Apex: Automatic Programming Assignment Error Explanation

Dohyeong Kim
Department of Computer Science,
Purdue University, USA
kim1051@purdue.edu

Yongwhi Kwon
Department of Computer Science,
Purdue University, USA
kwon58@purdue.edu

Peng Liu
Department of Computer Science,
Purdue University, USA
peng74@purdue.edu

I Luk Kim
Department of Computer Science,
Purdue University, USA
kim1634@purdue.edu

David Mitchel Perry
Department of Computer Science,
Purdue University, USA
perry74@purdue.edu

Xiangyu Zhang
Department of Computer Science,
Purdue University, USA
xyzhang@cs.purdue.edu

Gustavo Rodriguez-Rivera
Department of Computer Science, Purdue University, USA
grr@purdue.edu

Abstract

This paper presents Apex, a system that can automatically generate explanations for programming assignment bugs, regarding where the bugs are and how the root causes led to the runtime failures. It works by comparing the passing execution of a correct implementation (provided by the instructor) and the failing execution of the buggy implementation (submitted by the student). The technique overcomes a number of technical challenges caused by syntactic and semantic differences of the two implementations. It collects the symbolic traces of the executions and matches assignment statements in the two execution traces by reasoning about symbolic equivalence. It then matches predicates by aligning the control dependences of the matched assignment statements, avoiding direct matching of path conditions which are usually quite different. Our evaluation shows that Apex is every effective for 205 buggy real world student submissions of 4 programming assignments, and a set of 15 programming assignment type of buggy programs collected from `stackoverflow.com`, precisely pinpointing the root causes and capturing the causality for 94.5% of them. The evaluation on a standard benchmark set with over 700 student bugs shows similar results. A user study in the classroom shows that Apex has substantially improved student productivity.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging

Keywords Automated Feedback Generation, Computer-Aided Education

1. Introduction

According to a report in 2014 [18], computing related job opportunities are growing at two times the CS degrees granted in US. The US Bureau of Labor Statistics predicts there will be one million more jobs than students in just six years. As a result, CS enrollment surges in recent years for many institutes. With the skyrocketing enrollments, the luxury of one-to-one human attention in grading programming assignments may no longer be afforded. Automating grading is of a pressing need. In the current practice, automated programming assignments grading is mainly by running the submissions on a test suite. Failing cases are returned to the students, who may have to spend a lot of time to debug their implementation if they receive no hints about where the bug is and how to fix it. While the instructor may manually inspect the code and provide such feedback, these manual efforts can hardly scale to large classes.

In a recent notable effort [43], researchers have proposed to use program synthesis to correct buggy programming assignments. Given a correct version and a set of correction rules, the technique tries to sketch corrections to the buggy programs so that their behavior match with the correct version. Despite of the effectiveness of the technique, the demand

OOPSLA'16, November 2–4, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4444-9/16/11...$15.00
http://dx.doi.org/10.1145/2983990.2984031

of providing the correction rules adds to the burden of the instructor. Later in [26], a technique was proposed to detect the algorithm used in a functionally correct student submission and then suggest improvement accordingly. The onus is on the instructor to prepare the set of possible algorithms and the corresponding suggestions.

In this paper, we aim to develop an automatic bug explanation system for programming assignments. It takes a buggy submission from the student, a correct implementation from the instructor, and a failing test case, then produces a bug report that indicates the root cause and explains the failure causality. Since the submission and the correct implementation are developed by different programmers, they are usually quite different. Different variable names, control structures, data structures, and constant values may be used (Section 2). Note that the faulty statements are part of such differences. Recognizing them from the benign differences is highly challenging.

