Research Collection School Of Computing and Information Systems

School of Computing and Information Systems

10-2019

Explaining regressions via alignment slicing and mending

Haijun WANG

Yun LIN

Zijiang YANG

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Yang LIU

See next page for additional authors

Citation

WANG, Haijun; LIN, Yun; YANG, Zijiang; SUN, Jun; LIU, Yang; DONG, Jinsong; ZHENG, Qinghua; and LIU, Ting. Explaining regressions via alignment slicing and mending. (2019). *IEEE Transactions on Software Engineering*. 14, (8), 1-17.
Available at: https://ink.library.smu.edu.sg/sis_research/4759

Author

Haijun WANG, Yun LIN, Zijiang YANG, Jun SUN, Yang LIU, Jinsong DONG, Qinghua ZHENG, and Ting LIU

Explaining Regressions via Alignment Slicing and Mending

Haijun Wang, Yun Lin*, Zijiang Yang, Jun Sun, Yang Liu, Jinsong Dong, Qinghua Zheng, Ting Liu*

Abstract—Regression faults, which make working code stop functioning, are often introduced when developers make changes to the software. Many regression fault localization techniques have been proposed. However, issues like inaccuracy and lack of explanation are still obstacles for their practical application. In this work, we propose a trace-based approach to identifying not only where the root cause of a regression bug lies, but also how the defect is propagated to its manifestation as the explanation. In our approach, we keep the trace of original correct version as reference and infer the faulty steps on the trace of regression version so that we can build a causality graph of how the defect is propagated. To this end, we overcomes two technical challenges. First, we align two traces derived from two program versions by extending state-of-the-art trace alignment technique for regression fault with novel relaxation technique. Second, we construct causality graph (i.e., explanation) by adopting a technique called *alignment slicing and mending* to isolate the failure-inducing changes and explain the failure. Our comparative experiment with the state-of-the-art techniques including dynamic slicing, delta-debugging, and symbolic execution on 24 real-world regressions shows that (1) our approach is more accurate on isolating the failure-inducing changes, (2) the generated explanation requires acceptable manual effort to inspect, and (3) our approach requires lower runtime overhead. In addition, we also conduct an applicability experiment based on Defects4J bug repository, showing the potential limitations of our trace-based approach and providing guidance for its practical use.

Index Terms—Regression bug, trace alignment, alignment slicing and mending, fault localization.

✦

1 INTRODUCTION

Regression faults are often introduced after developers make changes to the software [1], [2], [3], [4], [5], [6], [7]. As a large number of changes can happen between the original correct version and current regression version, once a regression failure is observed, it is a non-trivial task to (1) isolate the failure-inducing changes and, more importantly, (2) understand how they lead to the final observable failure, before figuring out a potential fix.

Many approaches have been proposed, mainly focusing on isolating the failure-inducing changes. The state-of-the-art techniques can be roughly divided into the following three groups.

Dynamic Slicing. A classical technique for regression fault localization is dynamic slicing [8], [9] and its variations [10], [11]. Dynamic slicing eliminates statements irrelevant to the failure based on data and control dependence, and all modified statements which the failure depends on are reported. However, data and control dependence are often abundant in a program, and consequently dynamic slicing often reports an overwhelming number of program statements.

Delta Debugging. Zeller et al. pioneered delta debugging to isolate the failure-inducing changes by repetitively reverting different subsets of changes between original correct version and current regression version [12], [13]. Their approach reports a smallest subset of changes which can be reverted to recover the original program behavior.

Intuitive as the approach is, it suffers from an inherent drawback, i.e., the number of reverted subsets is exponential

to the number of changes. Assume that there are N changes between two versions, theoretically, we need to revert 2^N subsets to isolate the failure-inducing changes. Therefore, delta debugging based techniques [12], [13], [14], [15] have to apply various heuristics to balance the trade-off between the accuracy and efficiency. We further show how such heuristics sacrifice the accuracy in Section 2.

Symbolic Analysis. The other line of works is based on symbolic analysis [16], [17], [18], [19], [20]. A representative work is AFTER [20], which works as follows. Given a test case (with an assertion), its symbolic execution is captured as a conjunctive predicate. For example, the execution of the statements {a=0; b=1; b+=a; ...; assert(b<0)} can be converted into a conjunctive predicate $P = (a == 0) \wedge (b_0 == 1) \wedge (b_1 == b_0 + a) \wedge ... \wedge (b_k < 0)$. As the execution result is incorrect, the converted conjunctive predicate must be evaluated to be FALSE. Then, it selects a smallest subset of conjuncts from P (including the assertion) whose conjunction is still FALSE. Such subset is called a minimum unsatisfiable core. Finally, the approach reports the changed statements whose execution is responsible for minimum unsatisfiable core. Novel as this line of approaches are, the runtime overhead and capability of handling complicated program expressions are usually the bottleneck for their practical use. Moreover, only analyzing the code in regression version may miss reporting the deletion changes (see details in Section 2).

On locating and explaining the root cause of a program bug, trace-alignment-based approaches have inherently advantage [21], [22], [23], [24]. Those approaches align the correct and incorrect executions. Taking the correct execution as the reference, we can check incorrect execution to narrow down the infected *steps* towards the root cause. Through the execution information, programmers can not only know

---

where the root cause lies, but also the dynamic causalities on how the faulty code propagates the defect through infected steps. Johnson et al. [23] propose trace alignment technique to align the execution traces of two runs of the same malware program (with different input), building a *casual difference graph* for analyzing its behavior. Weeratunge et al. [24] propose their trace alignment technique for different traces of two runs of the same concurrent program to identify the concurrency bug. Despite these approaches have shown good potentials, we cannot apply their approach to locating regression bug because these approaches works for aligning variational traces derived from the same program while the regression bug localization requires to align traces derived from different versions with non-negligible number of code modifications. The challenges of identifying root cause of a regression bug lies in (1) how to align the execution traces for modified parts of code in correct and regression versions, and (2) how to report unexecuted modification (e.g., code deletion) as root cause through the aligned traces.

In this work, we propose a trace-based approach called Explain Regressions via Alignment Slicing and mEnding (ERASE) to address above challenges. Our approach not only has a decent accuracy over existing regression fault localization approaches [24], [25], [26], [27] but also can present how those changes propagate the defects to the observable failure. In this work, we enhance the existing trace alignment technique by proposing a novel relaxation approach to aligning traces derived from two versions of programs with considerable number of modifications. Then, we keep the trace of the original correct version as reference and infer the faulty steps on the trace of the regression version so that we can build a causality graph of how the defect is propagated. The causality graph answers not only the why questions (e.g., why a step is executed or why the variable value is 0) but the why-not questions (e.g., why certain expected step is not executed). Note that, in contrast to existing works [23], [24], [28], our causality graph describes how the defect propagates through both the execution steps and the unexecuted modifications like deletion.

We implement a proof-of-concept tool (ERASE) for both Java and C/C++ programs. Its screenshots and demo videos are available at [29] and [30]. We evaluate the effectiveness of our approach by conducting a comparative experiment on 24 real-world C regressions. We evaluate the applicability of our approach (i.e., when our approach can and cannot work) on 298 Java regressions in the Defects4J bug repository. The results of comparative experiment show that our approach is more accurate on isolating the failure-inducing changes than the state-of-the-art techniques. The results of applicability experiment show the potential limitations of trace-based approach and provide guidance for how to apply our approach in practice. (The discussion on the limitations of dynamic approach can be checked in Section 5.2.3).

To summarize, we make the following contributions:

- Given two traces from two versions of a program, we extend existing trace alignment techniques with a novel relaxation technique to align the executions of two program versions with considerable number of modifications.

```
1:void testLang(int a[]){        1:void testLang'(int a[]){
2:  for(int i=0; i<2; i++)       2:  for(int i=0; i<2; i++)
3:    if(i%2==1)                 3:                      //c1
4:      a[i]=a[i]+1;             4:      a[i]=a[i]+1;
5:  int max=a[0];                5:  int max=a[1];       //c2
6:  if(max<a[1])                 6:  while(max<a[0])     //c3
7:    max=a[1];                  7:    {max=a[0]; break;}//c4
8:  int out=max;                 8:  int out=max;
9:  out=out-a[0];                9:  out=out-a[0];
10: assert(out==2);             10: assert(out==2);
11:}                            11:}
```

(a) Original version P (b) Regression Version P'

Fig. 1: Two versions of motivating example.

- We adopt the alignment slicing and mending technique on locating regression bugs, which derives a causality graph including both execution steps and unexecuted modifications to explain the regression failure.
- We implement ERASE for both Java and C programs, providing an interactive tool for programmers to explore why a regression failure happens in practice.
- We study the effectiveness of our approach on an experiment on 24 real-world C regressions, which demonstrates the effectiveness of ERASE over state-of-the-art approaches. In addition, our applicability experiment on 298 Defects4J bugs allows us to understand the limitation of trace-based approach and provides guidance on its practical use.

The remainder of the article is organized as follows. Section 2 gives a motivating example to illustrate the idea of ERASE. Section 3 presents the detailed algorithms. Section 4 describes our C and Java implementations of ERASE. Section 5 reports the experimental results. Section 6 reviews the related work. Section 7 concludes the paper and discusses our future work.

2 MOTIVATING EXAMPLE

Figure 1 shows a simplified regression in the Apache Lang project. For clarity, we use simplified input and code while keeping its non-triviality. In this example, the new version `testLang'` makes four changes $c1$-$c4$ to the old version `testLang`. The method takes as input an array, increases the value of its elements with odd indices, and outputs the subtraction of the maximum and first elements. Given an input a which is $[1, 2]$, `testLang` passes while `testLang'` fails.

In this example, the failure of `testLang'` is caused by mistakenly deleting the if-condition in line 3. The changes $c2$, $c3$ and $c4$ only re-implement or improve the function to compute the maximum of the input array. Unfortunately, the existing techniques (including dynamic slicing [10], delta debugging [12], [13], and symbolic analysis [20]) fail to identify the change $c1$ as the root cause. In the following, we first present how our approach works for regression showed in Fig. 1, then we explain how state-of-the-art approaches fail to locate the failure-inducing change $c1$.

2.1 Illustrating ERASE

In our approach, we use the original trace as a reference to examine how the regression trace generates the failure and propagates the defects. We first align the passing and failing
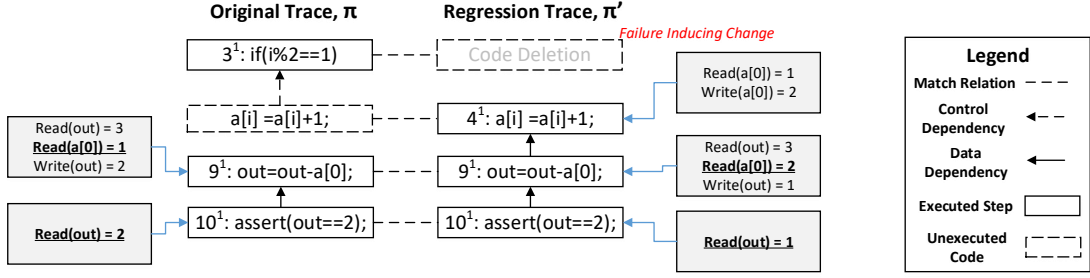
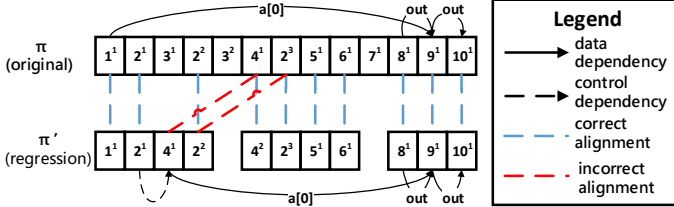Fig. 2: Causal graph generated by ERASE for the example in Fig. 1.

Fig. 3: Two traces of the example in Fig. 1.

traces. Based on the matching results, we apply *alignment slicing and mending* (see Section 3.4) to construct a causality graph to explain why and how the root cause is propagated towards the failure.

