HHS Public Access

Author manuscript

*IEEE Int Conf Web Serv*. Author manuscript; available in PMC 2023 September 16.

Published in final edited form as:

*IEEE Int Conf Web Serv*. 2022 July ; 2022: 266–275. doi:10.1109/icws55610.2022.00048.

# An Integrated Framework for Fault Resolution in Business Processes

**Muhammad Adeel Zahid**,

**Ahmed Akhtar**,

**Basit Shafiq**,

**Shafay Shamail**

Computer Science Department, Lahore University of Management Sciences, Lahore, Pakistan

**Ayesha Afzal**,

Computer Science Department, Air University, Multan, Pakistan

**Jaideep Vaidya**

MSIS Department, Rutgers University, Newark, USA

## Abstract

Cloud and edge-computing based platforms have enabled rapid development of distributed business process (BP) applications in a plug and play manner. However, these platforms do not provide the needed capabilities for identifying or repairing faults in BPs. Faults in BP may occur due to errors made by BP designers because of their lack of understanding of the underlying component services, misconfiguration of these services, or incorrect/incomplete BP workflow specifications. Such faults may not be discovered at design or development stage and may occur at runtime. In this paper, we present a unified framework for automated fault resolution in BPs. The proposed framework employs a novel and efficient fault resolution approach that extends the generate-and-validate program repair approach. In addition, we propose a hybrid approach that performs fault resolution by analyzing a faulty BP in isolation as well as by comparing 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.

**Index Terms—**

Fault resolution; business processes; web service compositions

## I. Introduction

Cloud and edge computing infrastructure facilitates rapid development of Internet-centered distributed applications, including distributed workflows, web mashups, and business processes. These applications are developed using data, storage, and computation web

muhammad.zahid@lums.edu.pk .

services available in enterprise networks and cloud data centers, as well as large number of IoTs and edge devices. Several new plug-and-play-based tools and platforms are now available [1]–[4] that support distributed application development in an automated or semi-automated manner by composing relevant service components. While these tools and platforms support efficient development of distributed applications, they lack support for identifying or repairing faults in distributed applications, specifically business process (BP) applications [5]. Faults in BP may occur due to errors made by BP designers because of their lack of semantic understanding of the underlying web services, misconfiguration of these services, or incorrect/incomplete BP workflow specifications. Such faults may not be discovered at design or development stage and may occur at runtime.

In this paper, we focus on detecting and resolving faults in BP applications that may cause unexpected or incorrect output. In [5], BP faults have been categorized in four main categories given in Table I. This categorization is derived from the fault categories and mutation operators defined by Estero-Botaro et al. [6] for fault injection in BPEL processes. The different design-time faults in a BP can be represented as combination of mutation operators given in Table I. For fault detection and resolution in BPs, we consider each web service as a black box and focus on those faults that may occur during service composition. Although faults in a BP may also occur due to service failures, service implementation errors, service deployment, or network/communication issues, we do not consider these faults in this paper. There is a significant body of work in the literature addressing service implementation errors [7], [8], service failure and unavailability, service deployment issues [9], and network failure [10]–[12].

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. 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 [13], [14]. G&V requires a fault localization mechanism for identifying suspected code blocks in a faulty program. Candidate fixes are created by applying mutations to the elements contained in these suspected code blocks. 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. In addition, we propose a hybrid approach that combines the EGV with a collaborative fault resolution (CFR) approach presented in [5]. CFR performs fault resolution by comparing a faulty BP with existing fault-free BPs that use similar services. This results in improved accuracy and broader coverage of fault types.

The key contributions of this work include:

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 [13]–[16] but it has not been adapted for automatic resolution of faults in BPs encoded in BPMN or BPEL.
2. *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 [5] 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.
3. We extend an existing framework for automated BP composition and management in a services cloud environment [1] 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 automatic resolution of faults.

The rest of the paper is organized as follows: Section II provides some basic definitions and problem statement; Section III discusses the proposed G&V based fault resolution approach and its extension to a hybrid approach; Section IV discusses the experimental evaluation results; Section V discusses the implementation details of the proposed integrated framework for fault resolution; Section VI presents related work in the problem domain and Section VII concludes the paper.

## II. Preliminaries and Problem Statement

This section outlines the basic notation to represent a BP and an illustrative example that will be used in the following sections to explain the proposed approach.

