataset

No.	$fops$	Status	Domain	Fault Types	Time (sec.)	Iterations	Overlap	Similarity
1	1	Resolved	Insurance	Relational operator replacement	68.30	1	0.63	0.25
2	1	Resolved	Insurance	Relational operator replacement, Variable identifier replacement	66.24	1	0.63	0.25
3	1	Resolved	Insurance	Variable identifier replacement, Activity order exchange	232.89	4	0.85	0.82
4	2	Failed	Insurance	Variable identifier replacement, Activity order exchange	804.45	13	N/A	N/A
5	2	Failed	Insurance	Variable identifier replacement, Activity order exchange	869.39	12	N/A	N/A
6	2	Resolved	Flight Reservation	Relational operator replacement, Path operator replacement, Variable identifier replacement	86.25	2	0.91	0.84
7	2	Resolved	Flight Reservation	Path operator replacement, Numeric constant modification, Variable identifier replacement	83.91	2	0.91	0.84
8	2	Resolved	Flight Reservation	Logical operator replacement, Path operator replacement	88.76	2	0.87	0.76
9	2	Resolved	Flight Reservation	Relational operator replacement, Branch path removal	330.96	6	0.79	0.63
10	1	Resolved	Flight Reservation	Branch path removal	332.08	6	0.79	0.63
11	1	Resolved	Flight Reservation	Path operator replacement, Numeric constant modification	112.98	2	0.91	0.86
12	2	Resolved	Ecommerce	Relational operator replacement, Variable identifier replacement	209.46	2	0.54	0.02
13	1	Resolved	Ecommerce	Numeric constant modification	237.16	4	0.82	0.66
14	1	Resolved	Ecommerce	Branch path removal, Variable identifier replacement	44.22	1	0.53	0.31
15	1	Failed	Ecommerce	Relational operator replacement, Variable identifier replacement	936.88	15	N/A	N/A

4.4.3 Parameter sensitivity

We now discuss the key parameters that affect the performance of CFR in terms of accuracy and fault resolution time. There are three key parameters: i) α that determines the size of the faulty BP subgraph for pair-wise comparison with existing BPs; ii) similarity between faulty BP and existing BPs; iii) service overlap between faulty BP and existing BPs.

As shown in Table 4.3, increasing α results in an increase in the number of rules as well as the number of transformations encoded in the rules, thus increasing the computation time. A large α value may result in a comparison of the faulty BP with unrelated BPs, which also results in the generation of transformation rules that may change the goal and scope of the original BP. On the other hand, a small α value may result in too small of a search region for fault resolution. Therefore, it is important to start with a small value of α and incrementally increase it until the fault is resolved as per the satisfaction of the BP designer.

The accuracy of CFR depends on the structural similarity and service overlap between the faulty BP and some minimum number of BPs in the repository of existing BPs. Higher the similarity and degree of service overlap, the more likely it is to correctly resolve the faults in a given BP as depicted in Fig. 4.6. The BP designer can compute the similarity and service overlap values for a given faulty BP, and if these values meet the threshold values only then the fault resolution approach is applied. These threshold values need to be determined beforehand considering the available BP repository. As shown in Table 4.2, we considered a repository of 48 existing BPs from 3 different domains. Our experimental evaluation results show that if the similarity and service overlap of the faulty BP with at least 25 percent of existing BPs in the repository is above 0.4 and 0.7 respectively, then the likelihood of fault resolution increases significantly.

4.5 Experimental Evaluation (Hybrid)

The hybrid approach combines both EGV and CFR in an interleaved fashion. As discussed in Section 4.3, we consider two versions of the hybrid approach, H1, and H2.

Table 4.8 shows the accuracy results of both H1 and H2 in comparison with CFR. As expected, both H1 and H2 have similar accuracy because both of them use the same underlying approaches but in a different order. Fig. 4.8 compares the execution time taken by CFR, H1 and H2. Overall, CFR performs slightly better than H1 and much better than H2 which takes almost twice as much time as CFR. However, H1 is more time-efficient for BPs with 1 or 2 $fops$, whereas, both H2 and CFR perform better than H1 for BPs with 3 and 4 $fops$. This is due to the fact the BPs with a higher number of $fops$ have complex faults that cannot be resolved in isolation as done by EGV.

Table 4.8: Accuracy of H1, H2 and CFR.

$fops$	No. of Bps	Accuracy			Fix Candidates		
		H1	H2	CFR	H1	H2	CFR
1	46	0.83	0.83	0.85	9	31	9
2	119	0.79	0.79	0.73	22	51	21
3	24	0.71	0.71	0.71	33	50	35
4	19	0.74	0.74	0.53	78	90	66
Overall	208	0.78	0.78	0.74	25	50	22

In the presence of such complex faults, CFR and H2 (executing CFR first) perform better with respect to execution time because of the higher fault coverage of CFR.

4.5.1 Discussion

EGV outperforms CFR in terms of computation time with comparable accuracy but CFR provides a broader fault coverage as discussed in Section 4.3. For example, CFR can resolve faults belonging to branches and activity removal. These faults cannot be resolved by the basic G&V and EGV. However, CFR requires a repository of existing BPs.

If such a repository exists, EGV can be combined with CFR to get the best of both worlds. This is exactly what the hybrid approach is designed to achieve. Both hybrid approaches H1 and H2 have similar accuracy across all $fops$ as depicted in Table 4.8. The accuracy of CFR is slightly higher than the hybrid approach (both H1 and H2) for one fop but is taken over by the hybrid approach as the number of $fops$ increases. Consequently, the hybrid approach is preferred over CFR for better accuracy especially when a higher number of $fops$ are expected. In terms of performance, H1 is a better candidate for BPs with 1 or 2 $fops$. However, for BPs with a higher number of $fops$ H2 performs better than H1.

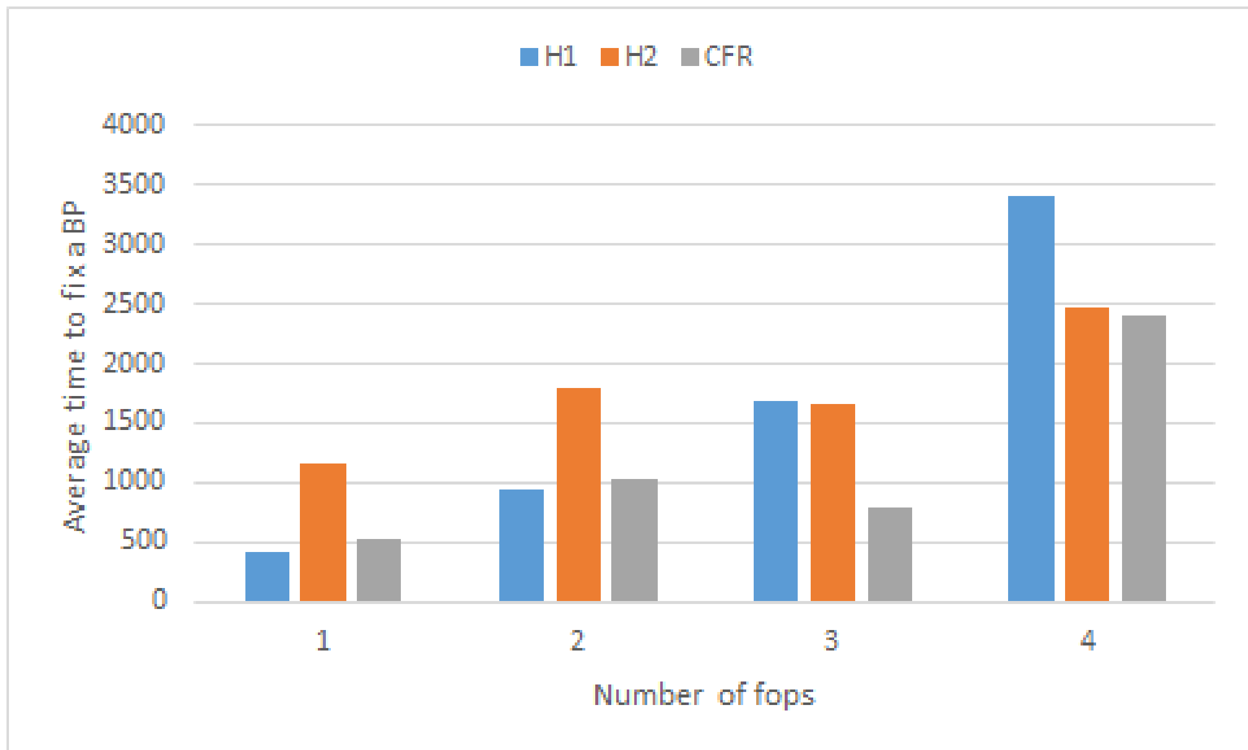


Figure 4.8: Execution time comparison of H1, H2, and CFR.