The original and regression traces for the code in Fig. 1 are shown in Fig. 3. We use L_π^k to represent a specific trace step, L represents the line number, the superscript k represents the times of L being executed, and the subscript π represents the original or regression trace (π for original one and π' for regression one). For example, 2_π^2 represents the second execution of line 2 on the original trace.

Trace Alignment. There are two challenges for aligning the original and regression traces. First, a program statement may be executed multiple times due to loops and function calls [23], it is non-trivial to match those steps executing same program statements. For example, it is a non-trivial task to automatically decide whether the step 4_π^1 matches $4_{\pi'}^1$ or $4_{\pi'}^2$ in Fig. 3. 4_π^1 and $4_{\pi'}^1$ are considered as matched if we regard the trace-matching problem as traditional sequence matching problem, as in [31], [32]. Nevertheless, such a match is inaccurate and prevents us from locating the regression bug. The inaccuracy lies in that simple sequential matching ignores trace structural semantics. Note that, $2_{\pi'}^1$ and $4_{\pi'}^1$ are in the same iteration but 2_π^1 and 4_π^1 are not, sequential matching breaks such structural semantics of execution trace. Second, we need to consider the semantic equivalence of dynamic traces with regard to source code changes. For example, given the input a=[1, 2], the trace $T_\pi = \langle 5_\pi^1, 6_\pi^1, 7_\pi^1 \rangle$ is equivalent to the trace $T_{\pi'} = \langle 5_{\pi'}^1, 6_{\pi'}^1 \rangle$ as both execution aims to compute the max value in a=[1, 2]. Note that, the challenge is that line 6–7 in original version is an if-condition while line 6–7 is a loop-condition. Their dynamic semantic is equivalent despite that static semantics is not. Classical trace alignment algorithm [23], [24] working on variational traces derived from same version of program cannot address this problem.

In this work, we develop a trace alignment algorithm designated for facilitating our alignment slicing and mending technique so that we can generate causal graph (as shown in Fig. 2) to (1) identify the failure-inducing change (see the box with red text in Fig. 2), and (2) show why and

how the root cause is propagated to the unexpected failure. In Fig. 2, solid rectangles represent execution steps, dashed rectangles represent unexecuted code, bi-directional lines represent matching relation between steps/code, solid lines represent data dependencies, and dashed lines represent control dependencies. In addition, each step is equipped with a list of its read/written variable values (see grey rectangles). We illustrate its construction as follows.

Alignment Slicing. Alignment slicing is designed for answering why questions, e.g., why the value of variable in a step in one trace is different from that in its counterpart in the other trace? It achieves this by comparing aligned trace steps in two traces so that we trace data dependence when variable value is different and control dependence when some steps cannot be aligned. In this example, our tool ERASE starts the process from the step $10_{\pi'}^1$, where the failure is observed. At this step, the value of used variable out is incorrect as its value is different from that of the matching step 10_π^1 (1 *vs.* 2). Thus, ERASE tracks data dependence backward from $10_{\pi'}^1$, through variable out to $9_{\pi'}^1$, where out is defined. At the step $9_{\pi'}^1$, there are two read variables: out and a[0]. The value of *out* is correct as its value is the same with that of the matching step 9_π^1 (3 *vs.* 3). However, the value of a[0] is incorrect as its value is different from that of the matching step 9_π^1 (2 *vs.* 1). In the same vein, ERASE follows backward data dependence from $9_{\pi'}^1$ to $4_{\pi'}^1$ through a[0]. The step $4_{\pi'}^1$ has no matching step on the original trace. In order to answer the question why $4_{\pi'}^1$ can be executed, we backward follows its control dependence to $2_{\pi'}^1$ (not shown in Fig. 2). However, the step $2_{\pi'}^1$ and its matching step 2_π^1 have the same branch evaluation. As a result, it seems to run into a dead-end, from which the slicing can no longer work.

Alignment Mending. Our approach bypasses such dead-end by asking and answering why-not questions by alignment mending. Before reaching the dead end at $2_{\pi'}^1$, ERASE asks why step $4_{\pi'}^1$ should NOT be executed (as $4_{\pi'}^1$ has no matching step on the other trace π). To answer such why-not question, ERASE switches to the other trace π to identify a step responsible for the deviated control flow. As a result, the step 3_π^1 (executing the if-statement if(i%2==1)) is *mend*ed because $4_{\pi'}^1$ can be matched (as line 4 will be executed in the first iteration in π) if the boolean expression i%2==1 is evaluated to true.

Moreover, the step 1_π^1 is identical to $1_{\pi'}^1$. Therefore, there is no more different steps to proceed, thus ERASE terminates and reports the change $c1$ as the root cause, i.e., failure inducing change.

What if trace alignment is incorrect? Then, we illustrate how incorrect trace alignment affects our root cause location

for regression bugs. Assume that the two steps 4_π^1 and $4_{\pi'}^1$ are matched (which is incorrect). When ERASE proceeds to $4_{\pi'}^1$, it asks why the value of used variable i in $4_{\pi'}^1$ is different from that in 4_π^1. As a result, ERASE will traverse through data flow and proceed to $2_{\pi'}^1$ and $1_{\pi'}^1$, and consequently misses reporting the change at line 3.

2.2 Comparing to Other Approaches

In this section, we briefly discuss how existing state-of-the-art approaches work and how they miss reporting the failure-inducing change $c1$ in Fig. 1.

Dynamic Slicing. Given a trace π and a criterion s, dynamic slicing [8] identifies the steps on π that contribute to s by data- and control-dependence. Considering the two steps 10_π^1 and $10_{\pi'}^1$ as criteria, dynamic slicing on π and π' reports the results $S_\pi=\{1^1, 6^1, 7^1, 8^1, 9^1, 10^1\}$ and $S_{\pi'}=\{1^1, 2^1, 4^1, 5^1, 8^1, 9^1, 10^1\}$. We can see that S_π reports 6 out of 13 steps and $S_{\pi'}$ reports 7 out of 11 are failure-relevant. Furthermore, the included changes $c2$, $c3$, and $c4$ in S_π and $S_{\pi'}$ are not the root causes.

Augmented Delta Debugging. Figure 4 shows the process of applying Augmented Delta Debugging (ADD) [15] to the example in Figure 1. Let $\mathcal{F}$ be the set of executed changes between two versions, i.e., $\{c1, c2, c3, c4\}$. First, $\mathcal{F}$ is partitioned into $d_1 = \{c1, c2\}$ and $d_2 = \{c3, c4\}$. Assuming that the changes in d_1 is irrelevant to the regression failure, ADD applies $\mathcal{F} \backslash d_1 = \{c3, c4\}$ to the correct version, it turns out that regression failure happens. Thus, ADD considers that the failure-inducing changes reside in $\mathcal{F} \backslash d_1 = \{c3, c4\}$. Next, ADD further partitions $\{c3, c4\}$ into $d_3 = \{c3\}$ and $d_4 = \{c4\}$. Assuming the change in d_3 is irrelevant to the regression failure, ADD applies $\mathcal{F} \backslash d_3 = \{c1, c2, c4\}$ to the correct version and the regression failure remains. Therefore, ADD reports $d_4 = \{c4\}$ responsible for the failure, which is different from the actual root cause $c1$. Note that, all the reverting process does not introduce syntactic or compilation error, i.e., no construction error [13] does not happen. Therefore, ADD cannot switch back to d_1 to locate the root cause.

ADD fails to report the root cause $c1$ for two reasons. First, ADD adopts a binary search heuristic to narrow down the failure-inducing change for efficiency, which miss a lot possible change combinations. Second, ADD assumes that the changes are independent with each other, which is not true for our example in Fig. 1.
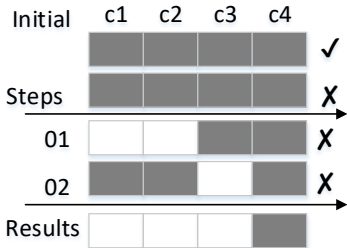

Fig. 4: Applying ADD to the example in Fig. 1.

Symbolic Approach (AFTER). AFTER [20] assumes the assertion (i.e., `assert(out==2)`, line 10 in Fig. 1) holds, and then conducts weakest precondition computation on the trace π', as shown in Table 1. Since the assertion fails, the

TABLE 1: Applying AFTER to the example in Fig. 1.

No.	Step	Weakest Precondition	Satisfiability
1	10^1	out=2	SAT
2	9^1	out-a[0]=2	SAT
3	8^1	max-a[0]=2	SAT
4	5^1	a[1]-a[0]=2	SAT
5	4^2	(a[1]+1)-a[0]=2	SAT
6	4^1	(a[1]+1)-(a[0]+1)=2	SAT
7	1^1	(2+1)-(1+1)=2	UNSAT

weakest precondition computation leads to an unsatisfiable (UNSAT) core: $(a[0] = 1) \land (a[1] = 2) \land (a[0]' = a[0] + 1) \land (a[1]' = a[1] + 1) \land (max = a[1]') \land (out_0 = max) \land (out_1 = out_0 - a[0]) \Rightarrow (out_1 \neq 2)$. Then, AFTER maps the UNSAT core to a set of program statements $\phi_0 = \{s_1, s_4, s_5, s_8, s_9, s_{10}\}$ where s_k represents the program statement at line k in Fig. 1b. Since only the change $c2$ that corresponds to statement $s_5 \in \phi_0$ is involved, AFTER reverts $c2$, re-compiles, and re-executes the program. Since the assertion still fails after $c2$ is reverted, AFTER concludes that ϕ_0 is not the root cause. Otherwise, reverting $c2$ would fix the failure. At the moment, there is no more critical predicate for ϕ_0. As a result, AFTER terminates without reporting any root cause. In this case, AFTER fails to report the root cause because the failure-inducing change is a deletion, which can never be included in the weakest precondition calculation.

Symbolic Approach (Darwin). Another classical symbolic approach is Darwin [33]. Given a test case t whose executed symbolic path condition in original program P is $f(t)$ and that in regression program P' is $f'(t)$, Darwin synthesizes a new test case t' so that $f(t') \land \neg f'(t')$. The difference between $f'(t)$ and $f'(t')$ is considered as the potential root cause. Novel as Darwin is, it fails to locate the root cause in our example.

Let $t \leftarrow$ a=[1, 2], it is possible for Darwin to synthesize $t' \leftarrow$ a=[2, 1] which meets the requirement $f(t') \land \neg f'(t')$. In our example, $f(t') = f(t) = $ "$i_0 < 2 \land i_0\%2! = 1 \land i_1 < 2 \land i_1\%2 == 1 \land a[1]_1 = a[1]_0 + 1 \land max \geq a[1]_1$". In contrast, $f'(t)$ and $f'(t')$ are different in that the condition in line 6 is `false` for t while `true` for t'. Therefore, Darwin will report $c4$ as the root cause, which is different from the actual root cause $c1$.
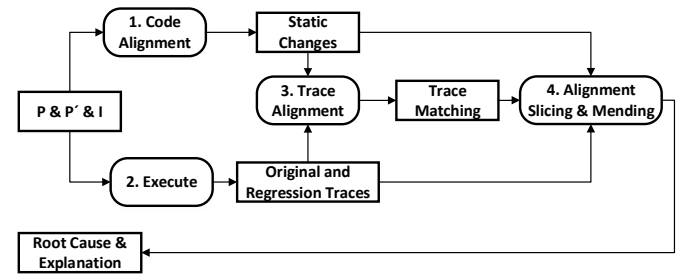
3 METHODOLOGY

Fig. 5: Overview of ERASE.

Fig. 5 illustrates the overall workflow of our approach. In Fig. 5, each rectangle represents an artifact and each rounded rectangle represents a sub-process in our approach. Our approach takes as input two versions P and P' of a program and a test case I that passes P while fails P', and

```
1:boolean equals(Object obj){      1:boolean equals(Object obj){
2:  if(obj == this)                2:  if(obj == this)
3:    return true;                 3:    return true;
4:  int s = size();                4:  if(obj == null)
5:  for(int i=0; i<s; i++)         5:    return false;
6:    if(!Util.equals(             6:  return super.equals(obj);
7:      (Shape)get(i),             7:}
8:      (Shape)that.get(i)))       8:
9:      return false;              9:
10: return true;                   10:
11:}                               11:
```

<center>(a) Original version P (b) Regression Version P'</center>

<center>Fig. 6: Regression Example in JFreechart.</center>

outputs the the failure-inducing changes and the explanation of how it propagates defects to the failure. As showed in Fig. 5, we first compute static source code changes (step 1) and execution traces of both original and regression version (step 2). Then, we match the original and regression traces with regard to the static source code changes (step 3). Along with the static changes, execution traces, and trace matching result, we apply alignment slicing and mending to isolate the failure-inducing changes as well as the explanation.

