Collaborative Business Process Fault Resolution in the Services Cloud

Muhammad Adeel Zahid*, Basit Shafiq* Jaideep Vaidya†, Ayesha Afzal‡ and Shafay Shamail*
*Department of Computer Science, Lahore University of Management Sciences, Lahore, Pakistan
Email: {muhammad.zahid, basit, sshamail}@lums.edu.pk
†MSIS Department, Rutgers University, Newark, NJ 07102
Email: jsvaidya@business.rutgers.edu
‡Computer Science Department, Air University Multan Campus, Multan, Pakistan
Email: ayesha@aumc.edu.pk

Abstract—The emergence of cloud and edge computing has enabled rapid development and deployment of Internet-centric distributed applications. There are many platforms and tools that can facilitate users to develop distributed business process (BP) applications by composing relevant service components in a plug and play manner. However, there is no guarantee that a BP application developed in this way is fault-free. In this paper, we formalize the problem of collaborative BP fault resolution which aims to utilize information from existing fault-free BPs that use similar services to resolve faults in a user developed BP. We present an approach 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 user developed BP. An extensive experimental evaluation over both synthetically generated faulty BPs and real BPs developed by users shows the effectiveness of our approach.

Index Terms—Fault resolution, Business processes, Web services.

1 Introduction

Cloud computing and Internetware software paradigm has 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 for developing distributed applications, but also changed the overall enterprise application development process.

However, BP applications developed using a plug and play approach are not guaranteed to be fault-proof. 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 [6].

In this paper, we address the problem of detecting and resolving faults in BPs that result in incorrect or unexpected output. Faults in a BP can be grouped into four broad categories listed in Table 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. [7] for fault injection in BPEL processes. Therefore, any design-time fault within a BP can be represented as combination of these mutation operators. Note that Estero-Botaro et al. [7] identified five different categories of mutation operators. Of these, 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.

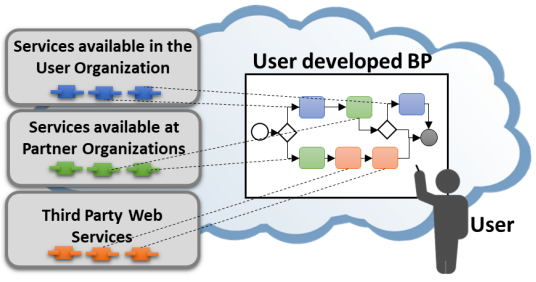


Fig. 1: Services cloud environment for BP composition.

The proposed approach enables detection and resolution of design-time faults in BPs in the services cloud environment depicted in Fig. 1. We consider each component service as a black box and focus only on the faults that may arise during service composition, which are categorized in Table 1. We do not address faults related to service implementation errors, service failures, service deployment, or network/communication issues, since these have already been

Manuscript received April 19, 2005; revised August 26, 2015.

TABLE 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 arithmatic 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

addressed in prior work. For example, component service implementation errors can be addressed in BP development using unit testing based approaches [8], [9]. Similarly, runtime 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].

In the services cloud environment of Fig. 1, we refer to a BP composed by a BP designer as *user BP*, which could be faulty. A user BP may include web services that are also used in BPs of other users. We refer to these other users' BPs that have one or more services common with the user BP as *existing BPs*. Our proposed approach 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 proposed approach as compared to the existing BP fault detection and debugging approaches, such as [14], [15], [16], [17], [18] that examine the faulty BP in isolation. Therefore, we refer to our proposed approach as a collaborative fault resolution approach. Essentially, our proposed approach performs 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.

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 in the BP. Once the BP designer is satisfied with the changes, the resulting BP is deployed.

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 related 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 organizations 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], [19], [20] to resolve differences in operation/attribute names before continuing with the fault resolution approach.

The main contributions of this paper are:

- We formalize the problem of Collaborative BP Fault Resolution that aims to resolve the faults in a user developed BP using information from existing 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.
- 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.

The rest of the paper is organized as follows. Section 2 discusses the related work in the literature. Section 3 presents the preliminaries and the problem statement. Section 4 presents the proposed approach. Section 5 presents the experimental evaluation. Finally, Section 6 concludes the paper and discusses future work.

2 Related Work

Most of the prior work focuses either on fault localization or on fault resolution, which we now discuss.