4.6 Chapter Summary

In this chapter, we have formalized the problem of collaborative fault resolution (CFR) which utilizes information from existing correct BPs that use similar services to resolve the faults in a user-developed BP. We present an approach based on association analysis to identify and iteratively select modifications to resolve the fault(s) in the user-developed BP. Additionally, we have also presented a hybrid approach by combining CFR and EGV in sequential order. The hybrid approach achieves superior accuracy and broader coverage of faults. Chapter 5 presents the implementation details of our prototype tool that enables a user to submit their BPs for fault resolution.

Chapter 5

Prototype Implementation

This chapter presents an overview of the BP-DEBUG prototype that exposes the fault resolution capability using Efficient Generate and Validate (EGV), Collaborative Fault Resolution (CFR) and Hybrid approaches.

5.1 Introduction

We have implemented a prototype, called BP-DEBUG, for fault resolution of business processes in a cloud-based environment. The prototype implements our Efficient Generate-and-Validate (EGV) presented in chapter 3 and, Collaborative Fault Management (CFR) and Hybrid approaches presented in chapter 4 to locate and fix faults in the faulty BP. The prototype is implemented as a web-based application that allows users to specify a particular fault resolution strategy. Users can choose to apply more than one approach for the given faulty BP. If BP-DEBUG is able to fix faults, it depicts the fixed BP on the web-based interface and a fault report by highlighting the differences between the submitted faulty BP and the fixed BP. BP-DEBUG is implemented as an enhancement to BP-Com [16] and is thus capable of generating BPEL source code of the fixed BP for deployment.

BP-DEBUG provides the following functionalities for BP fault resolution:

1. Specification of the faulty BP with its control flow, data flow, and corresponding test suite

2. Resolution of faults using the approach of user's choice

3. Executable code generation of the fixed BP for deployment

Intended Users: BP-DEBUG tool is developed for novice users with little technical expertise in debugging the BPs. In particular, it is suitable for small and medium enterprises that cannot dedicate sufficient resources to debugging. Additionally, experienced users can also use our system to speed up the debugging cycle for their BPs.

5.2 BP-DEBUG: Architecture and Implementation

Fig. 5.1 depicts a high-level architectural overview of the BP-DEBUG system. The fault resolution process starts when the user specifies the faulty BP with the associated test suite. After the input, BP-DEBUG performs fault resolution and generates the candidate fixes that are validated against the test suite. If faults are resolved, the fixed BP is presented to the user on the web-based interface, along with the corresponding fault report.

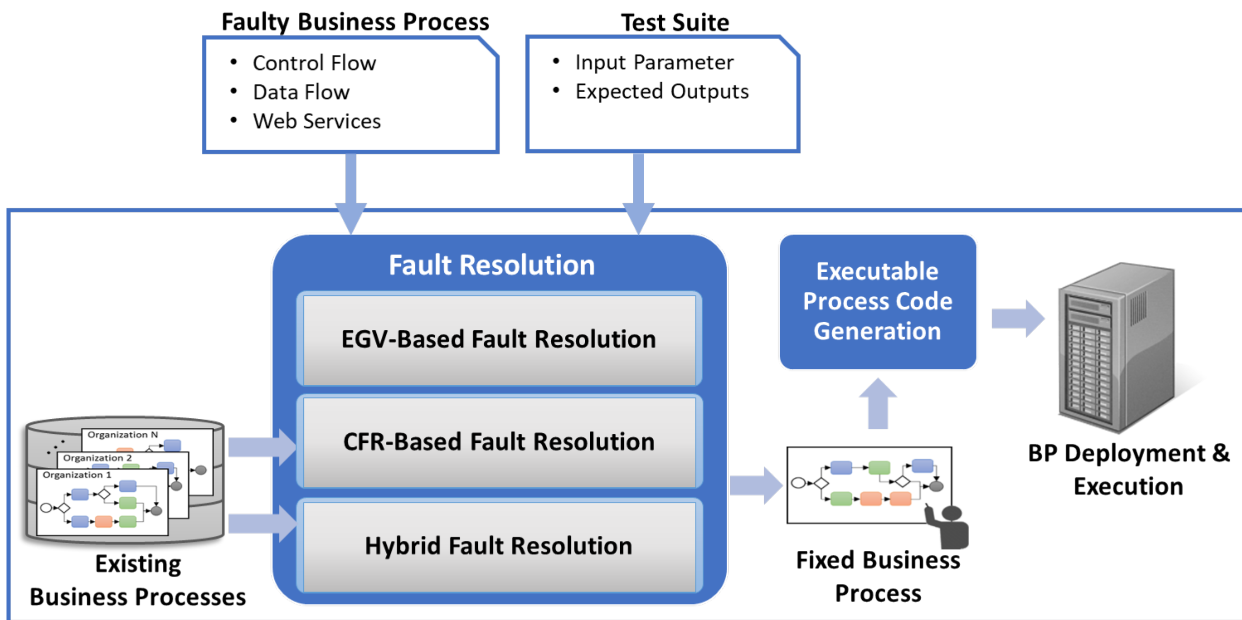


Figure 5.1: Architectural overview of the BP-DEBUG System

The BP designer can optionally make some changes to the fixed BP and proceed with code generation and deployment. Below, we discuss some of the key functions of the BP-DEBUG prototype.

5.2.1 Specification of the Faulty BP and the Test suite

The BP designer initiates the fault resolution process on BP-DEBUG by providing a faulty BP with its complete data flow control flow and the associated test suite. The control flow of the BP shown in Fig. 1.2 is shown in Listing 5.1.

Listing 5.1: Control flow of the BP of Fig. 1.2

```
1  #ss
2  searchProduct
3  #se
4  #ts
5  createInvoice
6  #te
7  #rs
8  searchProduct#verifyEmail
9  verifyEmail#calcSalesTax
10 calcSalesTax#createOrder
11 createOrder#xors_x1
12 xors_x1#processCODPayment
13 xors_x1#chargeCreditCard
14 processCODPayment#xore_x1
15 chargeCreditCard#xore_x1
16 xore_x1#createInvoice
17 #re
```

In Listing 5.1, Lines 1-6 specify the starting and ending vertices of the BP in its control flow graph.

In this example, *searchProduct* is specified as a starting vertex and *createInvoice* is specified as the ending vertex. Lines 7-17 specify the BP graph as an edge list. For instance, Line 7 specifies that there is an edge between *searchProduct* service operation and *verifyEmail* service operation. Similarly, Line 11 specifies that *creatOrder* service precedes the beginning of *xor* node in the control flow.

```
1  <?xml version='1.0' encoding='utf-8'?>
2  <BusinessProcess name="BP_1">
3  <Operation nodeType="invoke" name="createOrder"
4      url="http://localhost:8094/WrapperOFBiz_WebServices/services/CreateOrder">
5      <input>
6        <attribute name="product_code" datatype="STRING" source="searchProduct"
7          attributetype="OUTPUT" sourceAttrib="productId"/>
8        <attribute name="sales_tax" datatype="DECIMAL" source="calcSalesTax"
9          attributetype="OUTPUT" sourceAttrib="tax_amount"/>
10       <attribute name="quantity" datatype="INTEGER" source="USER" attributetype="NA"/>
11     </input>
12     <output>
13       <attribute name="ord_no" datatype="STRING"/>
14       <attribute name="total_amnt" datatype="DECIMAL"/>
15     </output>
16   </Operation>
```

Figure 5.2: Data flow of *createOrder* service in the BP of Fig. 1.6

Fig. 5.2 shows the portion of the data flow corresponding to *createOrder* service operation. The *Operation* tag specifies that *createOrder* is an *invoke* element or the service operation followed by its URL. The *input* and *output* tags contain the service's input and output attributes, respectively. Additionally, *attribute* tags within the *input* also specify data sources of input attributes. For instance, the value of *product_code* parameter is provided *productId*, which is an output parameter of *createOrder* service. Similarly, *quantity* parameter takes its value directly from the user at execution time.