In the following, we illustrate code alignment, trace alignment, and alignment slicing and mending in details.

3.1 Code Alignment

The challenge of code alignment is to align code elements with regard to fine-grained syntactic features. Fig. 6 shows an regression example taken from JFreechart project. Text differencing tool like *diff* in Linux may report the difference that line 4–10 in original version is replaced by line 4–6 in regression version. Such a differencing report is correct but too coarse because a step executing line 4–10 in original version could be matched to any step executing line 4–6 in regression version, leading to false causalities reported by our dynamic approach. To address this issue, we refine the differencing result with our previous syntax-aware diff algorithm [32], [34]. More specifically, we transfer the code into a token sequence where each token is attached with AST type information. Thus, we can calculate the similarity score of each pair of token with regard to their code position as well as AST syntax. Hence, we use LCS-based approach to align the token sequence by computing a common subsequence with maximum token similarity. With the reference to the common subsequence, we can compute more fine grained difference. In the example in Fig. 6, the line 9 in Fig. 6a can only match line 5 in Fig. 6b as they are of return-statement AST node. Readers can go through more details in [32], [34].

Based on the differencing results, we can have a code matching function $match$, given a code element c in original (regression) version, we can have its fine grained corresponding code element $match(c)$. Note that $match(c)$ can be ϵ (i.e., empty).

3.2 Trace Alignment

Different from existing approaches [24], [25], [26], [27], we align two traces by: (1) strictly following the boundary and appearing order of loop iteration in trace and (2) identifying equivalent semantics of trace steps with regard to static code changes. For clarity, we first introduce how we align intra-method traces. We will first justify our iteration order based

```
1:void test(int x){        1:void test'(int x){
2:  for(int i=0;i<2;i++){   2:  for(int i=0;i<2;i++){
3:    if(x<=1)              3:    while(x>0 && x<3)
4:      printf("%d", x);    4:      printf("%d", x++);
5:    if(x==0)              5:
6:      x = x + 1;          6:      x = x + 1;
7:  }                       7:  }
8:}                         8:}
```

<center>(a) Original Code   (b) Regression Code</center>

<center>Fig. 7: Two versions P and P' for trace alignment.</center>
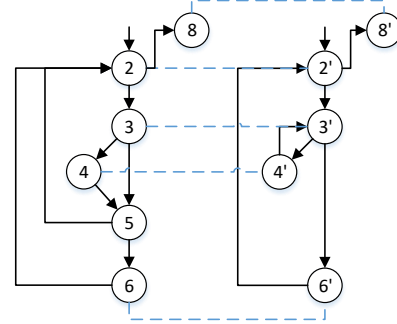

<center>Fig. 8: CFG Matching</center>

matching principle (Section 3.2.1) and how we identify equivalent trace semantics (Section 3.2.2). Then, we extend our argument to align inter-method traces (Section 3.2.3). Finally, we present the general trace alignment algorithm (Section 3.2.4).

3.2.1 Iteration Order Based Principle

Given a program P, its control flow graph (CFG) G, and an input I for P, we regard the execution trace π as a traversal on G with regard to I. Therefore, given an input, matching the traces of two versions of a program is equivalent to comparing two traversals on similar CFGs. Given an input I, two CFGs G and G', we denote the traversal of I on $G = \langle N, E \rangle$ and $G' = \langle N', E' \rangle$ as T and T'. T and T' are represented by a sequence of CFG nodes, namely, $T = \langle s_1, s_2, ..., s_m \rangle$ and $T' = \langle s'_1, s'_2, ..., s'_n \rangle$, $s_i (i = 1, ..., m)$ and $s'_j (j = 1, ..., n)$ are the nodes in N and N'. Moreover, we use T_i to denote the sequence of first i CFG nodes in T, $T[i]$ to denote the ith element (or step) in T (i starts with 1), and $T[i].node$ to denote the CFG node where the step $T[i]$ executes.

Let us take the code in Fig. 7a for example. Its CFG is showed in the left part of Fig. 8 where each CFG node is labelled corresponding to the line number in Fig. 7a. Given the input x as 0, the traversal $T = \langle 2, 3, 4, 5, 6, 2, 3, 4, 5, 2, 8 \rangle$, $T_4 = \langle 2, 3, 4, 5 \rangle$, and $T[5].node = 6$.

With static code matching technique, we can have a bilateral matching function $match$ so that $match(n) = n'$ and $match(n') = n$ where $n \in N$ and $n' \in N'$. Note that n and n' can be ϵ, which means that the static change is either a deletion (i.e., $match(n) = \epsilon$) or an addition (i.e., $match(n') = \epsilon$). Moreover, if $n \neq \epsilon$ and $n' \neq \epsilon$, $match(n) = n'$ is equivalent to $match(n') = n$. The blue dashed line in Fig. 8 shows the matching relation for CFG nodes of the code in Fig. 7.

A traversal is a dynamic walking process on a graph. Aligning two traversals T and T' is to identify when the walk of T' synchronizes with or deviates from T.
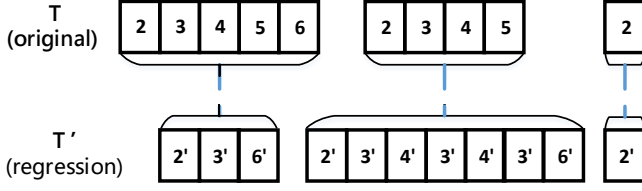
Fig. 9: Iteration Alignment Example

We can see that the complication of aligning two traversals lies in the loop in CFG as they incur repetitive CFG nodes in traversals. Assume that there is no loop in CFG, each step in a traversal walks to a unique CFG node. Thus, with regard to $match$ function, aligning two traversals is equivalent to finding the longest common subsequence of two sequences of CFG nodes.

Argument on Iteration Boundary. Let us call each time a traversal walks through a static loop is an iteration. We first argue that iterations should be matched with regard to its boundary, i.e., the steps in iteration I in T cannot be matched to steps in different iterations in T', and vice versa. Therefore, we annotate iteration boundary for each traversal so that we can regard each iteration as a separate unit as a single step. For example, given the input x=0, The traversal for code in Fig. 7a is $T = \langle [2, 3, 4, 5, 6]_1, [2, 3, 4, 5]_2, [2]_3, 8 \rangle$, and the traversal for code in Fig. 7b is $T' = \langle [2', 3', 6']_1, [2', 3', 4', 3', 4', 3', 6']_2, [2']_3, 8' \rangle$. We use brackets to represent the iterations generated by the loop in each traversal. The order of iteration is represented by the subscript.

Argument on Iteration Order. Next, given two traversals T and T', if they respectively go through the loops which are statically matched in CFGs G and G', then their derived iterations must be strictly aligned in the traversing order. For the example in Fig. 7 and the corresponding CFGs in Fig. 8. The CFGs in Fig. 8 has a matched loop with loop head of node 2. According to our argument, as the loop is matched, its kth iteration in T should only be aligned to the kth iteration in T', regardless how similar the traversing path of ith iteration in T and jth iteration in T' where $i \neq j$. Therefore, the iteration matching should be as Fig. 9. We provide a more sophisticated proof and argument in Section 8.

Therefore, we can align two traversals with the following routine. We first abstract each iteration so that each iteration is regarded as a single **iteration step** (e.g., $[2, 3, 4, 5, 6]$ in T) and the iterations derived from the same loop are assigned with its iteration order. Thus, a traversal can be converted into a sequence consists of either normal steps or iteration steps. Iteration steps can be matched only if they are derived from statically matched loops and share the same iteration order. In this regard, every step in the sequence is unique. Thus we adopt longest common subsequence for the converted traversal. If two iteration steps are matched (e.g., $[2, 3, 4, 5, 6]_1$ in T matches $[2', 3', 6']_1$ in T), we apply the same routine for their path until the alignment process ends.

3.2.2 Adapting Source Code Change

Compared to existing solutions [24], [26] designed for aligning traces derived from same program, our work is designed for aligning traces derived from two versions of a program under the same input. Thus, we need to address the challenge of *identifying the semantically equivalent parts of two executions even if their source code is different because of code change*. Semantically equivalent parts mean the control flow semantics to control the execution of the program. Note that, the tree showed in Fig. 10 is constructed from the program execution trace and the non-leaf nodes are usually branching node such as while, for, etc. Therefore, the code change applied on branching node affects our trace alignment result.

In this work, we study whether the code change of added/deleted/updated branching node makes a difference on the final execution output. If some do not, we consider the change keeps the flow semantics. With regard to the changes on control flow structure, we define four types of code changes which *may* preserve the flow semantics of dynamic execution even if the control flow structure is changed.

1. Add/Delete Selection: When the regression version adds or deletes a selection (e.g., if statement), the execution of regression version still preserve the flow semantics if the evaluation of the added or deleted condition is true. For example, Fig. 7b deletes an if statement (i.e., if(x==0)) in Fig. 7a, the trace alignment would not be influenced if the x==0 is evaluated to be true in original execution. That is, the execution of line 6 in Fig. 7a should be aligned with the execution of line 6 in Fig. 7b. Note that, existing approaches [24], [26] follows strict control flow, which can miss such alignment.

2. Replace Selection (or Loop) with Loop (or Selection): When the regression version replaces a selection with a loop, or vice versa, the execution of regression version still preserve the flow semantics between the execution of the selection and the first iteration derived from the loop. For example, Fig. 7b replaces if statement with a while statement (line 3), we should align the execution of line 4 in Fig. 7a with the execution of line 4 in Fig. 7b, even if the latter is under iteration.

3. Add/Delete Loop: Based on above two points, when the regression version adds or deletes a loop, the execution of regression version still preserve the flow semantics for the first iteration derived from the loop.

4. Replace Loop (or Selection) with Loop (or Selection): When the regression version replaces a loop (selection) with a loop (selection), the execution of regression version still preserve the flow semantics if the evaluation of the loop (selection) are the same. For example, the regression version replaces a for statement with a while statement, it does not influence the flow semantics if both boolean expression are evaluated to be true or false.

Our aforementioned iteration based approach can well address the first and fourth points. It is because that, comparing to control flow based alignment strategy [?], [26], the iteration order based alignment strategy does not strictly require the match of executed control nodes. As long as a CFG node is executed (regardless whether it is controlled by a selection), the longest subsequence is sufficient to capture the control flow difference. Nevertheless, we need to adjust our matching criteria for the second and third points.