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 [18], [21]. Traditional fault localization techniques include program analysis based techniques [22] and dynamic analysis based techniques that analyze the run-time behavior of passed and failed executions. Among program analysis based techniques, program slicing is most extensively used [21]. Program slicing involves analyzing the runtime program behavior to identify suspicious slice (i.e., statement blocks that directly affect program output) and prune the program slices that do not correspond to the incorrect output [17], [23], [24]. Among dynamic analysis based techniques spectrum-based techniques are the most common. 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 [15], [17], [25]. Tarantula is one such technique that performs statement-level fault localization [26] and has been employed in the context of BPEL programs [15], [17].

Sun et al. proposed BPELswice [17] 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 a 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 [27] and delta-debugging for accurate fault localization in a more efficient manner.

In our approach we make use of statistical fault localization. However, our approach is agnostic to the underlying fault localization technique.

Fault resolution. There is significant work done on automated repair of programs written in Java and C/C++, though none of these works address program repair in BPs developed through web service orchestration. One widely used methodology for automated program repair is *generate-and-validate* (G&V) [18], [28], [29], [30], [31], [32], [33]. G&V takes as input a faulty program and a group of passing and failing tests, and heuristically searches the program space to generate fix candidates. The validity of the fix candidates 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 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.

Both fault localization and resolution approaches discussed above examine the faulty BP in isolation. Unlike these approaches, our proposed approach leverages the knowledge of existing BPs for fault diagnosis and resolution which is a novel and unique aspect of our work. Moreover, the existing approaches do not consider all types of faults that may arise in BPs. For example, BPELswice is not designed for faults caused by removal of activities/elements.

3 Preliminaries and Problem Statement

In this section, we will explain the formalism used to model a BP and provide a formal problem statement.

3.1 Notations and background

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.*

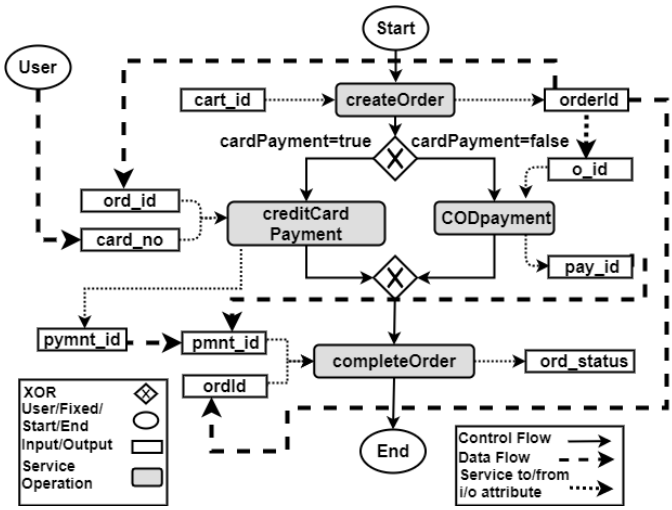


Fig. 2: Example of an e-commerce BP graph

Fig. 2 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. 2 are characterized as: i) control flow edges represented as solid arrows (e.g., edge from *createOrder* to *xor split*); and ii) dataflow edges. There are two types of dataflow edges. The first type, represented as arrow with dotted line, is drawn between a service operation and its input or output attribute (e.g., 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 arrow with dashed line that originates from output attribute of a service operation and terminates at 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*). 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 2.* 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 $u.type \in \{Service\ Operation, attribute\})$

3.2 Problem Statement

We now formally specify the problem statement. 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 removes the fault(s) and enables its correct execution with respect to the given set of test cases.

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

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

4 Proposed Approach

Our proposed approach exploits the knowledge of existing BPs for identification and resolution of faults in a user developed BP. Fig. 3 shows the different steps of the proposed approach. There are four main steps which 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 done (discussed in Section 4.1) 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), we perform pairwise comparison of a faulty BP with related 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 4.3) 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.