*Definition 1:* (**Business Process**) A business process (*BP*) is defined as a graph, $G = (V, E, \mathscr{E}, v^{start}, v^{end}, v^{user})$ where:

- $V$ is the vertex set which is partitioned into the following vertex types; (i) service operations; (ii) input/output attributes of service operations; and (iii) XOR splits and joins;
- $E \subseteq V \times V$ is the edge set in $G$ denoting the data flow and control flow.
- $\mathscr{E}$ = {true, false, Boolean Expression}, for each edge.
- $v^{start}$ denotes the start activity in BP.
- $v^{end}$ denotes the terminating activity in BP.

- $v^{user}$ denotes a user vertex which is linked to all the input attributes whose values must be supplied by the BP user during execution.

**Illustrative Example:**

Fig. 1 shows a small BP graph representing an elementary BP from the e-commerce domain. Shaded gray boxes represent activities corresponding to 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. 1, is a faulty BP where the edge from *ship_charges* to *sales_tax* and the edge from *tax_amount* to *shipping* (marked with ×) correspond to the incorrect attribute assignments.

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

**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, $\mathscr{C} = \{G_{c1}, G_{c2}, \ldots, G_{ck}\}$, 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 paper. We assume that the test cases are either provided by the user or existing test case generation techniques [17]–[21] can be applied for this purpose as discussed in Section VI.

## III. Proposed Fault Resolution Approach

In this section, we first present an efficient G&V (EGV) approach for fault resolution in BPs. EGV extends the traditional G&V approach which is widely used for repairing C and Java programs. For BP fault resolution the proposed EGV approach leverages mutation-based fault localization in combination with program slicing to improve localization accuracy and consider a minimal set of suspicious code blocks for generating candidate fixes. In addition, we propose a hybrid approach that combines the EGV with a collaborative fault resolution (CFR) approach presented in [5]. CFR performs fault resolution by comparing a faulty BP with existing fault-free BPs that use similar services. This results in improved accuracy and broader coverage of fault types.

### A. Efficient G&V approach (EGV) for fault resolution

Fig. 2 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 which 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 III-A1) to identify location(s) where faults are observed in the 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 b present at a location prior to *fop*.

Specifically, we perform statistical fault localization [22] to identify *fop*. For a given *fop*, we perform program slicing [23] 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 to keep the generated number of candidate fixes to a minimum 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 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 process is repeated in an iterative manner until all the faults are resolved or all the candidate fixes are exhausted.

Algorithm 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 III-A1 and III-A2. 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 – 13). Candidate fixes are generated by successively applying mutation operators and their combinations on the input BP $G_f$. Each candidate fix, $G_c$ is validated to check if it passes the failed test cases up to the computed *fop* (Line 17). If any such $G_c$ is found, it is tested against all the test cases for the entire BP (Line 18). 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 19). Otherwise, we recursively call the EGV fault resolution procedure (Line 21) until all the faults are fixed or the entire space of mutants is exhausted.

We provide detailed description of each component of Fig. 2 below.

**1) Fault Localization:** Fault localization aims to locate and isolate faulty software components or bugs to determine the likely causes errors or software failures [24], [25]. For fault localization in a BP, we employ a statistical analysis-based debugging approach [22]. 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 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. 1, *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.

**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 [23]. 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. 1 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 control flow and the service operations and their attributes that provide input to *createOrder*. Fig. 3 depicts the slice of BP shown in the illustrative example in Fig. 1. Fig. 3 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.

**3) Candidate Fix Generation:** After identifying the filtered slice before *fop*, we generate candidate fixes by applying different mutation operators shown in Table I. 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.