The test suite is also specified in XML format. Fig. 5.3 depicts one sample test case from the test suite. The *TestCase* tag represents a complete test case with specifications of input and output parameters. Input parameters are specified with their data types and values to be used for the execution of the containing test case. For instance, the values of *cvc* and *payMethod* parameters are *123* and *CREDIT_CARD* respectively. The output parameters are associated with expected values and assertions. Expected values can either be fixed or must conform to the given regular expression. The assertions can take the form of an output parameter being null or not null. For

```xml
<?xml version='1.0' encoding='utf-8'?>
<TestSuite name="ts1">
    <TestCase>
        <input>
            <attribute name="productName" datatype="STRING" value="Wireless Keyboard"/>
            <attribute name="quantity" datatype="INTEGER" value="2"/>
            <attribute name="payMethod" datatype="STRING" value="CREDIT_CARD"/>
            <attribute name="card_no" datatype="STRING" value="4111111111111"/>
            <attribute name="cvc" datatype="STRING" value="123"/>
        </input>
        <output>
            <attribute name="ord_no" datatype="STRING"/>
                <notnull/>
                <regex pattern="^UO\d+$"/>
            </attribute>
            <attribute name="invoiceId" datatype="STRING"/>
                <notnull/>
                <regex pattern="^UI\d+$"/>
            </attribute>
            <attribute name="invoiceDate" datatype="STRING"/>
                <notnull/>
                <date date-pattern="YYYY-mm-dd"/>
            </attribute>
            <attribute name="total_amnt" datatype="DECIMAL"/>
                <notnull/>
                <exact-match value="40.7"/>
            </attribute>
            <attribute name="trans_no" datatype="STRING"/>
                <notnull/>
                <regex pattern="^UT\d+"/>
            </attribute>
            <attribute name="exp_date" datatype="STRING"/>
                <null/>
            </attribute>
        </output>
    </TestCase>
</TestSuite>
```

Figure 5.3: Test suite for the BP of Fig. 1.6

instance, the *total_amnt* output parameter has to be verified using an exact value match against the provided value. *ord_no*, *invoiceId* and *trans_no* must be non-null and conform to the specified regular expression. *exp_date* parameter must be null for this particular test case because it is produced by *processCODPayment* which will not be executed in the execution path due to the specification of credit card as a payment method.

Although the control flow, the data flow, and the test cases are specified in a custom format, the user does not have to worry about the generation of these documents if they used BP-Com [16] for

the development of the BP. The control flow and data flow files are part of the downloaded BP from BP-Com and can be forwarded to the BP-DEBUG system without any modification. Furthermore, the test case specifications are optional for sales-order, insurance, and flight reservation BPs. A set of predefined test cases are used for verification if the user does not provide the test suite.

5.2.2 Fault Resolution

The fault resolution process starts after the faulty BP and the test suite is specified. The faults are resolved using the fault resolution scheme of the user's choice. The user can choose to repair the BP using Efficient Generate-and-Validate (EGV), Collaborative Fault Resolution (CFR), or Hybrid fault resolution scheme. The detail of these approaches has already been discussed in chapter 3 and 4. BP-DEBUG tool generates the candidate fixes according to the chosen approach and all the candidate fixes are validated against the test suite. If a candidate fix passes all the test cases, it is presented to the BP designer with the original faulty BP and the set of transformations that were required for the fixes. If a candidate fix does not completely fix the fault but passes some test cases failed by the original faulty BP, it is passed back to the fault localization component for identification of the remaining faults as well as for discovering new faults that were caused by applying the transformations. The process continues until the faulty BP is fixed or the entire search space is exhausted.

Fig. 5.4 and Fig. 5.5 depict two distinct faulty BPs from insurance domain and their relevant fixes on the user interface of our tool. The BP designer can navigate to different service operations both in fixed BP and faulty BP. Fig. 5.4 shows *fop* and control flow faults in its close vicinity. The fault shown in Fig. 5.4 is caused by composing two services in parallel that are data dependent. The corresponding fragment of the fixed BP composes the dependent services in a sequence for resolving the fault.

Fig. 5.5 shows incorrect mappings in the faulty BP for the attributes of *ValidateCardNumber* service operation. In this service operation, *CardNumber* attribute is incorrectly mapped to *CardType* attribute and vice versa. The corresponding fragment of the fixed BP depicts the fix of this fault. In addition to faults, the BP designer is also presented with a list of *fops*, the number

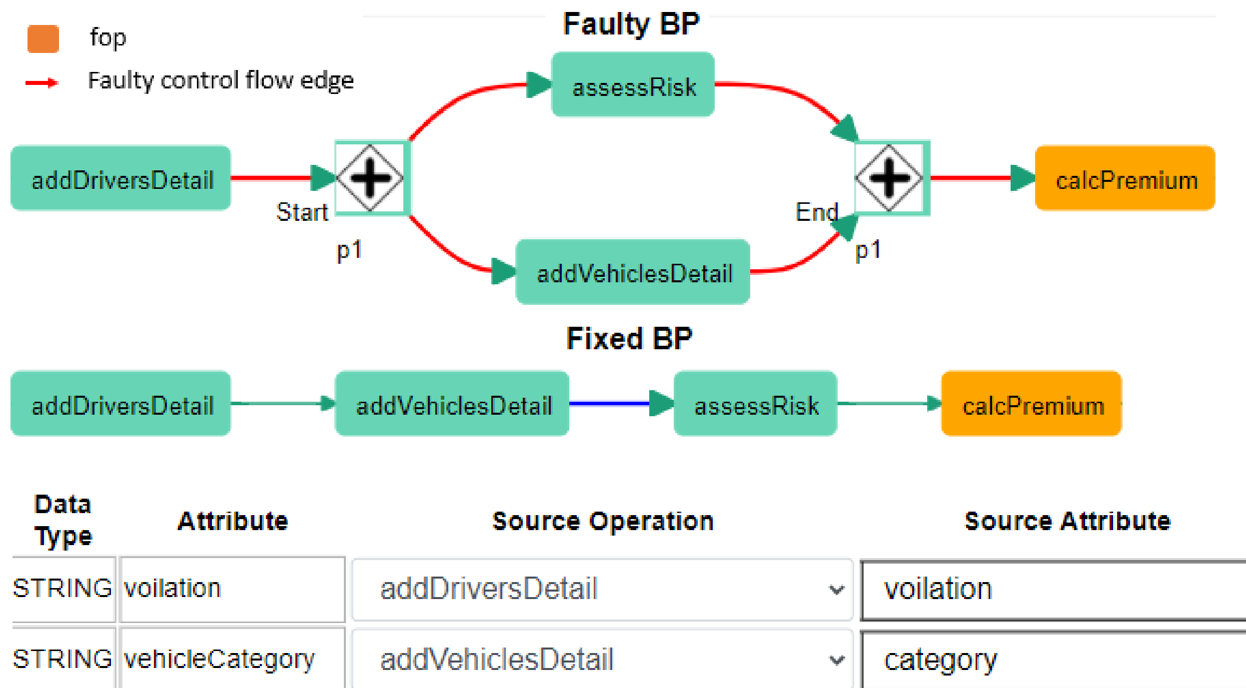


Figure 5.4: Visual representation of *fops* and control flow faults and their fixes in BP-DEBUG

of candidate fixes in each iteration, as well as the approach-related parameters like support and confidence in case CFR is chosen for resolving faults.

5.2.3 Code Generation and BP Deployment

The user can continue with the fixed BP from the previous stage and investigate its data flow and control flow. The tool is integrated with the authoring and refinement component of our existing BP composition and management tool (BP-Com) [16] to enable the user to optionally modify the fixed BP before proceeding with code generation. Finally, code generation component of BP-Com can be used to generate the executable BPEL code for the fixed BP. The generated code can be deployed within the BP designer's organization, at their site, or in the cloud on Apache ODE (Orchestration Director Engine) server, Kubernetes, or any server supporting BPEL specifications and interactions with the partner links.