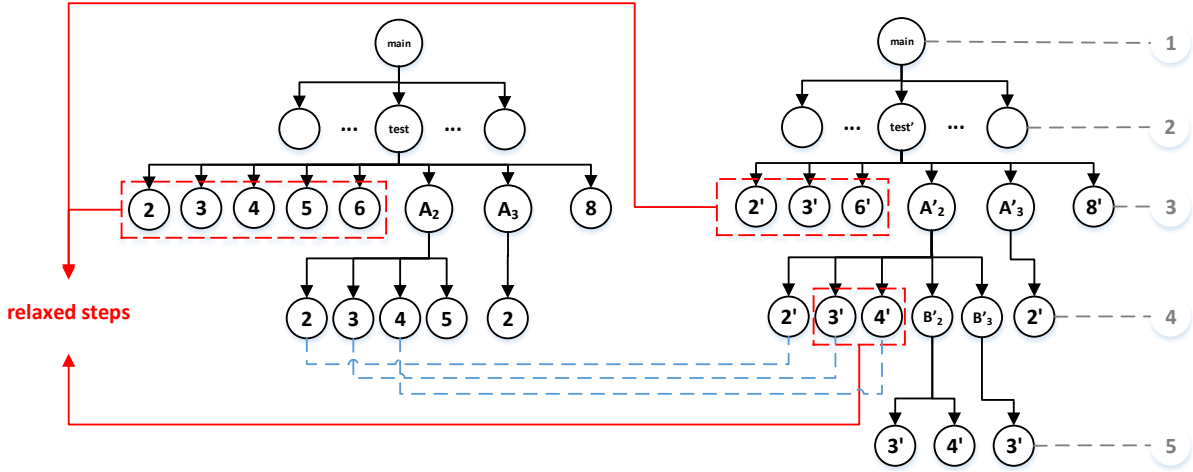**Alignment Complication Introduced by Change.** Consider the example in Fig. 7 (with input x=0) where we match

Fig. 10: Abstraction Tree

$I = [2, 3, 4, 5]_2$ in T with $I' = [2', 3', 4', 3', 4', 3', 6']_2$ in T', i.e., the second group of aligned iterations in Fig. 9. According to our argument for matching criteria (underlined text), $I[3]$ should match $I'[3]$ (i.e., 4 and 4' in CFGs) because I_2 and I'_2 are well aligned and $match(I[3].node) = I'[3].node$. However, if we abstract the traversal of two iterations, we have $I = [2, 3, 4, 5]_2$ while $I' = [2', [3', 4']_1, [3', 4']_2, [3']_3, 6']$. As an iteration step cannot match a normal step, which makes $I[3]$ and $I'[3]$ fail to match with each other. Source code modification causes such a complication as a loop structure is created by introducing an edge from node 4' to node 3' and the branch node 3 is turned into a loop head.

To this end, we relax the abstraction for first iteration to address this problem. Namely, we do not abstract the first iteration derived from any static loop. Fig. 10 shows the example of how relaxation helps match original and regression trees derived from Fig. 7 with input x=0. In Fig. 10, the relaxed steps from the first iteration are highlighted by red dashed rectangles. In the fourth layer, we have $I = [2, 3, 4, 5]_2$ (on the fourth layer) while $I' = [2', 3', 4', [3', 4']_2, [3']_3, 6']$ (on the fourth and fifth layer). Note that $I'[2]$ (i.e., 3') and $I'[3]$ (i.e., 4') are regarded as normal steps instead of being abstracted into an iteration step.

The relaxation is sound because traversing the first iteration of a loop is semantically equivalent to traversing through a true/false branch of a selection node of the same condition. Moreover, after relaxing the first iteration, the items under any non-leaf node are still unique. It is because the steps relaxed from the first iteration step must be different from either iteration step (because the step type is different) or other normal step (because otherwise other normal steps should form an iteration step). Therefore, we can still adopt longest common subsequence for the converted traversal. The blue dashed lines in Fig. 10 shows the alignment between I and I' after relaxation.

3.2.3 Extension to Function Call

Now, we discuss the case when the traversals include function call, which is another reason for repetitive CFG nodes in a traversal. Let us first call each time a traversal walks through a function as a **call**. Despite a function can be *called* many times, its calls should be aligned with the same

boundary requirement as iteration. That is, the steps in a call $C \in T$ cannot be matched to steps in different calls in T'. Therefore, we can also abstract the whole steps of a call by adding call boundary into a "call step".

3.2.4 Overall Alignment Algorithm

Based on the above arguments, we convert a traversal T into an **abstraction tree** $Tree = \langle N, E \rangle$ in which each $n \in N$ corresponds to a step $T[k] \in T$ and each $e \in E$ represents a parent-child relation, which is either of the following two cases:

- a step starting a call and all the derived steps in the call are its children.
- a step starting a *loop*, all its derived steps are its children, and each of its iteration (except the first one) are regarded as an separate step.

Fig. 10 shows an example of abstraction tree based on the code in Fig. 7b. The root of the tree is the entry method main. Each time a step calls a function or starts a loop, its derived steps are its children. There are nested loops in the example, where the relaxed steps are highlighted in dashed rectangles, and the iteration steps are represented by A_2, A_3, B_2, and B_3. A_2 and A_3 indicates the second and third iterations and B_2 and B_3 indicates the second and third nested iterations. For children of a loop step, its first iteration will not be abstracted, for example, the node 2', 3', and 4' under the invocation node of test' and the node 3' and 4' under node I_2.

Algorithm 1 shows our hierarchical trace alignment algorithm. The algorithm takes as input parent nodes of two abstraction tree r and r', the static code matching function $match$, and a matching relation set $MSet$. Before the algorithm starts, r and r' are two root steps of two trees, and $MSet$ is $\emptyset$. Our aims to obtain a $MSet \leftarrow T \times T'$.

Given two parents r and r', we first obtain their direct children in the tree, i.e., $children$ and $children'$ (line 1-2). Note that, $children$ and $children'$ are two lists with regard to execution order. Then we match these two lists with longest substring algorithm (line 3). The result is a list of pairs $pairList$. A pair $pair \in pairList$ is a pair of corresponding nodes in two trees. We use $pair.s$ to denote the child under r and $pair.s'$ to denote the child under r'. Note that, $pair.s$ and $pair.s'$ can be ϵ, i.e., some step cannot

---

Algorithm 1: TraceAlign(TreeNode r, TraceNode r', MatchFunction $match$, MatchSet $MSet$)

---

1 $children \leftarrow r.directChildren();$
2 $children' \leftarrow r'.directChildren();$
3 $pairList \leftarrow$
   $computeLongestSubsequence(children, children', match);$
4 **for** *each pair* $\in pairList$ **do**
5 $MSet \leftarrow MSet \cup pair;$
6   **if** *pair contains function call* **then**
7   TraceAlign($pair.s$, $pair.s'$, $match$);
8   **else if** *pair contains iteration* **then**
9   TraceAlign($pair.s$, $pair.s'$, $match$);
10 **return** $MSet$;

---

be matched. We first add the pair into our step matching relation set $MSet$. If a $pair$ contains either function call or iteration, we recursively call *TraceAlign()*. Thus, we can achieve $MSet$ by traversing both abstraction trees in such a top-down manner.

3.2.5 Alignment Result Taxonomy

Each element in the resulted match set (i.e., the returned output of Algorithm 1) is a pair of steps $\langle s, s' \rangle$ where s comes from the original trace and s' comes from the regression trace. Note that either s or s' can be ϵ, i.e., the step s or s' cannot be matched to any step in the other trace. In addition, we define that $s.match = s'$ if $s \neq \epsilon$ and $s'.match = s$ if $s' \neq \epsilon$. According to the results of trace alignment, for each step s in either original or regression trace, we denote the type of s as $s.type$, the matched step of s as $s.match$. We categorize a step s as follows:

- $s.type$=SRC: the source code of s is different between two versions;
- $s.type$=CTL: the step s has no matching step ($s.match = \epsilon$), i.e., it has a different control flow;
- $s.type$=DAT: the step s read variables with different values from its matching step, i.e., it has a different data flow;
- $s.type$=IDT: the step s is exactly the same with its matching step in terms of the source code, data flow, and control flow.

Note that, SRC is not exclusive from CTL and DAT.

3.3 Difference with Existing Approaches

We summarize the differences between our approach and existing approaches [24], [26] in three folds. First, we address different alignment problem from that of existing solutions [24], [26]. We align traces derived from running same input for two different versions of program and they align traces derived from running two different inputs for the same version of program. Second, existing solutions derive tree from trace via control flow point of view, in contrast, our approach derives tree from trace via iteration point of view. That is, they transfer a trace step into a non-leaf node in the tree if it is either a selection or loop, while we only transfer a trace step into a non-leaf node only if it is a loop. Iteration-based alignment has a coarser granularity, nevertheless, we show that it (1) has all the information to align the trace and (2) allows relaxation technique for better accuracy in our program settings. Third, our approach uses

```
1:void example(int x){    1:void example'(int x){
2: int a = 0;              2: int a = 0;
3: int b = 0;              3: int b = 0;
4:                         4:  if(x>0)
5: a=1;                    5:   a=1;
6: assert(a>0)            6: assert(a>0)
7:}                        7:}
```

(a) Original Code   (b) Regression Code

Fig. 11: Two versions P and P' for trace alignment.

relaxation technique to handle the effect introduced by the code changes.

3.4 Alignment Slicing and Mending

Rationale. Based on the result of trace alignment, given a step s on either trace, s can be a clue to find the root cause if $s.type \neq$IDT. Starting from a step s where the failure is manifested, the rationale of alignment slicing and mending is to keep asking *why* and *why not* questions so that we can track back from s to its root cause. In particular, alignment slicing is used for answering why questions while alignment mending is used for answering why-not questions.

Given a trace step s, if $s.type \neq$IDT, it could be caused by either (1) different data flow (i.e., $s.type$=DAT) from that in the other trace, (2) different control flow (i.e., $s.type$=CTL) from that in the other trace, and (3) code modification (i.e., $s.type$=SRC). In the following, we first discuss different data and control flow, and discuss the code modification at the end.

If $s.type$=DAT, there must be a variable var read by s and its match $s.match$ so that the value of var in s (denoted as val) is different from that in s' (denoted as val'). Therefore, we asks the following two questions:

- **Data Why Question:** Why the value of variable var is val?
- **Data Why-not Question:** Why the value of variable var is not val'?

Data Slicing and Mending. As for answering data why question, we apply data slicing on the variable var on s. More specifically, we locate the **data dominator** of s with regard to var, i.e., the latest step before s which defines var, denoted as $dd(s, var)$. The variable definition behavior of $dd(s, var)$ directly answers why variable var is of a different value from that in $s.match$. As for answering data why-not question, we first locate $s.match$ and compute $dd(s.match, var)$. As a result, we need to check three more steps, i.e., $dd(s, var)$, $s'.match$, and $dd(s.match, var)$. Algorithm 2 shows how we collect the set of steps through data slicing and mending process.

Algorithm 2: dataSlicingAndMending(Trace π, Trace π', Step s)

---

1 $varSet \leftarrow \{var | var$ reads different value in $s\};$
2 $set \leftarrow \emptyset;$
3 **for** var in $varSet$ **do**
4 $s_d \leftarrow dd(s, var);$
5 $s'_d \leftarrow dd(s.match, var);$
6 $set \leftarrow set \cup \{s_d, s'_d, s.match\}$
7 **return** set;

Fig. 11 shows an example and the execution traces T and T' for Fig. 11a and Fig. 11b is showed in Fig. 13.
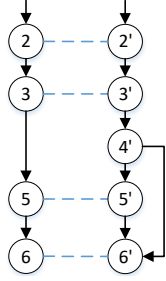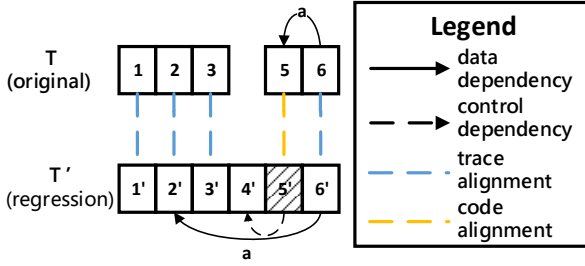
Fig. 12: CFG for Control Mending Example

Fig. 13: Example of Alignment Slicing and Mending

The change is that an if-statement is added in line 4. For formatting reason, we let the original line 4 be an empty line. Let the input be x=0, then both traces have 5 steps, i.e., $T = \langle 1, 2, 3, 5, 6 \rangle$ and $T' = \langle 1', 2', 3', 4', 6' \rangle$. For simplicity, we use line number in Fig. 11 to indicate the CFG node number, as showed in Fig. 12. In Fig. 13, we represent the data dependency with solid lines and control dependency with dashed lines. Suppose we are looking at the step $T[5]$ (i.e., $T[5].node$=6). Based on the alignment result, $T[5].match = T'[5]$. Moreover, the value of variable a in $T'[5]$ is 0 while that that in $T[5]$ is 1. Thus, $T[5].type$=DAT. In such case, we apply data slicing for $T[5]$ on variable a to reach $T[4]$, and apply data mending on $T[5]$ to reach $T'[5]$ and $T'[2]$.

Given a trace step s, if $s.type$=CTL, i.e., $s.match = \epsilon$, we asks the following two questions:
- **Control Why Question:** Why s is executed?
- **Control Why-not Question:** Why $s.match$ is not executed?