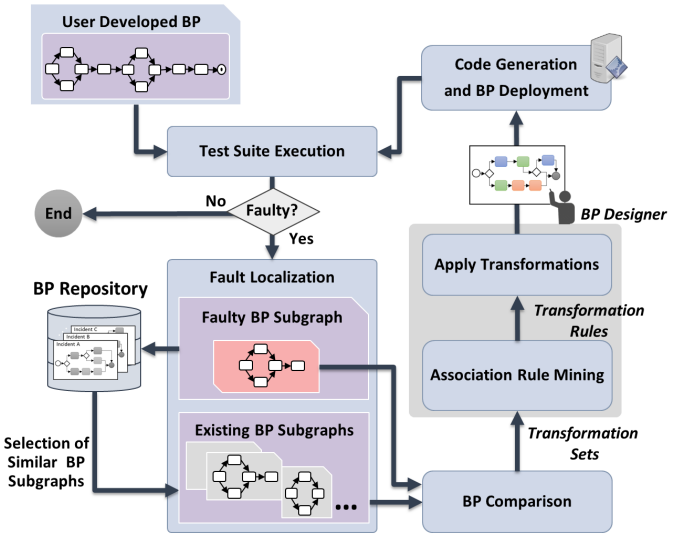


Fig. 3: Collaborative BP fault resolution approach

Algorithm 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 backwards 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.

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.

Our proposed approach 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 3, and described below.

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

4.1 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 [21], [34]. For localizing faults in a BP, we use the statistical debugging technique of Liu et al. [35]. 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 ones. Each predicate is assigned a fault-relevance score and the predicates are 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 run. 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:

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

ALGORITHM 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** $\mathcal{T}_i$

$distance(v_f^{start}, v) < distance(v_f^{start}, fop)\}$, and
- $E_{Sf} = \{(u, v) | u, v \in V_{Sf}$ and $(u, v) \in E_f\}$

4.2 Comparison with Existing BPs

We compare the subgraph S_f of the faulty BP with all the subgraphs of existing BPs that include 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 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 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 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.

TABLE 2: Results of pair-wise graph comparison for the BPs depicted in Fig. 4 and Fig. 5

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

As an example, consider a fragment of a faulty sales order BP G_f shown in rectangular box of Fig. 4. 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 BPs is depicted in the rectangular box in Fig.4. 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. 5(a), 5(b), and 5(c). The results of this comparison are shown in Table 2. 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.

4.3 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.

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 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

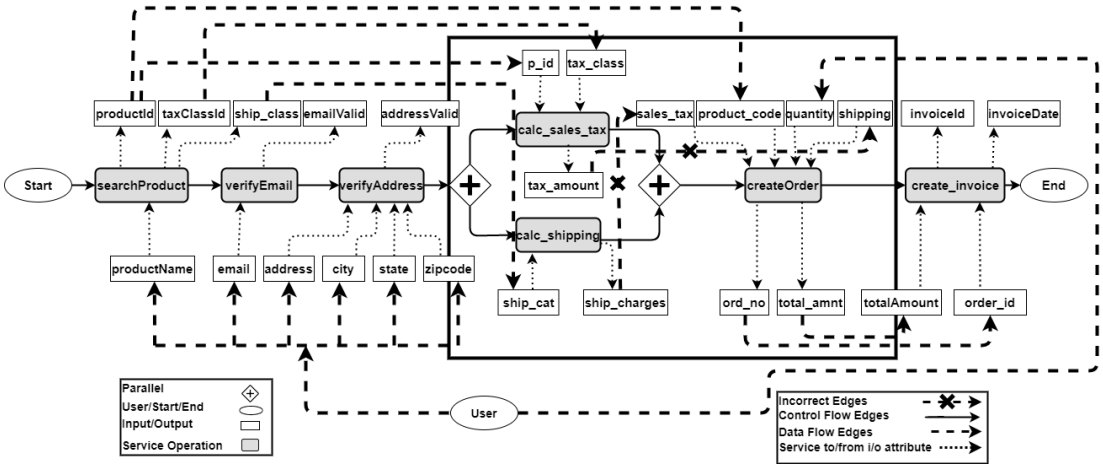
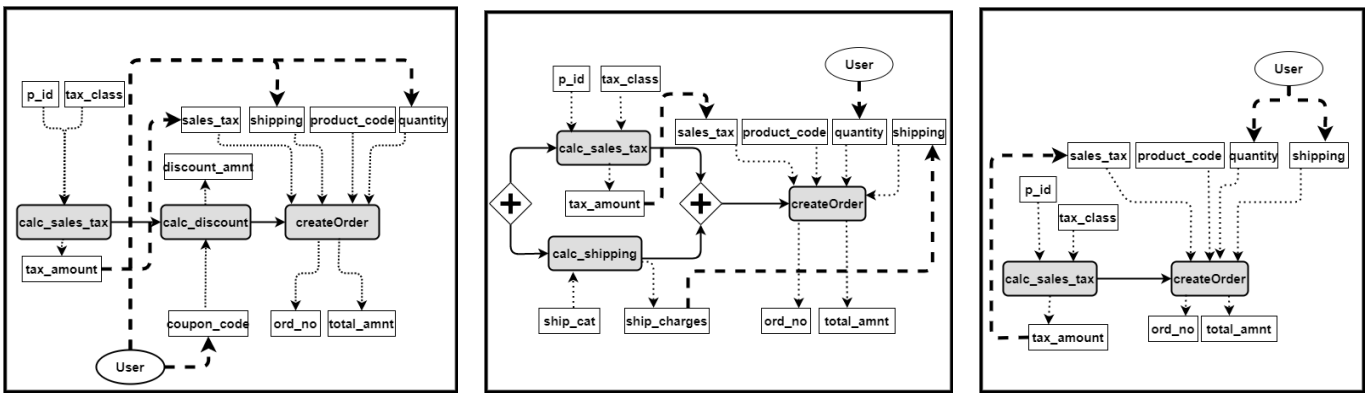