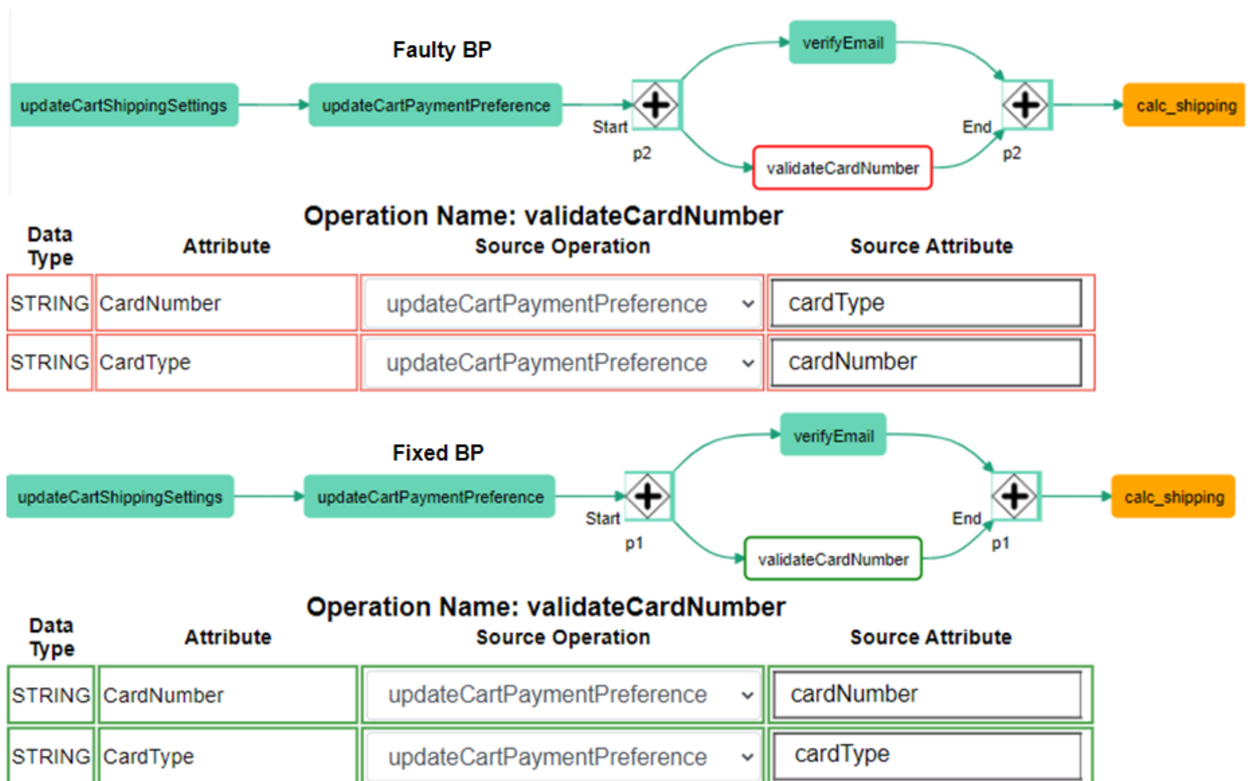


Figure 5.5: Identification and representation of data flow faults in BP-DEBUG fault report interface

5.3 Related Tools and Research Prototypes

Most of the existing tools for debugging and automated program repair are designed for Java and C/C++ programs. There is a lack of tools providing debugging and fault resolution support for BPs developed through web service orchestration. Below we discuss some of the related tools.

Web service testing: Significant work has been done on the testing of web services and several open-source and commercial tools are available to support unit testing of web services. Some examples are Apache JMeter, SoapSonar, SoapUI, and Storm tools. All these tools support skeleton code generation for invocation and testing of web service operations. WS-TAXI [8] is a web service testing tool that, given the WSDL definition of a web service, automatically generates test cases using a combination of coverage information and data-driven approaches. For coverage information, it uses SoapUI [158]. While these tools are designed for testing individual web services in isolation, there are some frameworks supporting integration testing. Integration testing can not

only test the individual web services but also their interactions, messages, and overall composition [124]. Notable work on integration testing for service composition includes [125, 127, 130]. Tarhini et al. [125] proposed an automatic test generation, execution, and validation approach from WSDL descriptions of service compositions. Huang et al. proposed an integration testing approach [127] that supports simulating unavailable component services to allow for continuous integration testing from the early phases of development, especially when some components are not available. Sun et al. [130] presented a framework for test case and test data generation given a service composition in BPEL. Since our tool requires a set of test cases as input, any of the existing test case generation approaches can be utilized for the purpose.

Fault localization and resolution: Fault localization is one of the key requirements for any automated tool for program repair & recovery. Several fault localization approaches and tools exist for traditional C/C++ and Java programs [18] with very few tools for BPs. BPELSwice [61] is a fault localization technique that performs predicate switching and program slicing to identify the BP fragments that are likely to contain faults. BPELDebugger [60] provides a framework with a set of BPEL-specific guidelines to localize faults. It allows using many fault localization formulae to rank the suspicious program fragments. In our implementation, we use the statistical fault localization technique [29] for identifying fault observation points. The existing tools for fault resolution and recovery in BPs such as WS-TAXI [8], VieDame [159], and BPEL Debugger [60] mainly consider faults that are related to service implementation, failure of services, deployment problems or issues related to network and communication. Our tool focuses on the detection and resolution of faults that are introduced in the BP at design time.

There is notable work on the automated resolution of faults in traditional programs written in C/C++ and Java [18, 94]. However, the proposed approaches are not designed to address fault resolution in BPs that are developed by composing web services. Xu et al. [18] have proposed a Generate-and-Validate (G&V) based approach that heuristically explores the search space of a faulty program to generate candidate fixes given a set of test cases. These candidate fixes are validated in a feedback loop to repair program faults in an efficient manner. However, this approach is implemented for Java programs and needs to be adapted for BPs.

5.4 Chapter Summary

In this chapter, we presented the BP-DEBUG prototype tool, which is a web-based implementation of our approach for fault resolution in the services cloud environment. The tool enables a user to upload a faulty BP with a set of test cases for the resolution of faults. BP-DEBUG employs Collaborative Fault Resolution (CFR), Efficient Generate-and-Validate (EGV), or Hybrid fault resolution approach based upon the user's choice. In the end, the BP designer is furnished with a fault report including the detected faults and their fixes in a graphical manner. The BP designer can make further modifications to the BP and proceed with code generation and deployment of the fixed BP.

Chapter 6

Conclusion and Future Work

This chapter concludes the dissertation by summarizing contributions and presenting future research directions.

6.1 Research Contributions

In this dissertation, we have presented two approaches for fault resolution of Business Processes (BPs) in the services cloud environment. The first approach, Efficient Generate, and Validate (EGV) fixes faults by generating the candidate fixes with the application of mutation operators defined for BPs. The second approach is Collaborative Fault Resolution (CFR) which fixes faults using the knowledge of existing correct BPs in the cloud. Then, we combine EGV and CFR in a hybrid approach and measure their effectiveness in resolving faults. All the approaches allow the user to fix faults in their BPs by providing a faulty BP with the corresponding test suite. Below we provide a summary of specific contributions in this dissertation:

1. *Efficient G&V (EGV) approach.* EGV presents an effective solution for resolving faults in Business Processes (BPs) by expanding upon the established G&V automated program repair methodology. To increase efficiency, our approach leverages statistical fault localization and predicate-based switching and slicing to analyze a smaller subgraph of the BP. Additionally, we enhance the efficiency of fault resolution through static analysis and con-

ditional mutant generation. These heuristics significantly reduce the number of mutants that need to be examined for fixing faults with reasonable accuracy. It also proved more accurate and efficient than Collaborative Fault Resolution on a subset of faulty BPs. However, EGV, by design, cannot generate fixes for certain kinds of faults. For instance, the fix cannot be generated when a branch path or an activity is removed from a BP. This issue is inherent in mutation-based fault resolution schemes. To address this challenge, the generation process of candidate fixes must be replaced with pattern or learning-based techniques.

2. We formalize the problem of Collaborative Fault Resolution, which involves resolving faults in a faulty BP by utilizing information of existing, fault-free BPs that use similar services. To achieve this, we have developed a heuristic approach that utilizes association analysis to identify potential transformations that are iteratively applied to address the fault(s) in the BP while minimizing changes to the original BP. This approach represents a significant improvement over current automated program repair methods, as it draws upon the knowledge of functional BPs rather than solely examining the problematic BP in isolation. CFR is a huge improvement over the basic G&V approach in the number of generated fix candidates. It is also capable of targeting more fault categories than EGV because it searches the fixes from existing fault-free BPs. However, the complete availability of existing BPs may not always be possible and the existing BPs may also include heterogeneous web services. It would be interesting to study this approach by relaxing the homogeneity assumption under a privacy-preserving environment.