Control Slicing and Mending. As for answering control why question, we apply control slicing on s. More specifically, we locate the **control dominator** of s (denoted as $cd(s)$), i.e., the latest step before s so that its CFG node is control depended by $s.node$. The flow altered by $cd(s)$ directly answers why step s is executed. As for answering control why-not question, we conduct control mending as follows. First, we locate the source code (or CFG node) $n' = match(s.node)$. Then, we locate the latest step s' which makes n' fail to be executed. As a result, we need to check two more steps $cd(s)$ and s', as well as a static source code n'. Algorithm 3 shows how we collect the set of steps through control slicing and mending process.

Algorithm 3: controlSlicingAndMending(Trace π, Trace π', Step s)

---

1   $s_c \leftarrow cd(s)$;
2   $n' \leftarrow match(s.node)$;
3   $s'_c \leftarrow$ the controlling step in π' to make n' unexecuted;
4   **return** $\{s_c, s'_c\}$;

For example in Fig. 11 and Fig. 13, suppose we are

looking at the step $T[4]$ (as it is the answer for *why the value of variable* a *read by* $T[5]$ *is not 1?*). The control slicing will reach the step f calling the example() method. The control mending will first locate the source code line 5 at Fig. 11b. Then, we can locate $T'[4]$ as it is the latest step which makes line 5 fail to be executed.

Now, we discuss the case when $s'.type$=SRC. Given regression code can be complicated (e.g., refactoring and feature enhancement), we take a conservative strategy so that we conduct both data/control slicing and mending.

Algorithm 4: AlignSliceMend(Trace π, Trace π', Step s'_w)

---

1   $worklist \leftarrow \{s'_w\}$;
2   $reasonSet \leftarrow \{s'_w\}$;
3   **while** $worklist \neq \emptyset$ **do**
4     $s \leftarrow worklist.pop()$;
5     $set_d, set_c \leftarrow \emptyset$;
6   **if** $s.type$=SRC **then**
7       $set_d \leftarrow dataSlicingAndMending(\pi, \pi', s)$;
8       $set_c \leftarrow controlSlicingAndMending(\pi, \pi', s)$;
9   **else if** $s.type$=DAT **then**
10      $set_d \leftarrow dataSlicingAndMending(\pi, \pi', s)$;
11   **else if** $s.type$=CTL **then**
12      $set_c \leftarrow controlSlicingAndMending(\pi, \pi', s)$;
13   **for** $s \in Set_d \cup Set_c$ **do**
14   **if** $!isVisited(s)$ **then**
15       $worklist \leftarrow worklist \cup \{s\}$;
16       $reasonSet \leftarrow reasonSet \cup \{s\}$;

17 **return** $reasonSet$;

Algorithm 4 presents the pseudo-code for alignment slicing and mending. Its input are two traces π and π', and a step s'_w where the failure manifests. Its output is a step set $reasonSet$, including the steps on both traces which can serve as the explanation for s'_w. The algorithm starts with a worklist $worklist$ containing only s'_w (line 1). We pop out and check each step in $worklist$ (line 4). Each time we reach a new non-IDT step through data/control slicing and mending, we add it into $worklist$ and $reasonSet$ (line 13–16). Note that, when applying data slicing and mending (line 7 and 10) on a step s, we locate the data dominator for every read variable in s of different value with those read in $s.match$. Finally, we return $reasonSet$ which records all the checked steps once appearing in $worklist$.

Explanation Manifestation. When we have the returned $reasonSet$ from Algorithm 4, it is straightforward to convert it into a bipartite graph $G_b = \langle G_c, G_b, E \rangle$ in which $G_c = \{s_c | s_c \in reasonSet \wedge s_c \in T\}$, $G_b = \{s_b | s_b \in reasonSet \wedge s_b \in T'\}$, $E = \{(s_c, s_b) | s_c \text{ matches } s_b \wedge s_c \in G_c \wedge s_b \in G_b\}$. Moreover, for the steps in G_c or G_b, we can also manifest the control or data dependence relation among them. The generated explanation for the example in Fig. 11 is showed in Fig. 14.

With the causality graph showed in Fig. 14, programmers can (1) know where is the failure inducing change (in this case, $T'[4]$) and (2) how such change can propagate the error to the final observable failure. In this case, the user can confirm the reported failure inducing change with the following causality chain: The fact that variable a is incorrect is due to the assignment is miss-executed; the fact
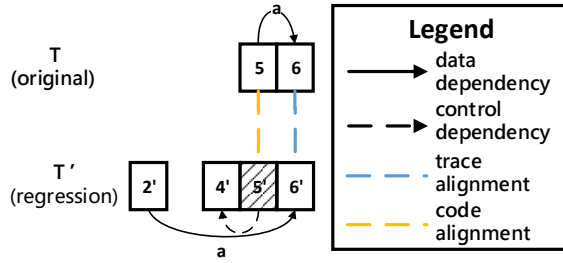
Fig. 14: Example of Explanation for Fig. 11

of miss-execution of the assignment is due to the addition of a change in line 4. The causality chain also allows an interactive tool for users to explore the causality chain, a demo video is available in [29].

4 IMPLEMENTATION

We implement our approach to support both C/C++ and Java programs. Both tools implement our approach. The C++ version is available at [30], which is based on PIN tool [35]. The Java version is available at [29], which is based on JVM instrumentation technique. Both PIN and Java instrumentation allows us to retrieve various runtime information, e.g., execution trace, read/written variable information, etc.

Noteworthy, we also support user to explore the explanation in an interactive manner. A demo video can be checked at [29]. Trace are visualized into our abstraction tree mentioned in Section 3.2. Our tool first visualize the regression and original traces in left and right views. When the programmer clicks a step on a trace, we can (1) highlight its matched step in the other trace, (2) compare its read/written variables and (3) compare the corresponding source code in the eclipse editor. Programmer can choose to apply slicing and mending for each step as well. Based on our tool, programmer can explore how the defects propagate from the root cause in a more vivid and understandable way.

5 EVALUATION

We evaluate our approach with a comparative experiment and an applicability experiment. In the comparative experiment, we apply our technique along with dynamic slicing, delta-debugging and symbolic execution technique on 24 Linux programs. The comparative experiment helps us to understand both the advantage and disadvantage of our approach over existing state-of-the-art approaches. In the applicability experiment, we apply our technique on 298 Defects4J bugs [36] where the changes between the correct and buggy version includes only bug fixing. In contrast to the comparative experiment which evaluates the (dis)advantages of our approach, we use 298 Defects4J bugs as "tests" for our approach to help us understand the scenarios where our trace-based approach can and cannot work with larger scale of bugs, providing guidance for the practical use of our approach.

5.1 Comparative Experiment

In the comparative experiment, we evaluate how accurate is ERASE to isolate the root causes of real-world regressions

comparing to state-of-the-art tools. We aim to answer the following research questions in this experiment:

- **RQ1:** How accurate is ERASE to isolate the root causes of real-world regressions?
- **RQ2:** How much effort is required to inspect the explanation of real-world regressions?
- **RQ3:** What is the runtime performance of ERASE on real-world regressions?

5.1.1 Experiment Setup

We collected 24 regressions in this experiment to compare our approach with dynamic slicing [10], augmented delta debugging (ADD) [15], and symbolic approach (AFTER) [20]. In this experiment, we implemented dynamic slicing in ERASE for comparison. Given that the tools of ADD and AFTER are not available, we reuse their experimental results (including accuracy and runtime performance) to compare with our approach. In this regard, we select the subject regressions as follows. We select all 12 regressions used in Yu et al.'s ASE'12 work (ADD) [15] and all 7 regressions used in Yi et al.'s ICSE'15 work (AFTER) [20]. Note that, Yi et al. [20] selected 7 regressions from the 12 regressions in the evaluation of ADD [15]. Thus, we have 7 regressions to compare all four approaches, and 5 (i.e., 12 - 7) regressions to compare dynamic slicing, ADD, and ERASE. Moreover, in order to generalize the results, we additionally choose 12 regressions with the criteria of large number of modifications and large program size. For large number of modifications, we select 8 regressions from CoREBench regression benchmark [37]. CoREBench regression benchmark has 4 projects, and we select 2 regressions with largest number of modifications from each project. For large program size, we select 4 regressions with average 664K lines of code (note that the largest program used in CoREBench has only 191K lines of code). Given the tools of ADD and AFTER are not available, we compare ERASE with dynamic slicing on these 12 regressions. Readers can refer to Table 2 for more details of all 24 subject regressions.

As the configuration for the subject regressions are very different and even CoREBench does not report failure inducing changes, we manually run ERASE on all 24 regressions, isolate the failure inducing changes by reading the source code and bug description, and calculate precision and recall for the reported failure inducing changes. In this experiment, the changes reported are based on source code line. Thus, we compare the reported *changed* lines covered by causal graph with the reported *changed* lines by state-of-the-art approaches. Given the set of all reported changes as A, the set of reported failure inducing changes as TP, the set of all failure inducing changes as $Root$ The precision p is calculated by $p = \frac{|TP|}{|A|}$ and the recall r is calculated by $r = \frac{|TP|}{|Root|}$.

This comparative experiment is conducted on 32-bits Ubuntu 12.0 with 3.5GHz Intel Xeon E5 CPU and 8G RAM.

5.1.2 Results

Table 3 shows the results of this comparative experiment, where each line indicates a subject regression, and their details including program name, size, the trace length of passing and failing version, number of modifications (#Ch),

TABLE 2: Subject Regressions Overview

	Name	LoC (K)	Pass	Fail	Regression Description	Report Site
benchmark in ICSE'15 (AFTER) and ASE'12 (ADD)	find-a	24	4.2.15	4.2.18	Using -L/-H produces wrong output	http://savannah.gnu.org/bugs/?12181
	find-b	40	4.3.5	4.3.6	Using -mtime produces wrong output	http://savannah.gnu.org/bugs/?20005
	find-c	40	4.3.5	4.3.6	Using -size produces error message	http://savannah.gnu.org/bugs/?30180
	make	23	3.8	3.81	Using -r produces wrong output	http://savannah.gnu.org/bugs/?20006
	bc	10	1.05a	1.06	Argument processing error	http://bugs.gentoo.org/show_bug.cgi?id=51525
	diff	20	2.8.1	2.9.2	Adds additional newline	http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=577832
	gwak	20	2.8.1	2.9.2	Adds additional newline	http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=577832
benchmark in ASE'12 (ADD)	grep	6	2.5.4	2.6	Using -include produces wrong output	http://savannah.gnu.org/bugs/?29876
	indent	15	2.2.9	2.2.10	Adds too many newlines	http://savannah.gnu.org/bugs/?27036
	tar	21	1.13.25	1.13.90	Wrong uid display	http://lists.gnu.org/archive/html/bug-tar/2004-10/msg00034.html
	ls	87	6.7	6.8	Using -x produces wrong output	http://lists.gnu.org/archive/html/bug-coreutils/2007-04/msg00000.html
	bash	20	2.8.1	2.9.2	Adds additional newline	http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=577832
CoRE-Bench (large number of modifications)	make-1	23	3.8	3.81	cannot turn off implicit rules for %.c and %.tex	http://savannah.gnu.org/bugs/?18622
	make-2	24	3.81	3.82	incorrect order only parsing in patterns	http://savannah.gnu.org/bugs/?31155
	grep-1	6	2.5.4	2.6	two options -i and -n do not work well	http://lists.gnu.org/archive/html/bug-grep/2012-08/msg00012.html
	grep-2	6	2.5.4	2.7	Core dump with pattern '(^—)*(—$)'	http://savannah.gnu.org/bugs/?33547
	find-1	40	4.3.5	4.3.6	incorrect error message on invalid argument	http://savannah.gnu.org/bugs/?28824
	find-2	49	4.3.2	4.3.5	using -mtime -2 error	http://savannah.gnu.org/bugs/?20139
	coreutils-1	91	6.7	7.3	rm -I vs. rm –interactive=once	https://debbugs.gnu.org/cgi/bugreport.cgi?bug=9308
	coreutils-2	107	7.4	7.5	tail –retry not re-attempting to open file	http://lists.gnu.org/archive/html/coreutils/2013-04/msg00003.html
benchmark (large size)	global	217	6.3.3	6.3.4	Failed to parse template	http://lists.gnu.org/archive/html/bug-global/2016-08/msg00000.html
	gettext-a	805	0.18.3	0.19.6	Mangling C escapes	http://savannah.gnu.org/bugs/?46756
	gettext-b	861	0.19.6	0.19.7	Glade file error	http://lists.gnu.org/archive/html/bug-gettext/2016-01/msg00002.html
	gettext-c	758	0.18.1	0.18.2	Behavior change	http://savannah.gnu.org/bugs/?func=detailitem&item_id=39157

number of executed modifications (#Exec Ch), number of actual failure inducing changes (#R), as well as the precision (Pre), recall (Rec), and runtime overhead (Time) for each approach. Particularly, the executed modifications refer to the covered changes by the test case executing on either the regression version or the original version of the program.