Debugging by comparing programs and program executions is not new. Equivalence checking [33, 38] was leveraged in [32] to derive simple and partial fixes to internal faulty state, guided by a correct execution. However, substantial structural changes between versions often make fixing internal state difficult. Weakest pre-conditions that induce behavioral differences across versions were identified and used to reason about bugs [22]. This technique relies on SMT solver and focuses on finding root cause conditions. It hardly explains the causality of failures, which is equally important. Another kind of techniques is dynamic analysis based. Comparative causality [44], dual slicing [45], and delta debugging [46] compare a passing run with a failing run and generate a causal explanation of the failure. However, they often assume the executions are from the same program to preclude syntactic differences that are difficult for dynamic analysis.

Apex is built on both symbolic and dynamic analysis, leveraging the former to handle syntactic differences and using the latter to generate high quality trace matches and causal explanations. It works by comparing the passing execution from the correct implementation and the failing execution from the buggy implementation. It collects both concrete execution traces and symbolic traces. The latter captures the symbolic expressions for the values occurring during execution. It then uses a novel iterative algorithm to compute matchings that map a statement instance to some instance(s) in the other version. The matchings are computed in a way aiming to maximize the number of equivalent symbolic expressions and respect a set of well-formedness constraints. A *comparative dependence graph* is constructed representing the dynamic dependences from both executions. It merges all the matched statement instances and their dependences to single nodes and edges respectively, and highlights the differences. A *comparative slice* is computed starting from the different outputs. A bug report is derived from the slice to capture the root cause and failure causality. Our contributions are summarized as follows.

- We formally define the problem and identify a few key constraints in constructing well-formed matchings. Specifically, we have formulated the main challenge of generating statement instance matchings as a *partial maximum satisfiability* (PMAX-SAT) problem.
- We develop an iterative algorithm that guarantees well-formedness while approximating maximality.
- We develop a prototype Apex. Our evaluation on 205 buggy real world buggy student submissions from 4 programming assignments and a set of 15 programming assignment type of programs collected from [17] shows that Apex can correctly identify the root causes and causality in 94.5% of the cases and generate very concise bug reports. The evaluation on a standard benchmark set [35] with over 700 student bugs shows similar results. A user study in the classroom shows that Apex has substantially improved student productivity.

2. Motivation

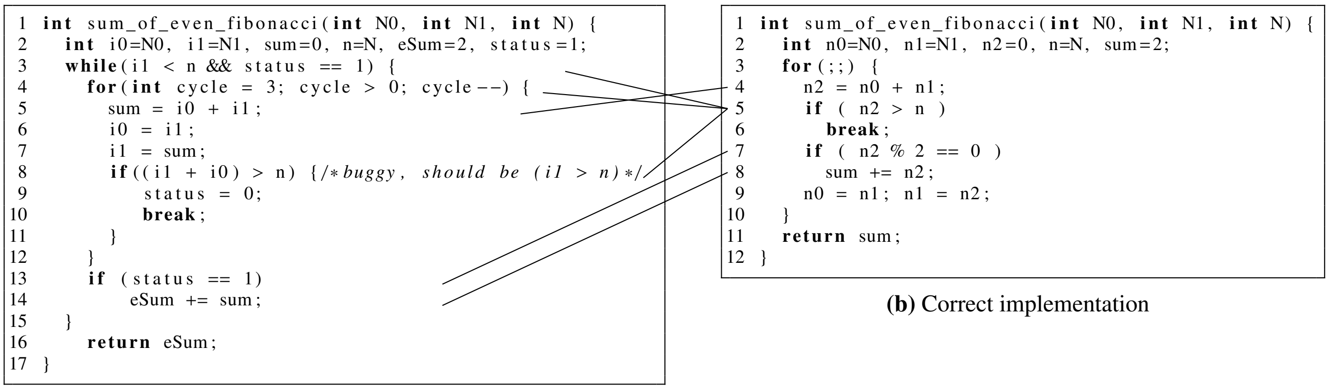
In our context, the buggy and the correct implementations are developed by different programmers. As such, they often have substantial differences representing the various ways to implement the same algorithm. We call them the *benign differences*. However, they are mixed with *buggy differences*. Our tool needs to distinguish the two. We classify popular benign differences into two categories.