3. *Experimental Evaluation*. In order to assess the effectiveness of our approach, we conducted a comprehensive experimental evaluation using both synthetic and real data. Synthetic data was generated by randomly introducing faults, allowing for comprehensive testing of all possible design time faults. Real data was obtained from a user study in which participants developed BPs as part of a class exercise, thereby testing the effectiveness of our approach in resolving faults introduced by actual users. We compared our approach to a baseline Generate-and-Validate (G&V) automated program repair methodology. The results of our evaluation demonstrate that our approach can successfully resolve a wider range of faults

with high accuracy, outperforming the baseline methodology by a significant margin.

4. *Hybrid Approach.* We propose a hybrid approach that combines Efficient G&V (EGV) and collaborative fault resolution (CFR) techniques to improve accuracy. In the hybrid approach, both CFR and EGV are reinforcing each other. For instance, if a faulty BP is fixable by either of the approaches, it will be fixed by the hybrid approach. Consequently, it has higher accuracy than CFR with comparable running time, especially in the case of H1 (H1 invokes EGV before CFR).

5. We enhance an established framework for automated BP composition and management in a services cloud environment [16] by incorporating automated fault resolution capabilities. Furthermore, we validate the potential of this integrated framework by creating a prototype implementation that facilitates BP composition and automates the resolution of faults.

6.2 Challenges

This dissertation presents a framework for fault resolution in Business Processes. The framework includes two approaches for fault resolution including EGV and CFR. EGV builds atop basic G&V and uses fault localization to identify the suspected program elements which are further filtered by predicate switching and program slicing. After that, the elements in the faulty region are explored efficiently for generating the candidate fixes. These fixes are validated against the test suite to verify their correctness. Although EGV improves the efficiency of basic G&V in the number of generated candidate fixes, it cannot fix certain types of faults. For instance, it cannot recover the removed activities and paths from a BP because the fix for such faults is not available in the search space of the faulty BP.

CFR, on the contrary, does not depend upon mutations to fix the faulty BP but it mines the candidate fixes from a set of existing fault-free BPs. Like EGV, CFR starts the fault resolution process with fault localization. Afterward, it selects a region around the fault observation point and compares it with similar regions of the existing BPs. The differences between the faulty and the existing BPs are analyzed using association rule mining to discover recurring candidate fixes across

different existing BPs. Again, the test suite is used to verify the correctness of generated candidate fixes. Although CFR achieves higher accuracy than EGV, it mandates the complete availability of existing fault-free and homogeneous BPs. Both heterogeneity and privacy are detrimental to the functioning of CFR. We also combined CFR and EGV to develop a hybrid approach that combines the strengths and challenges of both approaches.

Fault resolution approaches proposed in this dissertation consider the functional faults that occur during the development of a BP orchestration with no regard to deployment and configuration faults which are commonplace especially when BPs and web services are hosted on distributed cloud instances. It would be interesting to study how the existing approaches like [63] can be integrated into our framework for handling development, deployment, and configuration faults. The integration is expected to be straightforward for the simple scenarios where a BP contains only faults of one type, that is, it either contains development fault(s) or configuration fault(s). However, in a complex setup where the BP is failing due to a combination of faults would require a more sophisticated solution.

We have also created a prototype implementation that allows a user to provide a faulty BP and test cases for resolving faults. The prototype is integrated with the proposed fault resolution approaches and fixes the faulty BP using the approach of the user's choice. Although, it is quite helpful for users in their debugging efforts however a more user-friendly approach can be taken by providing the recommendations when the BP is still under development. Such a system would require the adaptation of proposed approaches to handle partially built BPs. The adaptation is required at all three levels including fault localization, generation of candidate fixes, and their verification.

6.3 Future Work

The work presented in this dissertation provides several directions for future research in the area of BP development and fault resolution. Below we present the future directions of our work.

6.3.1 Recommendation System

One natural extension of our research is a recommendation system that aids a BP designer during the development of a BP. The intended recommendation system is meant to monitor the subject BP in the development phase and informs the BP designer of the potential faults. For instance, the system can be integrated into a BP development tool like BP-Com [16] and can provide meaningful insights to the BP designer. This recommendation system can investigate the resulting BP and informs the user of potential faults and generate fixes.

The main challenge in realizing such a recommendation system lies in the ability to execute an under-development BP for fault identification, localization, and candidate fix generation.

6.3.2 Collaborative Resolution of Configuration Faults

Delta Debugging [63] is designed to discover the configuration faults on different copies of the same service hosted on different cloud instances. It applies a series of *deltas* (small changes) to the configurations of a working service instance to discover the minimal set of transitions that cause the failure.

CFR can be modified to automatically detect and fix configuration faults. Specifically, given a failed service instance and multiple passing instances of the same service, we can compute the pairwise differences among their configurations and discover faults by subsequently applying association mining to compute fault-covering transformations.

6.3.3 Collaborative Fault Resolution in Heterogeneous Environment

One assumption that we took with Collaborative Fault Resolution (CFR) is homogeneity. That is, we assume that there is no heterogeneity among the web services used for the composition of BPs. This assumption, however, limits the number of existing fault-free BPs that can be leveraged for fault resolution. One logical extension to our work on CFR could be to relax this assumption and consider fault-free BPs composed using heterogeneous services.

The problem is to identify services that potentially use different nomenclature for their at-

tributes and services but essentially provide the same functionality. The problem of heterogeneity can be addressed by adapting the probabilistic schema matching approach of ASSEMBLE [2]. ASSEMBLE uses attribute, structural, and semantic-based matching between different services for resolving heterogeneity between the service pairs.

6.3.4 Privacy-preserving Resolution of BP Faults

In CFR, we also assumed that we have complete information of the existing BPs hosted on the cloud and can be used for pairwise comparison and association rule mining without any privacy concerns from the organizations that own the BPs. One possible extension to our work could be to relax this assumption and implement CFR in a privacy-preserving environment. Specifically, we want to compute the pair-wise differences between the faulty BP and existing BPs when the owners of existing BPs choose to share only partial information about their BPs.

The privacy concerns can be addressed using secure multi-party computing that allows a public function to be computed while keeping the individual inputs of the participants private. However, the application of secure multi-party computing is more challenging in our scenario because we need pair-wise differences between the faulty BP and each of the existing BPs. Given the faulty BP and its difference from one of the existing BP, the complete existing BP can be reconstructed hence violating privacy.

Bibliography

[1] G. Huang, X. Liu, Y. Ma, X. Lu, Y. Zhang, and Y. Xiong, "Programming situational mobile web applications with cloud-mobile convergence: An internetware-oriented approach," *IEEE Transactions on Services Computing*, vol. 12, no. 1, pp. 6–19, 2016.

[2] A. Afzal, B. Shafiq, S. Shamail, A. Elahraf, J. Vaidya, and N. R. Adam, "Assemble: Attribute, structure and semantics based service mapping approach for collaborative business process development," *IEEE Transactions on Services Computing*, vol. 14, no. 2, pp. 371–385, 2021.

[3] A. Kurniawan, *Learning AWS IoT: Effectively manage connected devices on the AWS cloud using services such as AWS Greengrass, AWS button, predictive analytics and machine learning*. Packt Publishing Ltd, 2018.

[4] F. Brito e Abreu, J. Cardoso, J. Oliveira, C. Serrão, A. M. Pinto, F. Araujo, R. P. Paiva, J. Correia, and A. Lopes, "Taverna workflow management system." `https://taverna.incubator.apache.org/`, 2021 (accessed July 25, 2021).

[5] J. Kranjc, R. Orač, V. Podpečan, N. Lavrač, and M. Robnik-Šikonja, "Clowdflows: Online workflows for distributed big data mining," *Future Generation Computer Systems*, vol. 68, pp. 38–58, 2017.

[6] "Amazon web services (aws) - cloud computing services." `https://aws.amazon.com/`. (Accessed on 05/21/2019).

[7] "Bizagi - low-code automation leader." https://www.bizagi.com/en. (Accessed on 04/29/2023).

[8] C. Bartolini, A. Bertolino, E. Marchetti, and A. Polini, "Ws-taxi: A wsdl-based testing tool for web services," in *2009 International Conference on Software Testing Verification and Validation*, pp. 326–335, IEEE, 2009.

[9] C.-a. Sun, G. Wang, B. Mu, H. Liu, Z. Wang, and T. Y. Chen, "A metamorphic relation-based approach to testing web services without oracles," *International Journal of Web Services Research (IJWSR)*, vol. 9, no. 1, pp. 51–73, 2012.