Fig. 4: Faulty sales order BP (G_f) and its subgraph (shown in rectangular box) used for pair-wise comparison



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

Fig. 5: Subgraphs of existing BPs used for comparison with the faulty BP

can automatically identify the relationships/associations between sets of transformations, it may be possible to identify the groups, and to 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 3. **Fault covering transformation set.** With respect to a given fop in a faulty BP (G_f), a fault covering transformation set ($\mathcal{F_C}$) 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 $\mathcal{F_C}$ to G_f, the resulting BP passes all the test cases that validate conditions up to the given fop.

The proposed approach employs association rule mining to first identify the relationship/associations between the transformations across different groups of BPs w.r.t. 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 $\mathcal{F_C}$ of G_f, if $X \cup Y$ contains $\mathcal{F_C}$ then X cannot contain $\mathcal{F_C}$, i.e., $\mathcal{F_C} \subseteq X \cup Y \implies \mathcal{F_C} \nsubseteq X$.

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. 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 2), 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 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 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, addition of such edge requires removal of some other edge from G_f for the same reason.

Based on the above discussion and following the graph comparison algorithm (Algorithm 2),we can deduce that:

$$\bigcap_i (V_i^r \cup E_i^r) \neq \phi \tag{1}$$

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

$$\bigcap_i (V_i^r \cup E_i^r) \subseteq \bigcap_i \mathcal{T}_i \neq \phi \tag{2}$$

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

$$\bigcap_i \mathcal{T}_i \subseteq \mathcal{F_C} \tag{3}$$

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 $\mathcal{F_C} \subseteq X \cup Y$, we can show that:

$$\bigcap_i \mathcal{T}_i \cap Y \neq \phi \tag{4}$$

and

$$\bigcap_i \mathcal{T}_i \cap Y \subseteq \mathcal{F_C} \tag{5}$$

and

Considering (3), (4), (5) and given that $\mathcal{F_C} \subseteq X \cup Y$ and $X \cap Y = \phi$, we can deduce that $\mathcal{F_C} \nsubseteq X$. □

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 a association rules given the exponential number of rules that can be generated from a given itemset.

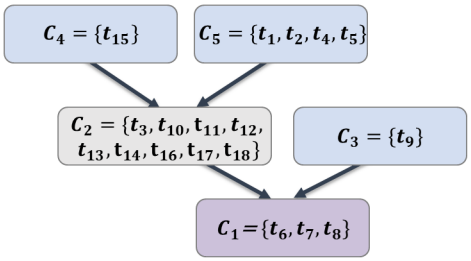


Fig. 6: SCC graph resulting from association analysis on transformations listed in Table 2

Our proposed approach discovers and searches the association rules for fault covering transformation set in a systematic and efficient manner. The specific steps of the proposed approach are listed in Algorithm 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. 6 shows a graph with 5 SCCs. This graph is generated using the transformations listed in Table 2 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. 6 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 source, sink, or internal. For example in Figure 6, 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 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 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 a path $C_4 \implies C_2 \implies C_1$ in Fig. 6, 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 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 3 and lines 10 - 14 of Algorithm 1).