Type I: Syntactic Differences. The two implementations may use different variable names and different expressions, such as `int pivot= low + (high - low) / 2` versus `int pivot= (hi - lo) / 2` . These differences may be eliminated by comparing their symbolic expressions.

Type II: Semantic Differences. (1) Different conditional statements or loop structures may be used.

Example. Consider the code snippets in Fig. 1. They are part of two programs collected from `stackoverflow.com` that compute the sum of even fibonacci numbers. The initial numbers are $N0$, $N1$, and the upper bound is N. Program (b) represents a correct implementation. The buggy version in (a) leverages that an even fibonacci number occurs after every two odd numbers. It hence uses a for loop in lines 4-12 to compute fibonacci numbers in groups of three and add the last one (the even number) to the sum. The bug is at line 8. While the predicate should test if the new fibonacci number exceeds the upper bound, the developer forgot that `i1` has been updated at line 7 and mistakenly used `i1+i0` to denote the new number. As a result, the execution terminates earlier, missing a fibonacci number in the sum. Observe that the two implementations have different control structures. □

(2) Different values may be used in achieving similar execution control. For example, in two Dijkstra implementations collected from *stackoverflow.com*, one uses a boolean array



```
1  int sum_of_even_fibonacci(int N0, int N1, int N) {
2    int i0=N0, i1=N1, sum=0, n=N, eSum=2, status=1;
3    while(i1 < n && status == 1) {
4      for(int cycle = 3; cycle > 0; cycle--) {
5        sum = i0 + i1;
6        i0 = i1;
7        i1 = sum;
8        if((i1 + i0) > n) {/*buggy, should be (i1 > n)*/
9           status = 0;
10          break;
11       }
12     }
13     if (status == 1)
14         eSum += sum;
15   }
16     return eSum;
17 }
```

(a) Buggy implementation

```
1  int sum_of_even_fibonacci(int N0, int N1, int N) {
2    int n0=N0, n1=N1, n2=0, n=N, sum=2;
3    for(;;) {
4      n2 = n0 + n1;
5      if ( n2 > n )
6        break;
7      if ( n2 % 2 == 0 )
8        sum += n2;
9      n0 = n1; n1 = n2;
10   }
11   return sum;
12 }
```

(b) Correct implementation

Figure 1: Sum of even fibonacci numbers from `stackoverflow.com` [14]. Both assume `N0`=1 and `N1`=2 so that `eSum` starts with 2.

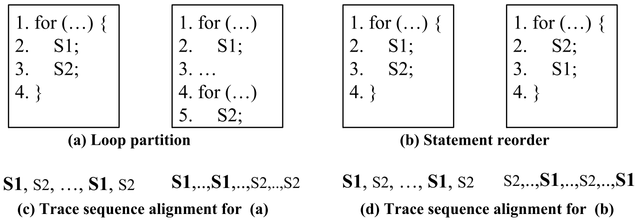


Figure 2: Program Differences Difficult for Sequence Alignment. Only the highlighted entries in (c) and (d) are matched.

`visited[i]` to denote if a node *i* has been visited whereas the other uses an integer array `perm[i]` with values `MEMBER` and `NONMEMBER` to denote the same thing.

(3) Various data structures may be used. These differences, when mixed with the differences caused by bugs, make it very challenging to meet our goal. Note that equivalence checking [33] that reasons about the symbolic equivalence of *final outputs* is less sensitive to these differences as it does not care about equivalence of internal states. However in our context, we need to align internal states to generate failure explanations.