[10] H. Wang, X. Chen, Q. Wu, Q. Yu, X. Hu, Z. Zheng, and A. Bouguettaya, "Integrating reinforcement learning with multi-agent techniques for adaptive service composition," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 12, no. 2, p. 8, 2017.

[11] W. Song and H.-A. Jacobsen, "Static and dynamic process change," *IEEE Transactions on Services Computing*, vol. 11, no. 1, pp. 215–231, 2016.

[12] L. Baresi and S. Guinea, "Self-supervising BPEL processes," *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 247–263, 2011.

[13] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Delta debugging microservice systems with parallel optimization," *IEEE Transactions on Services Computing*, pp. 1–1, 2019.

[14] X. Zhou, X. Peng, T. Xie, J. Sun, W. Li, C. Ji, and D. Ding, "Delta debugging microservice systems," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 802–807, IEEE, 2018.

[15] A. Estero-Botaro, F. Palomo-Lozano, and I. Medina-Bulo, "Mutation operators for WS-BPEL 2.0," in *21th International Conference on Software & Systems Engineering and their Applications*, 2008.

[16] A. Afzal, M. A. Zahid, A. Akhtar, B. Shafiq, S. Shamail, A. Elahraf, J. Vaidya, and N. Adam, "BP-Com: A service mapping tool for rapid development of business processes," in *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pp. 1235–1238, IEEE, 2020.

[17] L. Chen, Y. Pei, M. Pan, T. Zhang, Q. Wang, and C. A. Furia, "Program repair with repeated learning," *IEEE Transactions on Software Engineering*, pp. 1–1, 2022.

[18] T. Xu, L. Chen, Y. Pei, T. Zhang, M. Pan, and C. A. Furia, "Restore: Retrospective fault localization enhancing automated program repair," *IEEE Transactions on Software Engineering*, 2020.

[19] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2019.

[20] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 1–11, IEEE, 2018.

[21] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[22] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model checking*. MIT press, 2018.

[23] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model checking programs," *Automated software engineering*, vol. 10, no. 2, pp. 203–232, 2003.

[24] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. L. Dill, "Cmc: A pragmatic approach to model checking real code," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 75–88, 2002.

[25] F. Tip and T. Dinesh, "A slicing-based approach for locating type errors," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 10, no. 1, pp. 5–55, 2001.

[26] H. Cleve and A. Zeller, "Locating causes of program failures," in *Proceedings of the 27th international conference on Software engineering*, pp. 342–351, ACM, 2005.

[27] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," *Acm Sigplan Notices*, vol. 40, no. 6, pp. 15–26, 2005.

[28] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "Sober: statistical model-based bug localization," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 286–295, ACM, 2005.

[29] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: A hypothesis testing-based approach," *IEEE Transactions on software engineering*, vol. 32, no. 10, pp. 831–848, 2006.

[30] M. D. Weiser, *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. University of Michigan, 1979.

[31] M. Weiser, "Program slicing," *IEEE Transactions on software engineering*, no. 4, pp. 352–357, 1984.

[32] R. Lyle, "Automatic program bug location by program slicing," in *Proceedings 2nd international conference on computers and applications*, pp. 877–883, 1987.

[33] D. Liang and M. J. Harrold, "Equivalence analysis and its application in improving the efficiency of program slicing," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, no. 3, pp. 347–383, 2002.

[34] C. Mateis, M. Stumptner, and F. Wotawa, "Modeling java programs for diagnosis," in *ECAI*, pp. 171–175, 2000.

[35] R. Abreu and A. J. Van Gemund, "A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis," in *Eighth Symposium on Abstraction, Reformulation, and Approximation*, 2009.

[36] G. K. Baah, A. Podgurski, and M. J. Harrold, "The probabilistic program dependence graph and its application to fault diagnosis," in *Proceedings of the 2008 international symposium on Software testing and analysis*, pp. 189–200, 2008.

[37] W. Mayer, R. Abreu, M. Stumptner, A. J. Van Gemund, *et al.*, "Prioritising model-based debugging diagnostic reports," in *Proceedings of the 19th International Workshop on Principles of Diagnosis*, pp. 127–134, Citeseer, 2008.

[38] F. Wotawa, J. Weber, M. Nica, and R. Ceballos, "On the complexity of program debugging using constraints for modeling the program's syntax and semantics," in *Conference of the Spanish Association for Artificial Intelligence*, pp. 22–31, Springer, 2009.

[39] F. Wotawa, "Fault localization based on dynamic slicing and hitting-set computation," in *2010 10th International Conference on Quality Software*, pp. 161–170, IEEE, 2010.

[40] Z. A. Al-Khanjari, M. R. Woodward, H. A. Ramadhan, and N. S. Kutti, "The efficiency of critical slicing in fault localization," *Software Quality Journal*, vol. 13, no. 2, pp. 129–153, 2005.

[41] M. A. Alipour and A. Groce, "Extended program invariants: applications in testing and fault localization," in *Proceedings of the Ninth International Workshop on Dynamic Analysis*, pp. 7–11, 2012.

[42] X. Ju, S. Jiang, X. Chen, X. Wang, Y. Zhang, and H. Cao, "Hsfal: Effective fault localization using hybrid spectrum of full slices and execution slices," *Journal of Systems and Software*, vol. 90, pp. 3–17, 2014.

[43] X. Mao, Y. Lei, Z. Dai, Y. Qi, and C. Wang, "Slice-based statistical fault localization," *Journal of Systems and Software*, vol. 89, pp. 51–62, 2014.

[44] Y. Wang, H. Patil, C. Pereira, G. Lueck, R. Gupta, and I. Neamtiu, "Drdebug: Deterministic replay based cyclic debugging with dynamic slicing," in *Proceedings of annual IEEE/ACM international symposium on code generation and optimization*, pp. 98–108, 2014.

[45] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *Proceedings of the 28th international conference on Software engineering*, pp. 272–281, 2006.

[46] T. Gyimóthy, A. Beszédes, and I. Forgács, "An efficient relevant slicing method for debugging," *ACM SIGSOFT Software Engineering Notes*, vol. 24, no. 6, pp. 303–321, 1999.

[47] M. Renieres and S. P. Reiss, "Fault localization with nearest neighbor queries," in *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, pp. 30–39, IEEE, 2003.

[48] J. A. Jones, M. J. Harrold, and J. T. Stasko, "Visualization for fault localization," in *in Proceedings of ICSE 2001 Workshop on Software Visualization*, Citeseer, 2001.

[49] J. Jones and M. 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*, pp. 273–282, 01 2005.

[50] V. Debroy, W. E. Wong, X. Xu, and B. Choi, "A grouping-based strategy to improve the effectiveness of fault localization techniques," in *2010 10th International Conference on Quality Software*, pp. 13–22, IEEE, 2010.

[51] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, pp. 39–46, IEEE, 2006.

[52] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Transactions on software engineering and methodology (TOSEM)*, vol. 20, no. 3, pp. 1–32, 2011.

[53] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The dstar method for effective software fault localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, 2013.

[54] W. E. Wong, V. Debroy, and D. Xu, "Towards better fault localization: A crosstab-based statistical approach," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, no. 3, pp. 378–396, 2011.

[55] L. Naish, H. J. Lee, and K. Ramamohanarao, "Statements versus predicates in spectral bug localization," in *2010 Asia Pacific Software Engineering Conference*, pp. 375–384, IEEE, 2010.

[56] A. Zeller, "Isolating cause-effect chains from computer programs," *ACM SIGSOFT Software Engineering Notes*, vol. 27, no. 6, pp. 1–10, 2002.

[57] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[58] N. Gupta, H. He, X. Zhang, and R. Gupta, "Locating faulty code using failure-inducing chops," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pp. 263–272, 2005.

[59] F. Li, W. Huo, C. Chen, L. Zhong, X. Feng, and Z. Li, "Effective fault localization based on minimum debugging frontier set," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 1–10, IEEE, 2013.

[60] C.-a. Sun, Y. M. Zhai, Y. Shang, and Z. Zhang, "Bpeldebugger: An effective bpel-specific fault localization framework," *Information and Software Technology*, vol. 55, no. 12, pp. 2140–2153, 2013.

[61] C.-a. Sun, Y. Ran, C. Zheng, H. Liu, D. Towey, and X. Zhang, "Fault localisation for WS-BPEL programs based on predicate switching and program slicing," *Journal of Systems and Software*, vol. 135, pp. 191–204, 2018.