5.1.2.1　RQ1 (Accuracy): From Table 3, we can have the following conclusions. First, dynamic slicing achieves good recall (100.0% for 7 regressions, 95% for 12 regressions, and 97% for 24 regressions) with great cost of precision (6% for 7 regressions and 6% for 12 regressions, and 4% for 24 regressions). Second, ADD is outperformed by AFTER and ERASE in terms of both precision and recall. Third, AFTER is comparable to ERASE in terms of recall but reports more false positive than reported by ERASE. We qualitatively analyze the above conclusion as follows.

Dynamic Slicing's Great Sacrifice. Dynamic slicing slices the steps on the regression trace through every data and control dependency. As the data and control dependencies are usually abundant in program trace, it leads to a large number of reported steps. Therefore, it is not a surprise to see dynamic slicing achieves good recall with severe cost of precision. Nevertheless, dynamic slicing may still miss reporting failure-inducing change when some code is missed or misses execution in either regression or original version.

For example, dynamic slicing misses reporting a failure-inducing change in program *indent*, which is showed in

Listing 1: Dynamic Slicing Misses Change in *indent* program

```
1  +#if 0
2  +  ...
3  +  if(...)
4  +    prefix_blankline_requested = 0;
5  +  ...
6  +#endif
```

Listing 1. Its root cause lies in line 3 of Listing 1 where the boolean expression in if-condition should have been evaluated to `true` in the regression version. Thus, line 4 (i.e., `prefix_blankline_requested = 0`) is not executed. Note that dynamic slicing can only slice the incorrect data and control flow instead of missing flows. Hence, it cannot reach line 4 when slicing through the `prefix_blankline_requested` variable (as it is not executed). As a result, such a change is missed. In contrast, ERASE can report the change by comparing with the original trace via alignment mending technique.

ADD's Inherence Disadvantage. ADD's disadvantage in accuracy lies in that ADD regards the program logics as a black box and inducing failure-inducing change with a trial-and-error strategy through reverting changes. In contrast, AFTER and ERASE consider program semantics in terms of data and control dependencies. In the trial-and-error strategy, ADD regards a change as failure-inducing as long as its reversion can make the test case pass. In such case, ADD may induce inaccuracy. Taking *find-a* program for example. The regression in *find-a* program has two failure-inducing changes, however, ADD only report one of them. In

TABLE 3: Experiment Result

	Program	LoC (K)	Trace Length		#Ch	#Exec Ch	#R	Dynamic Slicing			ADD			AFTER			ERASE		
			Pass	Fail				Pre	Rec	Time (s)	Pre	Rec	Time (s)	Pre	Rec	Time (s)	Pre	Rec	Time (s)
All	find-a	20	2853	941	72	19	2	0.13	1.00	9.8	1.00	0.50	17	1.00	1.00	125	0.67	1.00	18.1
	find-b	40	6357	6360	244	20	1	0.09	1.00	104	0.08	1.00	61	0.20	1.00	321	1.00	1.00	110.9
	find-c	40	634	630	244	3	1	0.50	1.00	9.6	1.00	1.00	10	1.00	1.00	39	1.00	1.00	78.3
	make	23	52526	35188	1292	269	4	0.04	1.00	118.5	0.06	1.00	1833	0.67	1.00	946	0.80	1.00	76.3
	bc	10	40772	29	420	15	1	0.25	1.00	15	1.00	1.00	25	1.00	1.00	12	1.00	1.00	21.4
	diff	20	2741	2885	372	59	1	0.02	1.00	24.7	1.00	1.00	45	1.00	1.00	31	0.33	1.00	33.4
	gawk	20	8988	9771	701	2	1	0.50	1.00	128.6	1.00	1.00	73	1.00	1.00	5	1.00	1.00	46.5
	overall	24.7	16410.1	7972.0	477.9	55.3	1.6	0.06	1.00	58.6	0.12	0.91	294.9	0.65	1.00	211.3	0.73	1.00	55.0
DS vs AD vs ER	grep	6	2178	1358	596	53	3	0.13	1.00	27.7	0.14	1.00	/	/	/	/	1.00	1.00	35.6
	indent	15	10538	10723	802	47	1	0.00	0.00	58.5	0.50	1.00	/	/	/	/	1.00	1.00	25.5
	tar	21	5725	6059	619	51	1	0.04	1.00	22.1	0.00	0.00	/	/	/	/	0.33	1.00	25.3
	ls	87	1450	1401	73	13	2	0.14	1.00	13.5	0.18	1.00	/	/	/	/	1.00	1.00	134.6
	bash	20	111905	116345	1249	68	1	0.05	1.00	208.1	0.50	1.00	/	/	/	/	0.50	1.00	106.3
	overall	26.8	20555.6	15974.2	557.0	51.6	1.6	0.06	0.95	61.7	0.1	0.9	/	/	/	/	0.73	1.00	59.4
DS vs ER	make-1	23	77781	81466	1404	188	1	0.01	1.00	96.3	/	/	/	/	/	/	0.33	1.00	36.2
	make-2	23	433075	431620	2092	464	1	0.01	1.00	158.9	/	/	/	/	/	/	0.50	1.00	75
	grep-1	6	6705	5913	1428	93	1	0.03	1.00	25.6	/	/	/	/	/	/	0.50	1.00	52.4
	grep-2	6	4061	2848	1549	95	1	0.08	1.00	23.1	/	/	/	/	/	/	1.00	1.00	53.4
	find-1	40	538	544	1328	25	2	0.13	1.00	10.8	/	/	/	/	/	/	0.50	1.00	110.6
	find-2	40	7732	8384	1147	93	1	0.02	1.00	15.5	/	/	/	/	/	/	0.33	1.00	134
	coreutils-1	91	557	523	2531	38	1	0.09	1.00	9.8	/	/	/	/	/	/	1.00	1.00	83.1
	coreutils-2	107	4621	659	662	43	2	0.11	1.00	13.2	/	/	/	/	/	/	1.00	1.00	75.4
	global	217	119016	101140	24	13	1	0.14	1.00	2032.1	/	/	/	/	/	/	0.33	1.00	1877.6
	gettext-a	805	200753	214485	851	364	1	0.01	1.00	1306.2	/	/	/	/	/	/	0.25	1.00	619.4
	gettext-b	861	213847	215028	225	117	2	0.03	1.00	2837.7	/	/	/	/	/	/	0.33	1.00	758.9
	gettext-c	758	223464	223480	237	50	1	0.05	1.00	1806.4	/	/	/	/	/	/	0.33	1.00	716
	overall	137.5	64117.4	61574.2	840.1	91.8	1.5	0.04	0.97	378.2	/	/	/	/	/	/	0.56	1.00	221.0

Listing 2: False Positive of AFTER in *find-b* program

```
1  - if(get_comp_type(&s, &comp))
2  + if(get_comp_type(&timearg, &comp))
```

Listing 3: Additional Change in *diff* program

```
1  + while((char const*)p < suffix_begin)
2  - while(p< suffix_begin)
```

these two failure-inducing changes, the code of one change invokes the code of the other, and the test case fails once the latter change is triggered. Therefore, as long as ADD reverts the former change, the code of the latter change would not be triggered. As a result, ADD reports only one change. In contrast, both AFTER and ERASE can track through data flow (AFTER tracks the data flow by solving the weakest precondition and ERASE tracks the data flow by data slicing) so that both the failure-inducing changes will be tracked through the causality chain.

Comparison Between AFTER and ERASE. In Table 3, we can see that ERASE achieves the same recall with AFTER while better precision than AFTER. We investigate the reason for the ERASE's advantage over AFTER. We observe that, comparing to AFTER, ERASE can get rid of the changes introduced by refactoring or re-implementing. Note that ERASE matches the steps between original and regression traces and apply data alignment slicing only when the variable values of matched trace steps are different. As refactoring or re-implementing the code of partial function does not affect the values of their final written variables, ERASE can avoid reporting such changes. Note that, AFTER needs to transfer code into a conjunctive predicate P, as long as minimum satisfiable core of P involves refactoring or re-implementation code, it incurs false positive. Listing 2 shows a non-failure inducing change (i.e., rename) reported by AFTER as failure-inducing change in *find-b* program, which does not affect the control/data dependency.

False Positive Reported by ERASE. ERASE may include more changes than failure-inducing ones when an irrelevant change appears in the propagation of the regression bug. Listing 3 shows an example for *diff* program where the change (casting data type) appears in the cause of alignment slicing and mending. ERASE conservatively reports the change as failure-inducing as the change is control dependent by the observable faulty step. Nevertheless, with the explanation (i.e., causality graph) presented to the developers, it is convenient for them to manually inspect the false positives of ERASE.

5.1.2.2 RQ2 (Effort for Inspecting Explanation): ERASE also reports explanation in terms of causal graph. We investigate the inspecting effort by the size of generated causal graph. Table 4 shows the comparison between the reported steps in by dynamic slicing (DS) and the reported explanation by ERASE. In Table 4, we show the reported steps by dynamic slicing and ERASE on both passing and failing traces. Moreover, we also show the times of steps reported by dynamic slicing over that of ERASE (i.e., Adv (times)). We can see that dynamic slicing reported on average 7175.0 (3104.5 + 4046.5) steps for each regression, which is overwhelmingly large for developers to check. In contrast, ERASE reports a more reasonable number of explanation, i.e., on average 30.2 (13.1 + 17.1) steps).

5.1.2.3 RQ3: Performance: As showed in Table 3, in terms of the performance overhead, ERASE incurs less runtime overhead than dynamic slicing (221.0s vs 378.2s), ADD (55.0s vs 294.9s), and AFTER (55.0s vs 211.3s). In addition, we further investigate the runtime performance of ERASE, as shown in Table 5.