Limitations of Sequence Alignment. A widely used approach to aligning program traces is sequence alignment [27] that identifies the longest common subsequence of two traces. It seems that we could extend the algorithm to match the sequences of symbolic expressions to identify the parts that are bug-free. However, we found that such an algorithm did not perform well in our context because the two programs are often quite different. Consider Fig. 2. In (a), the loop in the left program is partitioned to two in the right program, which are semantically equivalent to the original loop. As a result, statements `S1` and `S2` have different orders in the traces in (c). The traces cannot be fully matched by sequence alignment although they are semantically equivalent. Similarly, the statement reordering in (b) also leads to that trace entries cannot be fully matched in (d). Furthermore, the two versions may use completely different path conditions (e.g., Fig. 1). As such, sequence alignment cannot match the symbolic expressions of the predicates even though they may serve the same functionalities.

Illustrative Example. Next, we are going to use the example in Fig. 1 to illustrate the results produced by Apex.

Fig. 3 shows part of the traces for the implementations in Fig. 1. The first columns show the dynamic labels (e.g. 5_2 denotes the second instance of statement 5). The second column presents the *dynamic control dependences* (DCD). For instance, DCD(4_1)=E-3_1 means that 4_1 is dynamically control dep. on 3_1, which is further control dep. on the entry E. The third columns show the executed statements. The fourth columns present the symbolic expressions with respect to the input variables (for the assignment statements). The last columns show the values. From the symbolic traces, our tool will identify the equivalent symbolic expressions leveraging a SMT solver, as illustrated by the lines in Fig. 3. The tool then matches the DCDs of the matched symbolic expressions. Note that we cannot match DCDs by the symbolic equivalence of the predicate expressions as they are often different. Instead, we match them by well-formedness constraints (Section 3).

The lines in Fig. 1 represent the computed statement matchings. Lines 3, 4 and 8 in (a) are matched with 5 in (b), as they are the loop conditions. Line 5 in (a) is matched with line 4 in (b), denoting the computation of the new fibonacci number. Lines 13-14 in (a) are matched with lines 7-8 in (b), both updating the sum. These statement matchings can be considered as a common sub-program of the two versions. Intuitively, we reduce the problem to analyzing the two executions of the common sub-program.

From the trace matching results, a *dynamic comparative dependence graph* (DCDG) is constructed. The graph represents dynamic dependences in both executions. It merges statement instances and dependences that match. In the presence of bugs, a statement may have some of its instances

label	DCD	code	symb expr	c.value
3_1	E	i1 < n	-	True
4_1	E-3_1	cycle > 0	-	True
5_1	E-3_1-4_1	sum=i0+i1	N0+N1	3
8_1	E-3_1-4_1	(i1 + i0) > N	-	False
4_2	E-3_1-4_1-8_1	cycle > 0	-	True
5_2	E-3_1-4_1-8_1-4_2	sum=i0+i1	N1+N0+N1	5
8_2	E-3_1-4_1-8_1-4_2	(i1+i0)>N	-	False
...	...	...	...	...
5_3	E-3_1-4_1-8_1-4_2-8_2-4_3	sum=i0+i1	N1+N0+N1+ N0+N1	8
...	...	...	...	...
13_1	E-3_1	status ≡ 1	-	True
14_1	E-3_1-13_1	eSum += sum	N1+N0+N1+ N0+N1+2	10

label	DCD	code	symb expr	c.value
4_1	E	n2= n0 + n1	N0+N1	3
5_1	E	n2 > n	-	False
7_1	E-5_1	n2%2==0	-	False
4_2	E-5_1	n2=n0+n1	N1+N0+N1	5
5_2	E-5_1	n2>n	-	False
7_2	E-5_1-5_2	n2%2==0	-	False
4_3	E-5_1-5_2	n2=n0+n1	N1+N0+N1+N0+N1	8
5_3	E-5_1-5_2	n2>n	-	False
...	...	...	...	
7_3	E-5_1-5_2-5_3	n2%2==0	-	True
8_1	E-5_1-5_2-7_3	sum+=n2	3×N1+2×N0+2	10