[62] Z. Ye, P. Chen, and G. Yu, "T-rank: A lightweight spectrum based fault localization approach for microservice systems," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 416–425, IEEE, 2021.

[63] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Delta debugging microservice systems with parallel optimization," *IEEE Transactions on Services Computing*, 2019.

[64] M. Mathur, *Leveraging Distributed Tracing and Container Cloning for Replay Debugging of Microservices*. University of California, Los Angeles, 2020.

[65] F. Schwander, R. Gopinath, and A. Zeller, "Inducing subtle mutations with program repair," in *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 25–34, IEEE, 2021.

[66] J. Lin, P. Chen, and Z. Zheng, "Microscope: Pinpoint performance issues with causal graphs in micro-service environments," in *International Conference on Service-Oriented Computing*, pp. 3–20, Springer, 2018.

[67] M. Ma, J. Xu, Y. Wang, P. Chen, Z. Zhang, and P. Wang, "Automap: Diagnose your microservice-based web applications automatically," in *Proceedings of The Web Conference 2020*, pp. 246–258, 2020.

[68] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, D. Liu, Q. Xiang, and C. He, "Latent error prediction and fault localization for microservice applications by learning from system trace logs," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 683–694, 2019.

[69] Y. Küçük, T. A. Henderson, and A. Podgurski, "Improving fault localization by integrating value and predicate based causal inference techniques," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 649–660, IEEE, 2021.

[70] M. Jia, Z. Cui, Y. Wu, R. Xie, and X. Liu, “Smfl integrating spectrum and mutation for fault localization,” in *2019 6th International Conference on Dependable Systems and Their Applications (DSA)*, pp. 511–512, IEEE, 2020.

[71] M. Wen, Z. Xie, K. Luo, X. Chen, Y. Yang, and H. Jin, “Effective isolation of fault-correlated variables via statistical and mutation analysis,” *IEEE Transactions on Software Engineering*, 2022.

[72] M. Zhang, Y. Li, X. Li, L. Chen, Y. Zhang, L. Zhang, and S. Khurshid, “An empirical study of boosting spectrum-based fault localization via pagerank,” *IEEE Transactions on Software Engineering*, vol. 47, no. 6, pp. 1089–1113, 2019.

[73] Q. I. Sarhan and Á. Beszédes, “Effective spectrum based fault localization using contextual based importance weight,” in *International Conference on the Quality of Information and Communications Technology*, pp. 93–107, Springer, 2022.

[74] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, “On the accuracy of spectrum-based fault localization,” in *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*, pp. 89–98, IEEE, 2007.

[75] M. Papadakis and Y. Le Traon, “Metallaxis-fl: mutation-based fault localization,” *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 605–628, 2015.

[76] S. Moon, Y. Kim, M. Kim, and S. Yoo, “Ask the mutants: Mutating faulty programs for fault localization,” in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pp. 153–162, IEEE, 2014.

[77] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, “Tbar: Revisiting template-based automated program repair,” in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 31–42, 2019.

[78] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. Le Traon, “You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated

program repair systems," in *2019 12th IEEE conference on software testing, validation and verification (ICST)*, pp. 102–113, IEEE, 2019.

[79] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, "A genetic programming approach to automated software repair," in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pp. 947–954, 2009.

[80] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *2009 IEEE 31st International Conference on Software Engineering*, pp. 364–374, IEEE, 2009.

[81] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen, "Automatic program repair with evolutionary computation," *Communications of the ACM*, vol. 53, no. 5, pp. 109–116, 2010.

[82] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, pp. 162–168, IEEE, 2008.

[83] A. Arcuri, "Evolutionary repair of faulty software," *Applied soft computing*, vol. 11, no. 4, pp. 3494–3514, 2011.

[84] V. Debroy and W. E. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *2010 Third International Conference on Software Testing, Verification and Validation*, pp. 65–74, IEEE, 2010.

[85] M. Nica, S. Nica, and F. Wotawa, "On the use of mutations and testing for debugging," *Software: practice and experience*, vol. 43, no. 9, pp. 1121–1142, 2013.

[86] C. Kern and J. Esparza, "Automatic error correction of java programs," in *International Workshop on Formal Methods for Industrial Critical Systems*, pp. 67–81, Springer, 2010.

[87] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 772–781, IEEE, 2013.

[88] S. Chandra, E. Torlak, S. Barman, and R. Bodik, "Angelic debugging," in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 121–130, 2011.

[89] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th international conference on software engineering*, pp. 691–701, 2016.

[90] F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus, "Automatic repair of buggy if conditions and missing preconditions with smt," in *Proceedings of the 6th international workshop on constraints in software testing, verification, and analysis*, pp. 30–39, 2014.

[91] S. R. L. Marcote and M. Monperrus, "Automatic repair of infinite loops," *arXiv preprint arXiv:1504.05078*, 2015.

[92] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 356–366, IEEE, 2013.

[93] L. Chen, Y. Pei, and C. A. Furia, "Contract-based program repair without the contracts," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 637–647, IEEE, 2017.

[94] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Towards practical program repair with on-demand candidate generation," in *Proceedings of the 40th International Conference on Software Engineering*, pp. 12–23, 2018.

[95] M. Martinez and M. Monperrus, "Astor: Exploring the design space of generate-and-validate program repair beyond GenProg," *Journal of Systems and Software*, vol. 151, pp. 65–80, 2019.

[96] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. Bissyandé, D. Kim, P. Wu, J. Klein, X. Mao, and Y. L. Traon, "On the efficiency of test suite based program repair: A systematic assessment

of 16 automated repair systems for java programs," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 615–627, 2020.

[97] H. Ye, M. Martinez, T. Durieux, and M. Monperrus, "A comprehensive study of automatic program repair on the quixbugs benchmark," *Journal of Systems and Software*, vol. 171, p. 110825, 2021.

[98] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 802–811, IEEE, 2013.

[99] S. H. Tan and A. Roychoudhury, "relifix: Automated repair of software regressions," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 471–482, IEEE, 2015.

[100] F. Logozzo and T. Ball, "Modular and verified automatic program repair," *ACM SIGPLAN Notices*, vol. 47, no. 10, pp. 133–146, 2012.

[101] F. Logozzo and M. Martel, "Automatic repair of overflowing expressions with abstract interpretation," *arXiv preprint arXiv:1309.5148*, 2013.

[102] Q. Gao, Y. Xiong, Y. Mi, L. Zhang, W. Yang, Z. Zhou, B. Xie, and H. Mei, "Safe memory-leak fixing for c programs," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 459–470, IEEE, 2015.

[103] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Thirty-First AAAI conference on artificial intelligence*, 2017.

[104] P. Muntean, V. Kommanapalli, A. Ibing, and C. Eckert, "Automated generation of buffer overflow quick fixes using symbolic execution and smt," in *International Conference on Computer Safety, Reliability, and Security*, pp. 441–456, Springer, 2014.

[105] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, "Fixminer: Mining relevant fix patterns for automated program repair," *Empirical Software Engineering*, vol. 25, no. 3, pp. 1980–2024, 2020.

[106] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pp. 101–114, 2020.

[107] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pp. 602–614, 2020.

[108] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 1161–1173, IEEE, 2021.

[109] M. Namavar, N. Nashid, and A. Mesbah, "A controlled experiment of different code representations for learning-based program repair," *Empirical Software Engineering*, vol. 27, no. 7, pp. 1–39, 2022.

[110] B. Lin, S. Wang, M. Wen, and X. Mao, "Context-aware code change embedding for better patch correctness assessment," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 3, pp. 1–29, 2022.

[111] H. Tian, Y. Li, W. Pian, A. K. Kabore, K. Liu, A. Habib, J. Klein, and T. F. Bissyandé, "Predicting patch correctness based on the similarity of failing test cases," *ACM Transactions on Software Engineering and Methodology*, 2022.

[112] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *Proceedings of the 40th international conference on software engineering*, pp. 789–799, 2018.

[113] D. Yang, Y. Lei, X. Mao, Y. Qi, and X. Yi, "Seeing the whole elephant: Systematically understanding and uncovering evaluation biases in automated program repair," *ACM Transactions on Software Engineering and Methodology*, 2022.

[114] S. Benton, Y. Xie, L. Lu, M. Zhang, X. Li, and L. Zhang, "Towards boosting patch execution on-the-fly," in *Proceedings of the 44th International Conference on Software Engineering*, pp. 2165–2176, 2022.