The path ($C_4 \implies C_2 \implies C_1$) in Fig. 6 transforms the subgraph of faulty BP S_f into the subgraph

ALGORITHM 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** $tRuleQ$

S_3 of existing BP G_3 depicted in Fig. 5(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. 5(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. 5(b), thus resolving G_f with minimal changes.

4.4 Computation complexity

Graph Comparison (Algorithm 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 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, 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 3 is $O(m^2) = O(n^2(|V| + |E|)^2)$.

Fault Resolution (Algorithm 1). The algorithm calls Algorithm 2 n times and Algorithm 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 fault resolution algorithm for a fixed search region is $O(n^2(|V| + |E|)^2)$.

5 Experimental Evaluation

We have performed an extensive experimental evaluation of the proposed approach. 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 resolve the faults injected. In the other case, we asked actual users to develop BPs using a BP composition tool. We 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 3 shows the detailed statistics of the developed BPs.

TABLE 3: Statistics of existing BPs

Domain	No. of BPs	Avg. no. of services	Avg. no. of branches
Ecommerce	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

TABLE 4: 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

parallelization was used (i.e., only one core was used and no significant amount of memory was used).

5.1 Random Fault Injection

As mentioned above, in this case, we started with a correct BP from the insurance domain (not included in 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. 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 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 generated.

After determining the fops, we ran Algorithm 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 the Table 4. 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 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 standard deviation of 6.07.

Table 5 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.

In terms of the computation time overhead associated with the proposed fault resolution approach, Fig. 7(a) shows the percentage of BPs fixed vs. number of iterations. Overall, 65% of the total BPs were fixed in 7 or fewer iterations.

TABLE 5: 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

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 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. 7(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. 7(a) and Fig. 7(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 4.3. 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:

$$Similarity = 1 - \left(\frac{1}{|M|} \sum_{G_i \in M} \frac{|\mathcal{T}_i|}{\max(|E_f| + |V_f|, |E_i| + |V_i|)} \right) \quad (6)$$

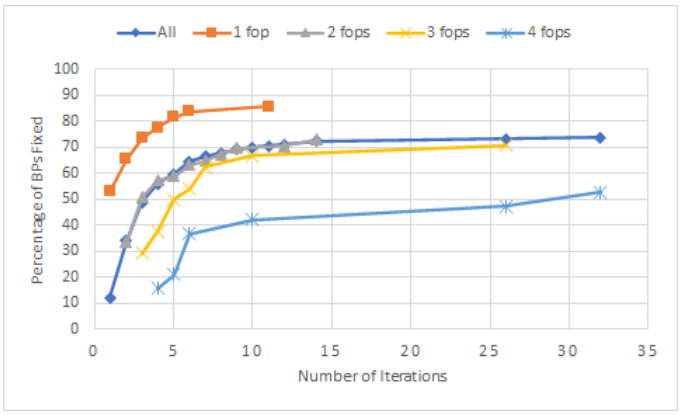
Similarly, the service overlap can be computed as the average of the number of common services between the faulty BP G_f and each $G_i \in M$, normalized for each G_i:

$$Overlap = \frac{1}{|M|} \sum_{G_i \in M} \frac{|V_f \cap V_i|}{\max(|V_f|, |V_i|)} \quad (7)$$

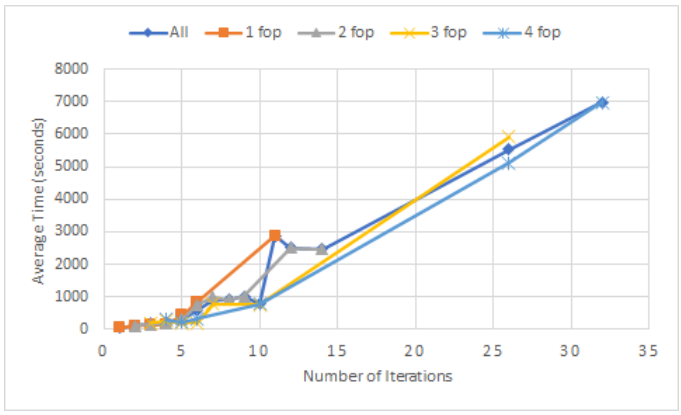
As per these definitions, it is highly unlikely to have a high service overlap but low similarity. Fig. 8(a) and Fig. 8(b) show the number of BPs fixed vs. similarity and service overlap, respectively. 68% of all the BPs were fixed at an average similarity of 0.43 or less. Moreover, less than 10% of 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.