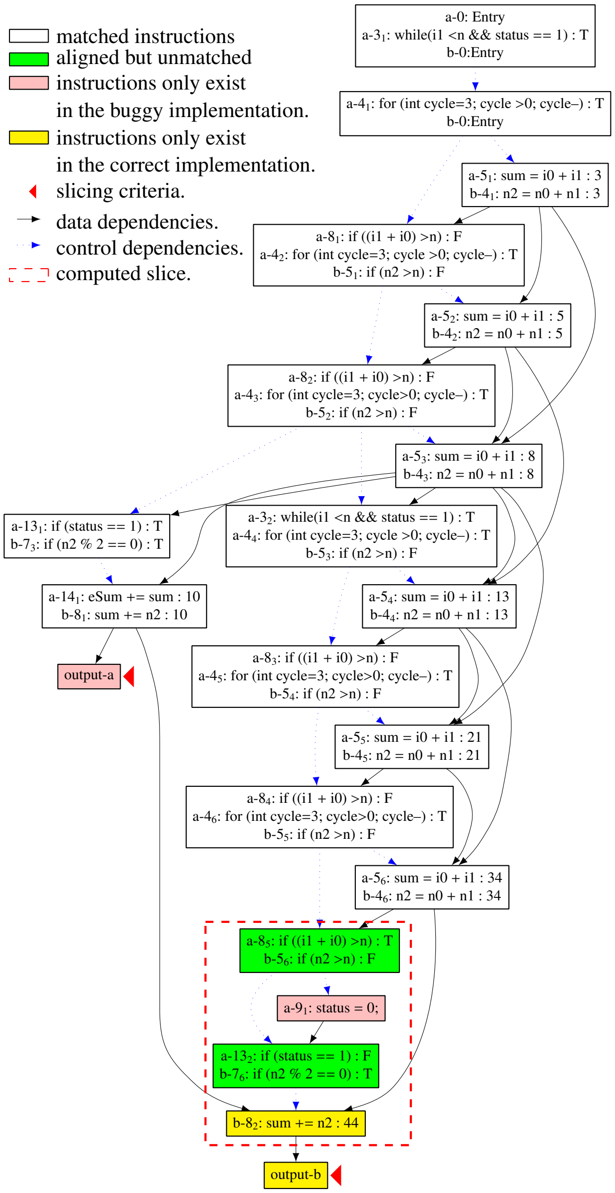
Figure 3: Part of the symbolic and concrete traces for Fig. 1 where N0=1, N1=2, N=32. The copy statements are precluded.

matched but not the others. These instances that are supposed to match but do not are called the *aligned but unmatched* instances. They are usually bug related. For example in Fig. 1, line 8 in (a) has all its instances matched with line 5 in (b) except the last one, which took the wrong branch outcome due to the bug. The last one is hence an aligned but unmatched instance. We also merge such instances in the graph but highlight their different values. Statement instances that are neither matched nor aligned are represented separately. Note that in the paper, words "*align*" and "*match*" have different meanings.

Fig. 4 presents the DCDG for the executions in Fig. 3. Plain nodes represent matched statement instances. Green nodes are instances that are aligned but unmatched. Each plain/green node contains instances from both runs. Red/yellow nodes are those only present in the buggy/correct run, each containing only one instance. Label "a-5_1" means the first instance of line 5 in version (a). The concrete values are also presented in the right side of the nodes. Observe that the computations of the fibonacci numbers 3, 5, 8, 13, 21, and 34 are matched (i.e. a-5_1 vs. b-4_1,..., a-5_6 vs. b-4_6). The corresponding loop conditions and the first updates of the sum (i.e. a-14_1 vs. b-8_1) are also matched.

The loop conditions a-8_5:`if (i1+i0>n)` and b-5_6:`if (n2>n)` are aligned but not matched (i.e. the first green node). Hence the buggy execution exits the loop whereas the correct execution continues. Consequently, the conditions guarding the updates of the sum are also aligned but not matched (i.e. the second green node). As such, the sum was updated in the passing run but not in the failing run.