The time overhead of ERASE is divided into five parts: *Diff* includes the time for source code comparison and rearrangement, *Execution* presents the time for executing the program and collecting the information, *Dependence* is the time for computing dynamic control- and data-

TABLE 4: Results reported by dynamic slicing and ERASE

Regression	Trace Length		Dynamic Slicing		ERASE		Adv
	Pass	Fail	Pass	Fail	Pass	Fail	(times)
find-a	2853	941	210	252	8	10	25.7
find-b	6357	6360	635	607	17	14	40.1
find-c	634	630	204	184	3	5	48.5
make	52526	35188	4825	2796	8	36	173.2
bc	40772	29	6	7	1	3	3.3
diff	2741	2885	599	576	25	37	19.0
gawk	8988	9771	870	1303	8	3	197.5
grep	2178	1358	342	145	10	3	37.5
indent	10538	10723	516	5732	8	6	446.3
tar	5725	6059	1802	1768	9	8	210.0
ls	1450	1401	182	178	20	20	9.0
bash	111905	116345	12495	13694	15	23	689.2
make-1	77781	81466	4476	5367	22	31	185.7
make-2	433075	431620	3689	3976	18	23	187.0
grep-1	6705	5913	1035	983	13	15	72.1
grep-2	4061	2848	689	1054	9	15	72.6
find-1	538	544	135	236	8	7	24.7
find-2	7732	8384	752	775	18	15	46.3
coreutils-1	557	523	85	103	7	9	11.8
coreutils-2	4621	659	256	218	10	9	24.9
global	119016	101140	18716	18679	8	38	812.9
gettext-a	200753	214485	7291	7317	28	28	260.9
gettext-b	213847	215028	245	16727	3	19	771.5
gettext-c	223464	223480	14454	14438	38	34	401.3
average	**64117.4**	**61574.2**	**3104.5**	**4046.5**	**13.1**	**17.1**	**198.8**

TABLE 5: Runtime performance in four techniques

Regression	Diff	Execution	Dependence	Alignment	S&M	Total
find-a	8.2	8.7	0.9	0.2	0.1	18.1
find-b	84.7	15.6	6.5	3.5	0.6	110.9
find-c	66.1	8.0	1.1	3.0	0.1	78.3
make	12.2	27.4	34.3	1.0	1.4	76.3
bc	5.7	7.7	7.2	0.2	0.6	21.4
diff	8.8	14.4	9.6	0.4	0.2	33.4
gawk	15.2	11.2	14.2	4.6	1.3	46.5
grep	7.8	12.2	15.3	0.2	0.1	35.6
indent	7.9	10.7	6.0	0.6	0.3	25.5
tar	6.8	12.6	4.8	0.8	0.3	25.3
ls	117.6	8.2	4.9	3.5	0.4	134.6
bash	23.5	62.3	10.2	3.1	7.2	106.3
make-1	9.8	13.8	1.9	2.2	8.5	36.2
make-2	13.9	21.8	25.8	10.6	2.9	75.0
grep-1	16.3	8.7	18.2	6.9	2.3	52.4
grep-2	12.3	10.7	24.2	3.5	2.7	53.4
find-1	67.9	25.8	8.9	3.2	4.8	110.6
find-2	82.3	32.1	10.6	5.4	3.6	134.0
coreutils-1	36.9	15.9	25.9	1.2	3.2	83.1
coreutils-2	28.3	16.5	23.7	4.6	2.3	75.4
global	1704.8	31.1	137.0	1.1	3.6	1877.6
gettext-a	274.7	37.9	290.0	7.6	9.2	619.4
gettext-b	383.7	45.4	307.5	7.3	15.0	758.9
gettext-c	299.7	36.6	340.3	27.0	12.4	716.0
average	**137.3**	**20.6**	**55.4**	**4.2**	**3.5**	**221.0**

TABLE 6: Subject Bugs

Bugs in Repository	Project						Total
	Chart	Closure	Lang	Math	Mockito	Time	
Inspected Bugs	28	24	60	72	40	25	237
Discarded Bugs	0	109	5	34	8	2	158
Total	26	133	65	106	38	27	395

trace alignment and alignment slicing and mending takes acceptable time, ERASE can output other approaches in terms of efficiency.

5.1.3 Threats to Validity

The main threat in our study is that we can only compare ERASE with ADD and AFTER on their reported regressions as AFTER is not publicly available. In the future, we will generalize the comparison results on more regressions by implementing their approaches. Moreover, 24 real regressions can still be not generalized enough. Nevertheless, we have tried our best to mitigate this threat by sampling regressions with different criteria to make our subject regressions representative.

5.2 Applicability Experiment

We applied our approach on the bugs in Defects4J repository [36]. Up till now, Defects4J repository records 395 real-world bugs from 6 open source Java projects. For each bug in the repository, it has a buggy version and fixed version, and the changes between two versions include only bug fixing. In our experiment, we regard the fixed version as the correct version and buggy version as the regression version. Note that, in this experiment, the changes from buggy version to fixed version for each Defects4J bug are pure failure-inducing change. Nevertheless, our trace-based approach reports changes involved in alignment slicing and mending process (see Section 3.4) as failure-inducing changes. Thus, we would like to know whether those pure changes in Defects4J bugs can always be reported by ERASE. The rationale is that we regard those large number of bugs as "tests" to evaluate the limitations of our approach to understand and categorize the scenarios where ERASE can and cannot work, providing guidance for the practical use of our approach.

5.2.1 Subject Bugs

In this experiment, we use 298 out of 395 bugs in Defects4J repository, as showed in Table 6. We discard 97 bugs because either (1) the regression bug is trivial as the regression error happens at last step on buggy trace, or (2) the buggy trace or the fixed trace is over-long (i.e., over 1 million steps).

5.2.2 Result

Of the 298 Defects4J bugs, our approach localizes 265 bugs, i.e., reports the steps executing the fixed code, which occupies 88.9% of the bugs. We summarize the reasons of our approach's failure on the remaining 33 regression bugs in Table 7.

Language Specific Implementation. The major reason (27+3 bugs) lies in that some language specific feature makes data and control flow analysis fail, including the use of Java native method and runtime exception. First, the use of Java native method such as System.arrayCopy() causes some

dependencies, *Alignment* lists the time for trace alignment, and *S& M* is the time for performing alignment slicing and mending. Note that ERASE computes dynamic data- and control-dependencies and performs alignment slicing and mending based on the practical demand for them.

As showed in Table 3, compared to techniques AFTER, ADD and dynamic slicing, ERASE achieves the speedups of 4.03X, 4.69X and 1.71X, respectively. AFTER uses symbolic execution and SMT sovler and ADD needs to repetitively revert the modification and execute the program, ERASE only needs to execute the original and regression version once. Comparing to the time overhead in dynamic slicing for computing almost every dynamic data- and control-dependency, ERASE only focuses on those dependencies different in original and regression version. Given that the

TABLE 7: Reasons for Failing Localizing the Root Cause

Reason	Number
Java Native Method	27
Miss Control of Runtime Exception	3
Multi-thread	2
JVM Keyword Regulation	1

data flow missed. In addition, some runtime exception (e.g., try-catch structure) alters the control flow in an implicit way. Missing such implicit exception-caused control flow breaks the causality chain leading towards the root cause.

Multi-threaded program. Our current implementation only supports recording trace steps for single-threaded programs. We may miss some steps when some key steps happen in other threads than `main` thread. Multi-threaded programs are out of the scope of this work. In order to fully support localizing multi-thread regression bugs, we can revise the trace matching algorithm described in Section 3.2. We will address this issue as our future work.

Untraced Static Change. We also observed that some static change may not be recorded in trace but they still take effect in the execution. For example, in the 56th bug of Lang project, the root cause lies in missing using `transient` keyword. More specifically, without the `transient` modifier to tell JVM not to serialize a field, the program mistakenly serializes some fields, leading to a runtime exception. In such case, we cannot trace to such a change as JVM plays a role to deal with the effect of `transient` keyword.

5.2.3 Discussion

In this study, we can see that our approach is effective to localize the root cause if the data and control dependence is complete. Nevertheless, our approach requires recording the whole trace of a program. We discard 52 bugs because recording a long trace and building data and control dependency are memory expensive. With regard to the scalability of recording a program with long trace, we deem that a more practical way of adopting our approach is to record traces partially while incorporating the debugging process with human feedback, as proposed in [38]. More specifically, we may first ask programmers' feedback for a small range of suspicious code. Based on the range, we can record a small portion of traces and apply alignment slicing and mending on the fly to generate a partial causality graph. Programmers may further provide feedback based on generated explanation so that we can generate a new portion of traces and apply our approach again. By this means, programmers can interactively reach the root causes. We will leave the implementation of such a strategy to our future work.

5.2.4 Threats to Validity

The major threat in our feasibility experiment lies in that we discard 52 bugs whose trace length are over 1 million. Thus, the performance of our approach on the regression bugs with over 1 million steps is not clear. In the future, we can try developing a more efficient trace collection technique to generalize our finding to all the Defects4J bugs.

6 RELATED WORK

Program Slicing. A classical debugging technique is dynamic slicing [8] and its variants [2], [10], [39], [40], which works on a single trace of the program and outputs statements relevant to the slicing criterion. Thin slicing [39] includes a subset of data-dependencies, data slicing [40] includes all data-dependencies, full slicing [8] includes data- and control-dependencies, relevant slicing [2], [10], and data- and control-dependencies, also includes potential-dependencies. The main difference between ERASE and them is that ERASE alternatively and iteratively conducts slicing and mending on the passing and failing traces.

Regression Fault Localization. Comparing to traditional fault localization solution [38], [41], [42], regression fault localization techniques usually can use the original version as a reference to infer root cause. Most regression fault localization works are based on delta debugging [12], [13] and symbolic analysis [16], [17]. Delta debugging pioneers this field by isolating failure-inducing changes through reverting different subsets of changes. Misherghi et al. [43] propose hierarchical delta debugging (HDD) to improve the effect of delta debugging on program inputs with hierarchical structure. Artho et al. [44] propose iterative delta debugging (IDD) [44] to enhance the technique by leveraging the whole evolutionary history of the program. Effective as they are, these approaches may miss the failure-inducing changes due to reverting inappropriate subset of changes.

Another line of work are based on symbolic analysis which regards the failure of a program execution as a constraint solving problem. Banerjee et al. [45] propose Golden to compute weakest precondition for the regression version and report the change Yi et al. [20] propose AFTER in which they iteratively collect weakest preconditions, compute unsatisfied core, and report the related changes as failure inducing change. Qi et al. [33] propose DARWIN by generating more passing test cases based on path constraints so that they can improve the accuracy. Kim et al. [46] propose Apex which can explain the program assignment bugs by comparing the passing symbolic execution trace of a correct implementation and the failing symbolic execution trace of the buggy implementation. They show that their approach works well in small student programming assignments, while its performance on large scale program is yet investigated. Symbolic analysis based techniques are usually novel and effective in isolating the failure-inducing changes. Nevertheless, they usually suffer from scalability problem.

Trace Alignment. The other line of work is to compare the traces of the original and regression traces [23], [24], [28]. The work most similar to ERASE includes dual slicing [24], and comparative causality [14], [23], [28]. Given two schedules, one inducing the failure and the other not, the technique collects the two traces and compares them to identify the differences. The causal relation between the differences is connected by using dual slicing algorithm. Differential slicing [23] produces a causal difference graph that captures the input or environment differences that cause the target differences, as well as a sequence of differences that lead the program from the input or environment differences to the target differences. comparative causality targets for two executions with different test inputs. The major difference is that existing approach is designed for aligning traces derived from same program under execution of different inputs or schedules, while our approach works is designed

for aligning traces derived from two version of programs of the same input.

Change Impact Analysis. Change impact analysis [47], [48], [49] determines the part of program will be affected by applying certain change. Its researches mainly include call graph based analysis [47], static slicing [50], dynamic slicing and so on. However, applying all the above technique in regression fault localization leads to over-approximated impact sets. For example, dynamic slicing identifies change impact with respect to the specific program executions. In contrast, ERASE compares the original and regression traces to avoid a lot of unnecessary program impact to be inspected.

7 CONCLUSION

We presented a trace-based technique to isolate the failure-inducing changes and generate a causality graph for explaining the failure. Given two versions of a program and a test case that passes the old version while fails the new version, ERASE aligns the passing and failing traces, and conducts alignment slicing and mending to isolate the failure-inducing changes. The experiments show that ERASE is more accurate than the state-of-the-art techniques. In the future, we will extend our work as an interactive approach for more practical use.

8 APPENDIX

A traversal is a continuous dynamic walking process on a CFG. Thus, we have the following matching criteria: Two steps $T[m]$ and $T'[m']$ can be matched if and only if (1) T_{m-1} and $T'_{m'-1}$ are well aligned and (2) $match(T[m].node) = T'[m'].node$. Note that it is a recursive definition in which matches $T[m]$ and $T'[m']$ based on the alignment of T_{m-1} and $T'_{m'-1}$. In this regard, we let $T[k]$ match $T'[k']$ if there $\nexists i < k$ and $i' < k'$ such that $match(T[i].node) = T'[i'].node$. That is, $T[k]$ and $T'[k']$ are the first matched steps in each trace. Let the loop head of the statically matched loops be lh and lh, and $match(lh) = lh'$ (e.g., the node 2 and 2' in Fig. 8). Assume that (1) the first iteration I in T starts at $T[s]$ and ends at $T[e]$, (2) the first iteration I' in T' starts at $T'[s']$ and ends at $T'[e']$, and (3) T_{s-1} and $T'_{s'-1}$ are well aligned. Thus, $T[s]$ is matched to $T'[s']$, further indicating that iteration I is matched to iteration in I' (because of iteration boundary).