**ALGORITHM 1:** EGV_FaultResolution

**Require:** $G_f = (V_f, E_f, \mathcal{E}_f, v_f^{start}, v_f^{end}, v_f^{user})$ - faulty BP
**Require:** $T$ - Set of test cases
**Ensure:** $G_c$ - Corrected BP

```
fop ← localize_fault(G_f)
slice ← get_process_slice(G_f, fop)
S ← φ
for each vertex v ∈ slice do
    if distance(v, v_f^start) ≤ distance(fop, v_f^start) then
        S ← S ∪ v
    end if
end for
for each edge (u, v) ∈ slice do
    if distance(u, v_f^start) ≤ distance(fop, v_f^start) or
    distance(v, v_f^start) ≤ distance(fop, v_f^start) then
        S ← S ∪ (u, v)
    end if
end for
C ← generate_candidate_fixes(S)
for each G_c ∈ C do
    Apply each test case t ∈ T on G_c
    if G_c fixes faults up to fop then
        if G_c passes all the test cases for the entire BP
        then
            return G_c
        else
            return EGV_FaultResolution(G_c, T)
        end if
    end if
end for
return NULL
```

In the slice shown in Fig. 3 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. The generated candidate fixes are shown in Table II. In $m_1$, *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. Similarly, *sales_tax* or *shipping* is not mapped to any of *productId*, *taxClassId* or *ship_cat* because they belong to a different data type.

**ALGORITHM 2:** Hybrid_H1

**Require:** $G_f = (V_f, E_f, \mathcal{E}_f, v_f^{start}, v_f^{end}, v_f^{user})$ - Faulty BP
**Require:** $T$ - Set of test cases
**Require:** $\mathcal{B} = \{G_1, G_2, \ldots G_n\}$ - Set of existing BPs
**Ensure:** $G_c$ - Corrected BP

```
1: fop ← localize_fault(G_f)
2: S ← subgraph_till_fop(G_f, fop)
3: G_c ← EGV_FaultResolution(S, T)
4: if G_c ≠ NULL then
5:    G_c ← combine_graph(G_f, S, G_c)
6:    Apply each test case t ∈ T on G_c
7:    if G_c passes all the test cases then
8:       return G_c
9:    else
10:      return Hybrid_H1(G_c, T)
11:   end if
12: else
13:   G_c ← CFR(S, T, B)
14:   if G_c ≠ NULL then
15:      G_c ← combine_graph(G_f, S, G_c)
16:      Apply each test case t ∈ T on G_c
17:      if G_c passes all the test cases then
18:         return G_c
19:      else
20:         return Hybrid_H1(G_c, T)
21:      end if
22:   else
23:      return NULL
24:   end if
25: end if
```

**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 w.r.t. the given test suite. If such a fix is found, the fault-free BP after application of the candidate fix is returned to the user. If a candidate fix passes some, but not all, of the test cases that were originally failed, we use that candidate fix for further discovery and repair of faults (Algorithm 1, Line 21).

## B. Hybrid Approach

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 [5] 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 fault in a faulty BP. However CFR requires a 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 allows for broader coverage of fault types by leveraging knowledge from existing BPs.

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 2 outlines the hybrid fault resolution approach, H1. The inputs to this algorithm are faulty BP graph, $G_f$, a set of test cases, $T$, and the set of existing BPs, $\mathscr{B}$. 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 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 were 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 13). Similar to the steps given in lines 4 – 11, we repeat these steps for CFR in lines 12 – 23. 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 2 except that it invokes CFR first followed by EGV.

## IV. Experimental Evaluation

We have performed extensive experiments to evaluate the performance of the proposed EGV and hybrid fault resolution approaches.

### Dataset:

For experimental evaluation, we consider the same dataset that was used for validation of CFR approach in [5]. This dataset consists of 48 fault-free BPs from three domains including, insurance, E-commerce, and flight reservation. For generation of faulty BPs, we used random fault injection to a correct BP from insurance domain which is not considered in above mentioned 48 BPs. Specifically, we applied random combinations of the mutation operators given in Table I to generate 208 faulty BPs which are considered for validation of the proposed approach. For generating a faulty BP, the number of mutation operators applied were between 1 and 4. The mean of the number of mutation operators applied was 2.24 and standard deviation was 0.81. The resulting faulty BPs covered all the fault categories (depicted in Table I) and their random combinations.

For comparison with CFR as well as evaluation of hybrid approach, we built a repository of existing BPs including all 48 fault-free BPs.

### EGV:

For comparison of EGV with basic G&V, we performed experiments on 40 BPs as shown in Table III. 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 BP whereas EGV only uses 10 candidate fixes. The accuracy of EGV is quite low (0.42) for variable assignment fault types as 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.

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*). As discussed in Section III-B EGV is not designed for resolving fault types in which one or more BP elements are removed. Table IV 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. In terms of execution time 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 increase significantly with the increase in *fops*.