A *comparative slice* is computed starting from the two different outputs, denoting the causal explanation of the differences. A bug report is generated from the slice, explaining (1) what the buggy version has done wrong and (2) what is the correct thing to do. Since part (2) is usually derived from the correct version invisible to the student, our tool translates it using the variable names in the buggy version. In Fig. 4, the two output nodes with triangles are the slicing criterion. The dotted box represents the slice. The root of the slice is exactly the buggy statement and its alignment (a-8_5 vs. b-5_6),



matched instructions
aligned but unmatched
instructions only exist in the buggy implementation.
instructions only exist in the correct implementation.
slicing criteria.
data dependencies.
control dependencies.
computed slice.
a-0: Entry
a-3_1: while(i1 <n && status == 1) : T
b-0:Entry
a-4_1: for (int cycle=3; cycle >0; cycle–) : T
b-0:Entry
a-5_1: sum = i0 + i1 : 3
b-4_1: n2 = n0 + n1 : 3
a-8_1: if ((i1 + i0) >n) : F
a-4_2: for (int cycle=3; cycle >0; cycle–) : T
b-5_1: if (n2 >n) : F
a-5_2: sum = i0 + i1 : 5
b-4_2: n2 = n0 + n1 : 5
a-8_2: if ((i1 + i0) >n) : F
a-4_3: for (int cycle=3; cycle>0; cycle–) : T
b-5_2: if (n2 >n) : F
a-5_3: sum = i0 + i1 : 8
b-4_3: n2 = n0 + n1 : 8
a-13_1: if (status == 1) : T
b-7_3: if (n2 % 2 == 0) : T
a-3_2: while(i1 <n && status == 1) : T
a-4_4: for (int cycle=3; cycle >0; cycle–) : T
b-5_3: if (n2 >n) : F
a-14_1: eSum += sum : 10
b-8_1: sum += n2 : 10
a-5_4: sum = i0 + i1 : 13
b-4_4: n2 = n0 + n1 : 13
output-a
a-8_3: if ((i1 + i0) >n) : F
a-4_5: for (int cycle=3; cycle>0; cycle–) : T
b-5_4: if (n2 >n) : F
a-5_5: sum = i0 + i1 : 21
b-4_5: n2 = n0 + n1 : 21
a-8_4: if ((i1 + i0) >n) : F
a-4_6: for (int cycle=3; cycle>0; cycle–) : T
b-5_5: if (n2 >n) : F
a-5_6: sum = i0 + i1 : 34
b-4_6: n2 = n0 + n1 : 34
a-8_5: if ((i1 + i0) >n) : T
b-5_6: if (n2 >n) : F
a-9_1: status = 0;
a-13_2: if (status == 1) : F
b-7_6: if (n2 % 2 == 0) : T
b-8_2: sum += n2 : 44
output-b

Figure 4: DCDG for the Example in Fig. 1.

DEFINITIONS:

Label ℓ, t *LabelInst* ℓ_i/t_j : the i/j-th instance of label ℓ/t

$ispred(\ell_i)$: if ℓ_i is a predicate instance

$sym\_expr(\ell_i)$: symbolic expression of ℓ_i

$DCD(\ell_i)$: predicate instances that ℓ_i directly/transitively control dep. on

$\ell_i \rightsquigarrow \ell'_k$: ℓ'_k is directly/transitively dependent on ℓ_i

$\ell_i \leftrightarrow t_j$: the statement instance at ℓ_i matches with that at t_j

WELL-FORMEDNESS CONSTRAINTS:

[WF-SYM]

$$\ell_i \leftrightarrow t_j \implies (\neg ispred(\ell_i) \implies sym\_expr(\ell_i) \equiv sym\_expr(t_j))$$