As we are discussing aligning steps running into repetitive CFG nodes, we can assume that the traverses T and T' walk back to the loop heads after the first iterations. Therefore, we have that $T[s].node = T[e+1].node = lh$ and $T'[s'].node = T'[e'+1].node = lh'$. Thus, $match(T[e+1].node) = match(lh) = lh' = T'[e'+1].node$. Moreover, T_e and $T_{e'}$ are expected to be well aligned because of iteration boundary. Hence, $T[e+1]$ and $T'[e'+1]$ must be matched. As they are the start of second iterations, the second iteration in T is matched to the second iteration in T'.

The same logic applies for the kth iterations until one of two loops stop deriving new iteration (i.e., the traversal walk out of the loop).

REFERENCES

[1] Fabrizio Pastore, Leonardo Mariani, Antti EJ Hyvärinen, Grigory Fedyukovich, Natasha Sharygina, Stephan Sehestedt, and Ali Muhammad. Verification-aided regression testing. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014.

[2] Hiralal Agrawal, Joseph Robert Horgan, Edward W Krauser, and Saul London. Incremental regression testing. In *ICSM*, pages 348–357. Citeseer, 1993.

[3] Yafeng Lu, Yiling Lou, Shiyang Cheng, Lingming Zhang, Dan Hao, Yangfan Zhou, and Lu Zhang. How does regression test prioritization perform in real-world software evolution? In *Proceedings of the 38th International Conference on Software Engineering*, 2016.

[4] John Backes, Suzette Person, Neha Rungta, and Oksana Tkachuk. Regression verification using impact summaries. In *Model Checking Software*, pages 99–116. Springer, 2013.

[5] Valerio Terragni, Shing-Chi Cheung, and Charles Zhang. Recontest: Effective regression testing of concurrent programs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 246–256. IEEE, 2015.

[6] Ripon K Saha, Lingming Zhang, Sarfraz Khurshid, and Dewayne E Perry. An information retrieval approach for regression test prioritization based on program changes. In *Proceedings of the 37th International Conference on Software Engineering*, pages 268–279. IEEE Press, 2015.

[7] Guowei Yang, Suzette Person, Neha Rungta, and Sarfraz Khurshid. Directed incremental symbolic execution. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(1):3, 2014.

[8] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[9] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[10] Tibor Gyimóthy, Árpád Beszédes, and Istán Forgács. An efficient relevant slicing method for debugging. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 303–321, 1999.

[11] Yi Li, Julia Rubin, and Marsha Chechik. Semantic slicing of software version histories. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering*, pages 686–696. IEEE, 2015.

[12] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28, 2002.

[13] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 253–267, 1999.

[14] William N Sumner and Xiangyu Zhang. Comparative causality: Explaining the differences between executions. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 272–281. IEEE Press, 2013.

[15] Kai Yu, Mengxiang Lin, Jin Chen, and Xiangyu Zhang. Practical isolation of failure-inducing changes for debugging regression faults. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 20–29. IEEE, 2012.

[16] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[17] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[18] Jaco Geldenhuys, Matthew B Dwyer, and Willem Visser. Probabilistic symbolic execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 166–176. ACM, 2012.

[19] Saswat Anand, Corina S Păsăreanu, and Willem Visser. Jpf–se: A symbolic execution extension to java pathfinder. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 134–138. Springer, 2007.

[20] Qiuping Yi, Zijiang Yang, Jian Liu, Chen Zhao, and Chao Wang. A synergistic analysis method for explaining failed regression tests. In *IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 257–267. IEEE, 2015.

[21] W. N. Sumner and X. Zhang. Identifying execution points for dynamic analyses. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 81–91, 2013.

[22] D. H. Ahn, B. R. de Supinski, I. Laguna, G. L. Lee, B. Liblit, B. P. Miller, and M. Schulz. Scalable temporal order analysis for large scale debugging. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, 2009.

[23] Noah M Johnson, Juan Caballero, Kevin Zhijie Chen, Stephen McCamant, Pongsin Poosankam, Daniel Reynaud, and Dawn Song. Differential slicing: Identifying causal execution differences for security applications. In *2011 IEEE Symposium on Security and Privacy (SP)*, pages 347–362. IEEE, 2011.

[24] Dasarath Weeratunge, Xiangyu Zhang, William N. Sumner, and Suresh Jagannathan. Analyzing concurrency bugs using dual slicing. In *International Symposium on Software Testing and Analysis*, pages 253–264, 2010.

[25] Xiangyu Zhang, Sriraman Tallam, Neelam Gupta, and Rajiv Gupta. Towards locating execution omission errors. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 415–424. ACM, 2007.

[26] Bin Xin, William N Sumner, and Xiangyu Zhang. Efficient program execution indexing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 238–248. ACM, 2008.

[27] Ulf Kargén and Nahid Shahmehri. Towards robust instruction-level trace alignment of binary code. In *IEEE/ACM International Conference On Automated Software Engineering*, ASE 2017, pages 342–352, 2017.

[28] Jens Meinicke, Chu-Pan Wong, Christian Kästner, and Gunter Saake. Understanding differences among executions with variational traces. *CoRR*, abs/1807.03837, 2018.

[29] Java version of erase. https://github.com/llmhyy/tregression. Accessed: 2018-07-02.

[30] C++ version of erase. https://github.com/macromachine/ERASE. Accessed: 2018-07-02.

[31] Rylan Cottrell, Joseph J. C. Chang, Robert J. Walker, and Jörg Denzinger. Determining detailed structural correspondence for generalization tasks. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 165–174, 2007.

[32] Yun Lin, Xin Peng, Zhenchang Xing, Diwen Zheng, and Wenyun Zhao. Clone-based and interactive recommendation for modifying pasted code. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 520–531, 2015.

[33] Dawei Qi, Abhik Roychoudhury, Zhenkai Liang, and Kapil Vaswani. Darwin: An approach to debugging evolving programs. *ACM Transactions on Software Engineering and Methodology*, 21(3):19, 2012.

[34] Yun Lin, Zhenchang Xing, Yinxing Xue, Yang Liu, Xin Peng, Jun Sun, and Wenyun Zhao. Detecting differences across multiple instances of code clones. In *International Conference on Software Engineering*, pages 164–174, 2014.

[35] Pin. https://software.intel.com/sites/landingpage/pintool/docs/81205/Pin/html/. Accessed: 2018-07-02.

[36] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *International Symposium on Software Testing and Analysis*, pages 437–440, 2014.

[37] Marcel Böhme and Abhik Roychoudhury. Corebench: Studying complexity of regression errors. In *Proceedings of the 23rd ACM/SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA, pages 105–115, 2014.

[38] Yun Lin, Jun Sun, Yinxing Xue, Yang Liu, and Jinsong Dong. Feedback-based debugging. In *International Conference on Software Engineering*, pages 393–403, 2017.

[39] Manu Sridharan, Stephen J Fink, and Rastislav Bodik. Thin slicing. *ACM SIGPLAN Notices*, 42(6):112–122, 2007.

[40] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Precise dynamic slicing algorithms. In *25th International Conference on Software Engineering*, pages 319–329. IEEE, 2003.

[41] Haijun Wang, Xiaofei Xie, Shang-Wei Lin, Yun Lin, Yuekang Li, Shengchao Qin, Yang Liu, and Ting Liu. Locating vulnerabilities in binaries via memory layout recovering. In *ESEC/FSE'9*, pages 718–728, 2019.

[42] Yun Lin, Jun Sun, Lyly Tran, Guangdong Bai, Haijun Wang, and Jinsong Dong. Break the dead end of dynamic slicing: Localizing data and control omission bug. In *ASE 2018*, pages 509–519, 2018.

[43] Ghassan Misherghi and Zhendong Su. Hdd: hierarchical delta debugging. In *Proceedings of the 28th international conference on Software engineering*, pages 142–151. ACM, 2006.

[44] Cyrille Artho. Iterative delta debugging. *International Journal on Software Tools for Technology Transfer*, 13(3):223–246, 2011.

[45] Ansuman Banerjee, Abhik Roychoudhury, Johannes A Harlie, and Zhenkai Liang. Golden implementation driven software debugging. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 177–186. ACM, 2010.

[46] Dohyeong Kim, Yonghwi Kwon, Peng Liu, I. Luk Kim, David Mitchel Perry, Xiangyu Zhang, and Gustavo Rodriguez-Rivera. Apex: Automatic programming assignment error explanation. In *OOPSLA 2016*, pages 311–327, 2016.

[47] Robert Arnold and Shawn Bohner. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[48] Alessandro Orso, Taweesup Apiwattanapong, James Law, Gregg Rothermel, and Mary Jean Harrold. An empirical comparison of dynamic impact analysis algorithms. In *Proceedings of the 26th International Conference on Software Engineering*, pages 491–500, Washington, DC, USA, 2004. IEEE Computer Society.

[49] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: A tool for change impact analysis of java programs. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 432–448, New York, NY, USA, 2004. ACM.

[50] Mithun Acharya and Brian Robinson. Practical change impact analysis based on static program slicing for industrial software systems. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 746–755, New York, NY, USA, 2011. ACM.

Haijun Wang received the Ph.D. degree in system engineering from School of Electronic and Information, Xi'an Jiaotong University, China. He is currently a Research Fellow at Nanyang Technological University, Singapore. His research interests include program analysis, regression testing, fault localization and software security.

Yun Lin is a Senior Research Fellow in School of Computing, National University of Singapore. He received his Ph.D. degree from Fudan University, China. His research interest is software engineering, his work includes code recommendation, software testing, and software debugging.

Zijiang Yang is an associate professor in computer science at Western Michigan University. He holds a Ph.D. from the University of Pennsylvania, an M.S. from Rice University and a B.S. from the University of Science and Technology of China. Before joining WMU he was an associate research staff member at NEC Labs America. He was also a visiting professor at the University of Michigan from 2009 to 2013. His research interests are in the area of software engineering with the primary focus on the testing, debugging and verification of software systems. He is a senior member of IEEE.

Qinghua Zheng is a professor in School of Computer Science, Xian Jiaotong University, China. He received the B.S. degree in computer software in 1990, the M.S. degree in computer organization and architecture in 1993, and the Ph.D. degree in system engineering in 1997 from Xian Jiaotong University, China. He was a post-doctoral researcher at Harvard University in 2002. His research areas include computer network security, intelligent e-learning theory and algorithm, multimedia e-learning, and trustworthy software.

Jun Sun is currently an associate professor at Singapore University of Technology and Design (SUTD). He received Bachelor and PhD degrees in computing science from National University of Singapore (NUS) in 2002 and 2006. In 2007, he received the prestigious LEE KUAN YEW postdoctoral fellowship. He has been a faculty member of SUTD since 2010. He was a visiting scholar at MIT from 2011-2012. Jun's research interests include software engineering, formal methods, program analysis and cyber-security. He is the co-founder of the PAT model checker.

Ting Liu is a professor in School of Cyber Science and Engineering, Xian Jiaotong University, China. He received his B.S. and Ph.D. degree from Xian Jiaotong University, in 2003 and 2010. He was a visiting professor at Cornell from 2016-2017. His research interests include software engineering and cyber-physical system.

Yang Liu graduated in 2005 with a Bachelor of Computing (Honours) in the National University of Singapore (NUS). In 2010, he obtained his PhD and started his post doctoral work in NUS, MIT and SUTD. In 2012 fall, he joined Nanyang Technological University (NTU) as a Nanyang Assistant Professor. He is currently an associate professor and Director of the cybersecurity lab in NTU. Dr. Liu specializes in software verification, security and software engineering. His research has bridged the gap between the theory and practical usage of formal methods and program analysis to evaluate the design and implementation of software for high assurance and security. By now, he has more than 250 publications in top tier conferences and journals. He has received a number of prestigious awards including MSRA Fellowship, TRF Fellowship, Nanyang Assistant Professor, Tan Chin Tuan Fellowship, and 8 best paper awards in top conferences like ASE, FSE and ICSE.

Jinsong Dong is a Professor in School of Computing, National University of Singapore. He received his Ph.D. degree from University of Queensland, Australia. His research interest is software engineering, program analysis, formal verification, and model checking.