### Hybrid Approach:

Hybrid approach combines both EGV and CFR in an interleaved fashion. As discussed in Section III-B, we consider two versions of hybrid approach, H1 and H2.

Table V 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. 5 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. 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.

## A. Discussion

As depicted in Table III, EGV significantly improves the performance in terms of number of candidate fixes by reducing the search space. However, the overall accuracy of EGV is lower than the basic G&V approach.

EGV also outperforms CFR in terms of computation time with comparable accuracy but CFR provides a broader fault coverage as discussed in Section III-B. For example, CFR can resolve faults belonging to branch 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 V. 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* increase. Consequently, the hybrid approach is preferred over CFR for better accuracy especially when 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 higher number of *fops* H2 performs better than H1.

## V. Implementation

We have developed a prototype implementation of the proposed framework for the fault resolution. For prototype implementation we have used Java (J2EE), BPEL and Apache ODE. The user (BP designer/developer) interacts with our system using a command line interface to provide inputs, including faulty BP and a set of test cases. In addition, we have also developed a repository of different BPs in various domains that can be used for fault resolution using CFR and hybrid approach.

After receiving the required inputs, the prototype invokes the selected fault resolution algorithm and checks the resulting BP for correctness by applying all test cases provided by the user. During this fault resolution process, the user is kept in the loop for reviewing the relevant candidate fixes and the resulting BP to ensure that the domain and the scope of the BP are not changed. Once the user is satisfied with the fixed BP, it is deployed to Apache ODE server and its associated user interface (UI) is deployed on Apache Tomcat web server for further user testing.

## VI. Related Work

We discuss the related work in the context of test case generation, fault localization, and fault resolution.

### Test case generation.

Test case generation is an essential step towards automated testing of programs. Significant work has been done on automated testing and test case generation in the context of BPs [17]–[21]. Bartolini et al. [17] presented an approach that uses test services for white-box testing of third-party services and a tool, WS-TAXI, for automated generation of test cases from the WSDL definition of web services. Tarhini et al. [26] addressed the problem of test case generation for service compositions and validation of service compositions using WSDL definitions. Sun et al. [27] proposed a scenario-oriented framework for test case generation for web service compositions in BPEL. BP graph model is considered to derive

test scenarios based on which test cases are generated. A constraint-based technique is employed for test data generation for execution of generated test cases.

### Fault localization.

The purpose of fault localization is to identify locations (branches, statements, or blocks) in a faulty program that are suspicious i.e., likely to be erroneous and associated with the fault. The objective is to support programmers fix the faults by focusing only on the suspicious locations and to support automated repair and recovery of the given program [13], [25]. Existing works on fault localization techniques can be categorized into static and dynamic analysis-based techniques [25]. The static techniques mostly rely on model checking and static program analysis. Dynamic techniques compare and contrast the runtime behavior of correct and incorrect executions to isolate the fault-relevant locations.

Program slicing is one of the most frequently used techniques [25] that analyzes the runtime profile of a program to isolate the statements and blocks that are fault-relevant. Fault-relevant slice is obtained by filtering out the program slice that does not correspond to incorrect output(s) [23], [28], [29]. The most common fault localization techniques are spectra-based and employ dynamic analysis and coverage information of test cases to assign suspicious score to program elements. The program elements are then ranked by this score and programmers are expected to examine these elements in the ranked order [23], [30], [31]. Tarantula [32] is one representative technique in this category that has also been adapted in the context of business processes [23]. BPELSwice [23] targets the fault localization problem for business processes developed using BPEL. It uses predicate switching and program slicing to isolate the suspicious statements with greater precision. BPELSwice significantly reduces the size of the slice by acting at the statement level and identifying only suspicious statements from within BPEL blocks rather than marking the whole block as fault-relevant. Our proposed EGV approach employs the BPELswice approach for identifying suspicious code blocks in the faulty BP.

Delta debugging [9] works by applying small changes (deltas) to the programs and monitors their effect on the output. It can be used to discover development, deployment, and configuration faults. Delta debugging has also been adapted in the context of microservices [9] to unveil deployment and configuration faults as well as the faults that occur in execution orders of microservices. Delta debugging has also been used in combination with mutation testing [33] for accurate fault localization.