[115] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 263–272, 2005.

[116] C. Bartolini, A. Bertolino, S. Elbaum, and E. Marchetti, "Whitening soa testing," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 161–170, ACM, 2009.

[117] X. Bai, S. Lee, W.-T. Tsai, and Y. Chen, "Ontology-based test modeling and partition testing of web services," in *2008 IEEE International Conference on Web Services*, pp. 465–472, IEEE, 2008.

[118] C. Lenz, J. Chimiak-Opoka, and R. Breu, "Model driven testing of soa-based software," in *Proceedings of the Workshop on Software Engineering Methods for Service-Oriented Architecture (SEMSOA2007)*, pp. 99–110, Citeseer, 2007.

[119] H. Zhu and Y. Zhang, "Collaborative testing of web services," *IEEE Transactions on Services Computing*, vol. 5, no. 1, pp. 116–130, 2012.

[120] A. Barros, C. Ouyang, and F. Wei, "Static analysis for improved modularity of procedural web application programming interfaces," *IEEE Access*, vol. 8, pp. 128182–128199, 2020.

[121] A. Martin-Lopez, S. Segura, and A. Ruiz-Cortés, "Restest: Black-box constraint-based testing of restful web apis," in *International Conference on Service-Oriented Computing*, pp. 459–475, Springer, 2020.

[122] E. Viglianisi, M. Dallago, and M. Ceccato, "Resttestgen: automated black-box testing of restful apis," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pp. 142–152, IEEE, 2020.

[123] J. C. Alonso, A. Martin-Lopez, S. Segura, J. M. Garcia, and A. Ruiz-Cortes, "Arte: Automated generation of realistic test inputs for web apis," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 348–363, 2022.

[124] M. Bozkurt, M. Harman, and Y. Hassoun, "Testing and verification in service-oriented architecture: a survey," *Software Testing, Verification and Reliability*, vol. 23, no. 4, pp. 261–313, 2013.

[125] A. Tarhini, H. Fouchal, and N. Mansour, "A simple approach for testing web service based applications," in *International Workshop on Innovative Internet Community Systems*, pp. 134–146, Springer, 2005.

[126] P. M. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[127] H. Y. Huang, H. H. Liu, Z. J. Li, and J. Zhu, "Surrogate: A simulation apparatus for continuous integration testing in service oriented architecture," in *2008 IEEE International Conference on Services Computing*, vol. 2, pp. 223–230, IEEE, 2008.

[128] H. Liu, Z. Li, J. Zhu, H. Tan, and H. Huang, "A unified test framework for continuous integration testing of soa solutions," in *2009 IEEE International Conference on Web Services*, pp. 880–887, IEEE, 2009.

[129] L. Peyton, B. Stepien, and P. Seguin, "Integration testing of composite applications," in *Proceedings of the 41st Annual Hawaii International Conference on System Sciences (HICSS 2008)*, pp. 96–96, IEEE, 2008.

[130] 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*, vol. 11, no. 4, pp. 616–629, 2015.

[131] L. Leal, L. Montecchi, A. Ceccarelli, and E. Martins, "Exploiting mde for platform-independent testing of service orchestrations," in *2019 15th European Dependable Computing Conference (EDCC)*, pp. 149–152, IEEE, 2019.

[132] W. Bousanoh and T. Suwannasart, "Test case generation for ws-bpel from a static call graph," in *Journal of Physics: Conference Series*, vol. 1195, p. 012004, IOP Publishing, 2019.

[133] P. Nakngern and T. Suwannasart, "A design of ws-bpel test case generation tool based on path conditions," in *International MultiConference of Engineers and Computer Scientists*, 2017.

[134] E. Shamsoddin-Motlagh, "Automatic test case generation for orchestration languages at service oriented architecture," *International Journal of Computer Applications*, vol. 80, no. 7, 2013.

[135] L. Leal, A. Ceccarelli, and E. Martins, "The samba approach for self-adaptive model-based online testing of services orchestrations," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, pp. 495–500, IEEE, 2019.

[136] A. Arcuri, "RESTful API automated test case generation with EvoMaster," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 1, pp. 1–37, 2019.

[137] M. Zhang, B. Marculescu, and A. Arcuri, "Resource-based test case generation for restful web services," in *Proceedings of the genetic and evolutionary computation conference*, pp. 1426–1434, 2019.

[138] Y. Hammal, K. S. Mansour, A. Abdelli, and L. Mokdad, "Formal techniques for consistency checking of orchestrations of semantic web services," *Journal of Computational Science*, vol. 44, p. 101165, 2020.

[139] A. Arcuri and J. P. Galeotti, "Handling sql databases in automated system test generation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–31, 2020.

[140] E. de Jager and S. de Gouw, “Hybrid analysis of bpel models with grammars.,” in *SOFSEM (Doctoral Student Research Forum)*, pp. 73–84, 2020.

[141] L. Leal, L. Montecchi, A. Ceccarelli, and E. Martins, “Using metamodels to improve model-based testing of service orchestrations,” in *2020 IEEE 25th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pp. 130–139, IEEE, 2020.

[142] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, “Prioritizing test cases for regression testing,” *IEEE Transactions on software engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[143] L. Chen, Z. Wang, L. Xu, H. Lu, and B. Xu, “Test case prioritization for web service regression testing,” in *2010 Fifth IEEE International Symposium on Service Oriented System Engineering*, pp. 173–178, IEEE, 2010.

[144] S.-S. Hou, L. Zhang, T. Xie, and J.-S. Sun, “Quota-constrained test-case prioritization for regression testing of service-centric systems,” in *2008 IEEE International Conference on Software Maintenance*, pp. 257–266, IEEE, 2008.

[145] L. Mei, W. K. Chan, T. Tse, and R. G. Merkel, “Xml-manipulating test case prioritization for xml-manipulating services,” *Journal of Systems and Software*, vol. 84, no. 4, pp. 603–619, 2011.

[146] C. D. Nguyen, A. Marchetto, and P. Tonella, “Test case prioritization for audit testing of evolving web services using information retrieval techniques,” in *2011 IEEE International Conference on Web Services*, pp. 636–643, IEEE, 2011.

[147] S. Yoo and M. Harman, “Regression testing minimization, selection and prioritization: a survey,” *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[148] L. Mei, W. K. Chan, T. Tse, B. Jiang, and K. Zhai, “Preemptive regression testingof workflow-based web services,” *IEEE Transactions on Services Computing*, vol. 8, no. 5, pp. 740–754, 2014.

[149] S. Ji, B. Li, and P. Zhang, "Test case selection for all-uses criterion-based regression testing of composite service," *IEEE Access*, vol. 7, pp. 174438–174464, 2019.

[150] P. Godefroid, D. Lehmann, and M. Polishchuk, "Differential regression testing for rest apis," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 312–323, 2020.

[151] L. Chen, J. Wu, H. Yang, and K. Zhang, "Does pagerank apply to service ranking in microservice regression testing?," *Software Quality Journal*, vol. 30, no. 3, pp. 757–779, 2022.

[152] C.-a. Sun, M. Li, J. Jia, and J. Han, "Constraint-based model-driven testing of web services for behavior conformance," in *International Conference on Service-Oriented Computing*, pp. 543–559, Springer, 2018.

[153] N. Gupta, V. Yadav, and M. Singh, "Automated regression test case generation for web application: A survey," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–25, 2018.

[154] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," *IEEE Transactions on Software Engineering*, 2019.

[155] E. Rahm, "Towards large-scale schema and ontology matching," in *Schema matching and mapping*, pp. 3–27, Springer, 2011.

[156] R. Shraga, A. Gal, and H. Roitman, "Adnev: Cross-domain schema matching using deep similarity matrix adjustment and evaluation," *Proceedings of the VLDB Endowment*, vol. 13, no. 9, pp. 1401–1415, 2020.

[157] M. A. Zahid, B. Shafiq, J. Vaidya, A. Afzal, and S. Shamail, "Collaborative business process fault resolution in the services cloud," *IEEE Transactions on Services Computing*, pp. 1–1, 2021.

[158] "The world's most popular api testing tool — soapui." `https://www.soapui.org/`. (Accessed on 11/18/2021).

[159] O. Moser, F. Rosenberg, and S. Dustdar, "Viedame-flexible and robust bpel processes through monitoring and adaptation," in *Companion of the 30th International Conference on Software Engineering*, pp. 917–918, ACM, 2008.