5.1.1 Comparison with baseline approach

We would also like to establish the effectiveness of our approach with respect to an existing baseline. However, there

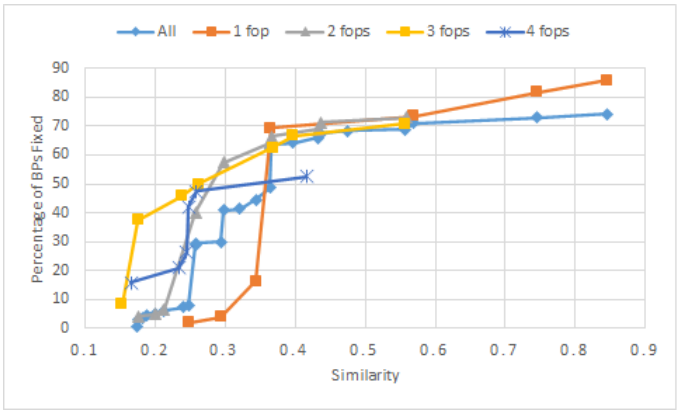


(a) Percentage of BPs fixed vs. number of iterations

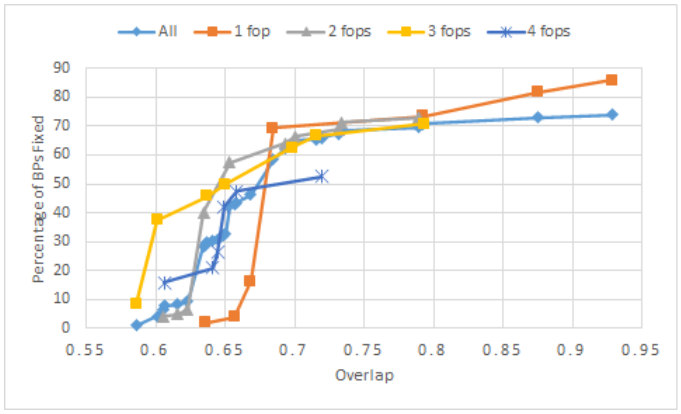


(b) Average time taken to fix a BP vs. number of iterations

Fig. 7: Comparison between percentage of BPs fixed vs. number of iterations and iteration-wise average time taken for fault resolution per BP



(a) Percentage of fixed BPs vs. similarity



(b) Percentage of fixed BPs vs. service overlap

Fig. 8: Comparison between percentage of BPs fixed vs. similarity and service overlap based on top 3 rules

TABLE 6: Accuracy comparison of proposed approach with G&V fault repair approach

Fault Category	No. of BPs	G&V with BPELswice		Proposed Approach	
		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

is no existing solution for BPEL fault resolution that we can directly compare with. Therefore, we compare the proposed collaborative BP fault resolution approach with 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 fix candidates. The validity of the fix candidates 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 [17] to identify suspicious BPEL statements. For generating the candidate fixes, we applied the mutation operators (listed in Table 1) to the suspicious statements.

As mentioned in [17], BPELswice is not designed for faults caused by 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 6. For variable assignment and control flow fault categories (excluding activity/element removal), G&V achieves higher accuracy than the proposed approach. For

expression faults both G&V and proposed approach 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 the proposed approach for all three categories. This clearly shows the efficiency of the proposed approach 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 include 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, the proposed approach resolves faults by applying very few transformation rules (candidate fixes).

The last three rows of Table 6 shows the accuracy results of the proposed approach 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 combination of multiple faults from different categories. Our proposed approach achieves high accuracy in resolving branching and multiple faults. Overall, out of the 208 faulty BPs, our proposed approach was able to fix 153 BPs resulting in an accuracy of 0.73.

These results clearly show that the proposed approach 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 the proposed approach w.r.t. variable assignment faults and control flow faults (caused by activity removal) is low as compared to other faults 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. We plan to investigate a hybrid approach to improve the accuracy for these cases as well in the future.

5.2 User Developed BPs

For this evaluation, we applied our fault resolution approach 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 ecommerce sales, insurance sales, and flight reservation. We selected those BPs that did not execute correctly. The users were not aware that their developed BPs would be used for 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 ecommerce 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 7 lists all user developed BPs, with the type of faults that were made by the users and the results of our approach.