In our proposed EGV fault resolution approach, we use a combination of statistical fault localization and program slicing. However, our approach is independent of the underlying technique used for fault localization.

### Fault resolution.

Considerable work has been done on automated program repair for software programs developed in C/C++ and Java but the existing approaches do not address automated repair in the context of BPs. Generate-and-validate (G&V) is a widely used technique [13]–[16] for automated repair and recovery 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 that explores the search space of a program and finds candidate fixes by applying variety of changes heuristically. G&V requires a fault localization mechanism for identifying suspected code blocks in a faulty program. Candidate fixes are computed by applying mutations to the elements contained in these suspected code blocks. These candidates are then validated against the available test suite. Xu et al. have proposed RESTORE which is an efficient G&V-based framework focusing repair of Java programs [13]. RESTORE first performs fault localization to compute suspicious program snapshots. Candidate fixes are then generated for every suspicious snapshot by taking into account different mutations of the given program. To address the efficiency issue, only selected candidate fixes are validated by executing test cases in order of their suspicion scores. In case no candidate fix passes the entire test suite, partially successful fixes passing a subset of test cases are considered for generating variants of the given faulty program. All steps are then repeated for the generated variant until valid program fixes are identified. This retrospective strategy improves the efficiency of G&V by avoiding exhaustive validation of all candidate fixes.

In our recent work, we proposed the Collaborative Fault Resolution (CFR) approach [5] for BPs. CFR relies on a set of existing BPs that are composed of similar services and are assumed to be correct. Rather than examining the faulty BP in isolation, CFR makes use of the knowledge of these BPs to discover and fix faults in a faulty BP. However CFR requires a certain minimum overlapping services between the faulty BP and existing BPs for providing accurate results. It uses statistical fault localization to identify the *fop*s in the given BP and then performs a pairwise analysis of the given faulty BP in comparison with related BPs of other users to identify their structural and semantic differences. Association rule mining is applied to these differences to identify the minimal overlapping set of differences that can be used to fix the faults. In this work, we have extended our proposed EGV approach with CFR for higher coverage of fault types as discussed in Subsection III-B.

## VII. Conclusion

In this paper, we addressed the problem of fault resolution in distributed BPs developed by composing web services. We have proposed an efficient fault resolution approach for BPs by extending the traditional generate-and-validate (G&V) automated program repair methodology. We have also proposed a hybrid approach that combines the proposed efficient G&V-based approach with a collaborative fault resolution approach for improved accuracy and broader coverage of fault types. Our extensive experimental evaluation results validate the effectiveness of the proposed approach. In addition, we have presented the prototype implementation of an integrated framework for fault diagnosis and resolution in BPs to demonstrate the viability of the proposed approach.

## Acknowledgment

The work of Basit Shafiq is supported by the HEC and Planning Commission of Pakistan and LUMS FIF grants. The work of Jaideep Vaidya is supported by the NIH grants (R35GM134927, R01GM118574). 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]. Afzal A, Zahid MA, Akhtar A, Shafiq B, Shamail S, Elahraf A, Vaidya J, and Adam N, "BP-Com: A service mapping tool for rapid development of business processes," in 2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS). IEEE, 2020, pp. 1235–1238.

[2]. Yao L, Wang X, Sheng QZ, Benatallah B, and Huang C, "Mashup recommendation by regularizing matrix factorization with API co-invocations," IEEE Transactions on Services Computing, 2018.

[3]. Microsoft, "Microsoft Flow - Power Automate," https://flow.microsoft.com/, 2022 (Accessed 2022-06-11).

[4]. Outsystems, "Outsystems- Low Code Platform," https://outsystems.com/, 2022 (Accessed 2022-06-11).

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

[6]. Estero-Botaro A, Palomo-Lozano F, Medina-Bulo I, Domínguez-Jiménez JJ, and García-Domínguez A, "Quality metrics for mutation testing with applications to ws-bpel compositions," Software Testing, Verification and Reliability, vol. 25, no. 5–7, pp. 536–571, 2015.

[7]. Chaturvedi A and Binkley D, "Web service slicing: Intra and inter-operational analysis to test changes," IEEE Transactions on Services Computing, 2018.

[8]. Qiu D, Li B, Ji S, and Leung H, "Regression testing of web service: a systematic mapping study," ACM Computing Surveys (CSUR), vol. 47, no. 2, pp. 1–46, 2014.