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 semantic, as well as the extensive debugging that they perform.

5.3 Parameter sensitivity

We now discuss the key parameters that affect the performance of the proposed approach 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 pairwise 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, increasing α results in 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 comparison of the faulty BP with unrelated BPs, which also results in 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 satisfaction of the BP designer.

The accuracy of the proposed approach 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. 8. 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 3, 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 are above 0.4 and 0.7 respectively, then the likelihood of fault resolution increases significantly.

6 Conclusions and Future Work

In this paper, we have formalized the problem of collaborative BP fault resolution 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. An extensive experimental evaluation over both synthetically generated faulty BPs and real BPs developed by users shows the effectiveness of our approach.

One limitation of our approach is that it only works if there is no syntactic or semantic heterogeneity between common services across BPs. Furthermore, we also assume that the existing BPs are fault-free. In the future, we plan

TABLE 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

to relax both of these assumptions, and develop solutions that can take into account potential heterogeneity and work even when the existing BPs are not completely fault-free. Our current work also does not take privacy into account in that it assumes that complete knowledge of the existing BPs is available. In the future, we also plan to extend the proposed approach taking into account privacy concerns.

ACKNOWLEDGMENTS

Research reported in this publication was supported by the HEC and Planning Commission of Pakistan and LUMS FIF grant as well as the NSF (CNS-1624503, CNS-1747728) and the NIH (R01GM118574, R35GM134927). The content is solely the responsibility of the authors and does not necessarily represent the official views of the agencies funding the research.

REFERENCES

[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] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*, 2018.

[7] 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.

[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*. IEEE, 2009, pp. 326–335.

[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)*. IEEE, 2018, pp. 802–807.

[15] C. ai 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.

[16] 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.

[17] 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.

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

[20] 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.

[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] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "Jfix: semantics-based repair of java programs via symbolic pathfinder," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 376–379.

[23] 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.

[24] E. Soremekun, L. Kirschner, M. Böhme, and A. Zeller, "Locating faults with program slicing: an empirical analysis," *Empirical Software Engineering*, vol. 26, no. 3, pp. 1–45, 2021.

[25] P. Li, M. Jiang, and Z. Ding, "Fault localization with weighted test model in model transformations," *IEEE Access*, vol. 8, pp. 14 054–14 064, 2020.

[26] 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*, 01 2005, pp. 273–282.

[27] 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)*. IEEE, 2021, pp. 25–34.

[28] 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)*. IEEE, 2013, pp. 356–366.

[29] 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)*. IEEE, 2017, pp. 637–647.

[30] 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*, 2018, pp. 12–23.

[31] 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.

[32] 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*, 2020, pp. 615–627.

[33] 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.

[34] 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.

[35] 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.

Muhammad Adeel Zahid is a PhD candidate in the Department of Computer Science at Lahore University of Management Sciences, Pakistan. He is also serving as a Lecturer in Computer Science at Government College University Faisalabad, Pakistan. His research interests are in the areas of distributed systems and service-oriented computing.

Basit Shafiq is an Associate Professor and Chair of the Department of Computer Science in the Syed Babar Ali School of Science and Engineering at Lahore University of Management Sciences, Pakistan. He has published over 70 research papers in international conferences and journals. His research interests are in the areas of distributed systems, security and privacy, semantic Web and Web services. He is on the editorial board of IEEE Transactions on Dependable and Secure Computing, Computers and Security, and ACM Digital Government: Research and Practice.

Jaideep Vaidya is a Professor of Computer Information Systems at Rutgers University and the Director of the Rutgers Institute for Data Science, Learning, and Applications. He has published over 190 papers in international conferences and journals. His research interests are in privacy, security, and data management. He is an ACM Distinguished Scientist and IEEE Fellow. He is the Editor-in-Chief of the IEEE Transactions on Dependable and Secure Computing.

Ayesha Afzal is an Assistant Professor at Air University, Multan, Pakistan. Her research interests are in the areas of distributed systems, business process management and big data analytics.

Shafay Shamail is a Professor of Computer Science in the Syed Babar Ali School of Science and Engineering at Lahore University of Management Sciences, Pakistan. He has worked both in software industry and academia. His research interests include software quality, cloud computing and e-government architectures. His publications include over 60 research papers.