[9]. Zhou X, Peng X, Xie T, Sun J, Ji C, Liu D, Xiang Q, and He C, "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, 2019, pp. 683–694.

[10]. Yu J, Sheng QZ, Swee JK, Han J, Liu C, and Noor TH, "Model-driven development of adaptive web service processes with aspects and rules," Journal of Computer and System Sciences, vol. 81, no. 3, pp. 533–552, 2015.

[11]. Marrella A, Mecella M, and Sardina S, "Intelligent process adaptation in the SmartPM system," ACM Transactions on Intelligent Systems and Technology (TIST), vol. 8, no. 2, pp. 1–43, 2016.

[12]. Hassan S, Bahsoon R, Minku L, and Ali N, "Dynamic evaluation of microservice granularity adaptation," ACM Transactions on Autonomous and Adaptive Systems (TAAS), vol. 16, no. 2, pp. 1–35, 2022.

[13]. Xu T, Chen L, Pei Y, Zhang T, Pan M, and Furia CA, "RESTORE: Retrospective fault localization enhancing automated program repair," IEEE Transactions on Software Engineering, 2020.

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

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

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

[17]. Bartolini C, Bertolino A, Elbaum S, and Marchetti E, "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. ACM, 2009, pp. 161–170.

[18]. Sun C.-a., Wang G, Mu B, Liu H, Wang Z, and Chen TY, "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.

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

[20]. Arcuri A, “RESTful API automated test case generation with Evo-Master,” ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 28, no. 1, pp. 1–37, 2019.

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

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

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

[24]. Zou D, Liang J, Xiong Y, Ernst MD, and Zhang L, “An empirical study of fault localization families and their combinations,” IEEE Transactions on Software Engineering, 2019.

[25]. Wong WE, Gao R, Li Y, Abreu R, and Wotawa F, “A survey on software fault localization,” IEEE Transactions on Software Engineering, vol. 42, no. 8, pp. 707–740, 2016.

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

[27]. Sun C.-a., Shang Y, Zhao Y, and Chen TY, “Scenario-oriented testing for web service compositions using bpel,” in 2012 12th International Conference on Quality Software. IEEE, 2012, pp. 171–174.

[28]. Lei Y, Mao X, and Chen TY, “Backward-slice-based statistical fault localization without test oracles,” in 2013 13th International Conference on Quality Software, 2013, pp. 212–221.

[29]. Mastroeni I and Zanardini D, “Abstract program slicing: An abstract interpretation-based approach to program slicing,” ACM Transactions on Computational Logic (TOCL), vol. 18, no. 1, pp. 1–58, 2017.

[30]. ai Sun C, Zhai YM, Shang Y, and Zhang Z, “BPELDebugger: An effective BPEL-specific fault localization framework,” Information and Software Technology, vol. 55, no. 12, pp. 2140–2153, 2013.

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

[32]. Jones J, Harrold M, and Stasko J, “Visualization of test information to assist fault localization,” in Proceedings of the 24th International Conference on Software Engineering. ICSE 2002, 2002, pp. 467–477.

[33]. Li X and Orso A, “More accurate dynamic slicing for better supporting software debugging,” in 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST). IEEE, 2020, pp. 28–38.

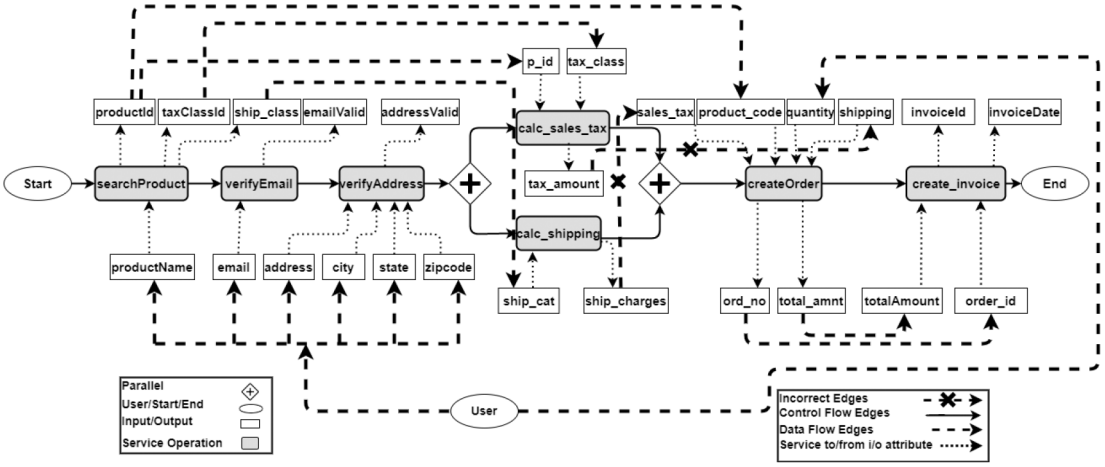


**Fig. 1.**

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

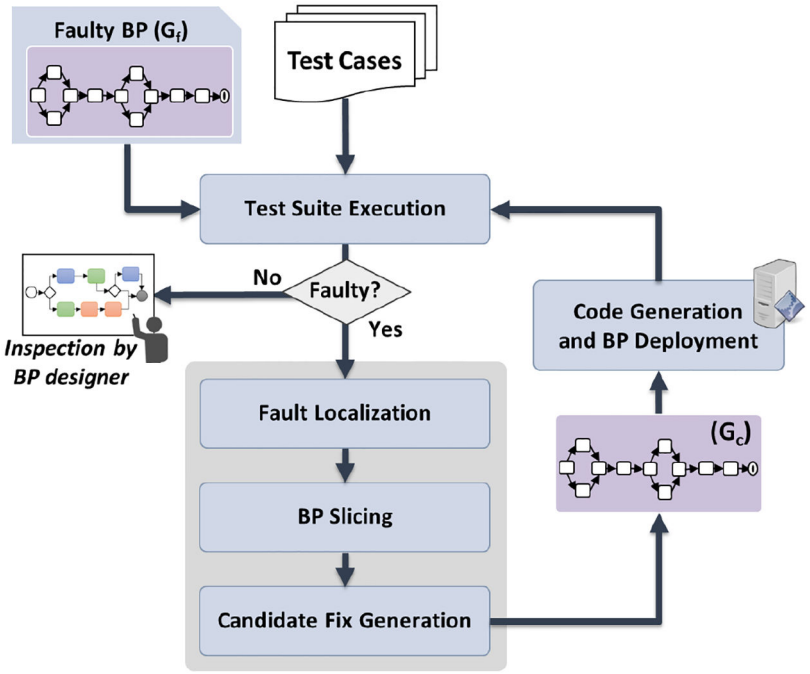


**Fig. 2.**
Efficient G&V fault resolution approach for BPs.

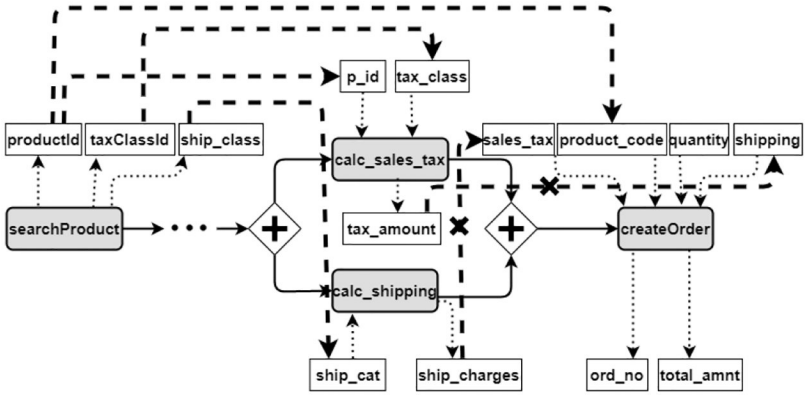


**Fig. 3.**
Slice of faulty BP $G_f$.

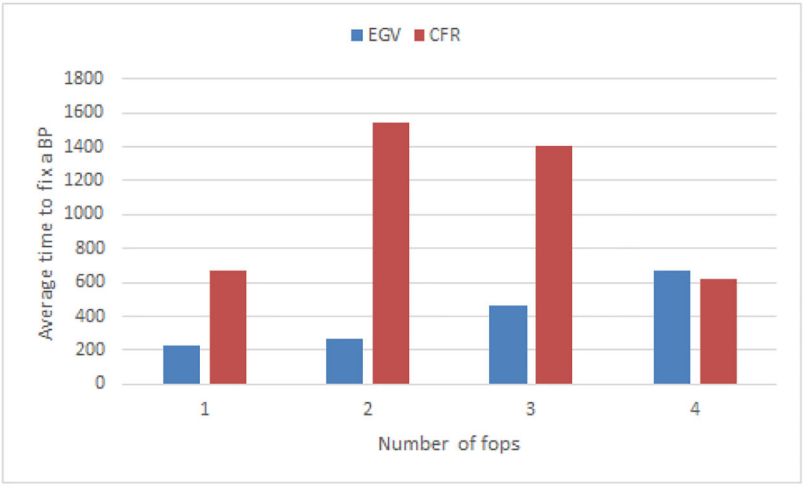


**Fig. 4.**
Execution time comparison of EGV and CFR.

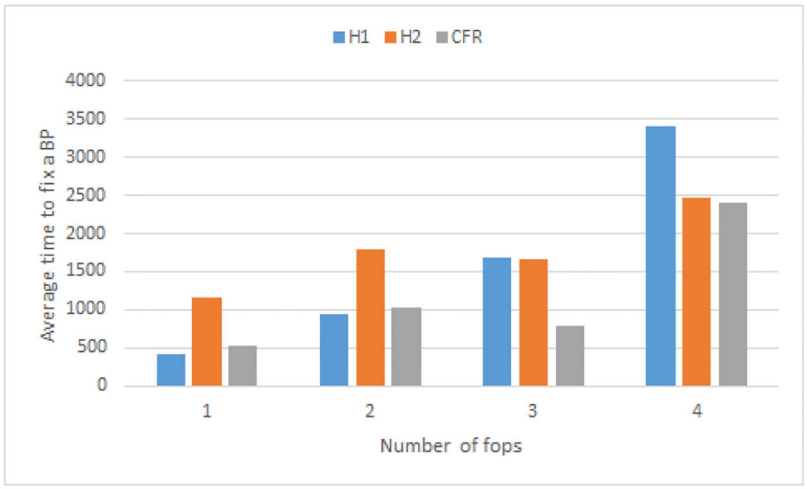


**Fig. 5.**
Execution time comparison of H1, H2, and CFR.

**TABLE I**

Fault categories, types and mutation operators specific to BPs [5].

| Fault Category | Type | Equivalent Mutation Operator | Description |
|---|---|---|---|
| Variable assignment | Replacement of Variable Identifier | ISV | Replaces variable identifiers of same type, i.e, $service_B.var1 = service_A.var1$ to $service_B.var1 = service_A.var3$ or $service_B.var1 = c$ |
| Control flow | Activity removal | AEL | Removes an activity |
| | Change of Activity Order | ASI | Changes the order of two child activities in a sequence |
| | Replacement of Sequential to Parallel loop | AFP | Replaces activities in a sequential loop with a parallel structure |
| | Replacement of Sequence to Flow Structure | ASF | Replace a *sequence* activity by a *flow* activity |
| Branching | Removal of Branch Path | AIE | Removes an *Elseif* element from an *If* structure |
| | Removal of Join Condition | AJC | Removes *joinCondition* of an activity |
| Expression | Replacement of Arithmetic operator | EAA | Replaces arithmetic operators (+, −, ×, /, *mod*) with another one |
| | Removal of Unary Operator | EEU | Removes unary operator (− or +) in an expression |
| | Replacement of Relational Operator | ERR | Replaces a relational operator (<, ≤, >, ≥, ≠, =) by another relational operator |
| | Replacement of Logical Operator | ELL | Replaces a logical operator (∧, ∨) by another logical operator |
| | Replacement of Path Operator | ECC | Replaces a path operator (/, //) by another path operator |
| | Modification of Numeric Constant | ECN | Incrementing/decrementing the value of a numeric constant by 1 or changing one digit |

*IEEE Int Conf Web Serv*. Author manuscript; available in PMC 2023 September 16.

**TABLE II**

Candidate fixes for BP graph in Fig. 1.

| 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*)$^+$ |

**TABLE III**

Accuracy of EGV, CFR, 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 |

**TABLE IV**

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 |

**TABLE V**

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 |