[WF-CD]

$$\begin{aligned}\ell_i \leftrightarrow t_j \implies & (\forall \ell'_k \in DCD(\ell_i),\ \exists t'_l \in DCD(t_j)\ \ \ell'_k \leftrightarrow t'_l) \wedge \\ & (\forall t'_l \in DCD(t_j),\ \exists \ell'_k \in DCD(\ell_i)\ \ \ell'_k \leftrightarrow t'_l)\end{aligned}$$

[WF-X]

$$\begin{aligned}\ell_i \leftrightarrow t_j \implies & \neg\exists \ell'_k, t'_l,\ \ell'_k \leftrightarrow t'_l \wedge ((\ell'_k \rightsquigarrow \ell_i \wedge t_j \rightsquigarrow t'_l) \vee \\ & (\ell_i \rightsquigarrow \ell'_k \wedge t'_l \rightsquigarrow t_j))\end{aligned}$$

Figure 5: Definitions and Constraints for Instance Matching.

which have different branch outcomes. The interpretation of the slice, in the language of the buggy version, is that "*Statement 8* `if(i1+i0>n)` *should have taken the false branch. As a result, statement 9* `status=0` *should not have been executed. Consequently, statement 13* `if(status==1)` *should have taken the true branch,* `eSum+=sum` *should have been executed, and eventually* `eSum` *should have been 44 instead of 10*". It precisely catches the root cause and failure causality, and provides strong hints about the fix. Note that the yellow node b-8_2:`sum+=n2` is translated to `eSum+=sum` in the buggy version.

3. Problem Formalization

The key challenge is to generate statement instance matchings. We use labels ℓ and t to denote statements in the two respective implementations. Due to implementation differences, an instance in one execution can match with multiple instances in the other execution.

Intuitively, if two assignment instances match, their symbolic expressions should be equivalent. Furthermore, their control dependences need to match. It does not mean the corresponding comparison expressions (at the control dependence predicates) need to be equivalent. In fact they are often different. Hence, we ignore the expressions in predicates, treating them as place holders. We match the label instances of these predicates based on the matchings of the assignments control dependent on the predicates and a set of *well-formedness* constraints defined in Fig. 5. Rule [WF-SYM] denotes that if the matching is for assignments, the symbolic expressions must be equivalent. Rule [WF-CD] means that if two instances ℓ_i and t_j match, a (transitive) dynamic control dependence of ℓ_i/t_j must match with a (transitive) control dependence of t_j/ℓ_i. Rule [WF-X] indicates that if ℓ_i and t_j match, there must not be another match ℓ'_k and t'_l such that ℓ_i is dependent on ℓ'_k and t'_l is dependent on t_j, or vice versa. Otherwise, a cycle of dependence is formed, which is impossible in program semantics. Note that matching of transitive data dependences is already implicitly enforced by the symbolic equivalence in [WF-SYM], because two symbolic expressions are equivalent means that their computations (i.e., data slices) are equivalent.

Example. Fig. 6 shows part of the executions from Fig. 3 in their dependence graph view. The lines across executions denote matches. Figure (a) shows matchings satisfying the well-formedness constraints. To satisfy a-5_1 $\leftrightarrow$ b-4_1, their dynamic control dependences need to match (Rule [WF-CD]). The only legitimate matchings are $ENTRY \leftrightarrow ENTRY$, a-$3_1$ $\leftrightarrow$ $ENTRY$ and a-4_1 $\leftrightarrow$ $ENTRY$. To satisfy the second assignment matching a-5_2 $\leftrightarrow$ b-4_2, the DCDs of a-5_2, including $ENTRY$, a-3_1, a-4_1, a-8_1 and a-4_2, should match with those of b-4_2, including $ENTRY$ and b-5_1.

Figures (b) and (c) show two options. In (b), $I_1 \leftrightarrow E$ (i.e. a-8_1 $\leftrightarrow$ $ENTRY$). However, this matching and the assignment matching $+_1 \leftrightarrow +_1$ together violate Rule [WF-X], because of $+_1 \rightsquigarrow I_1$ on the left (line 8 depends on line 7 and then line 5 according to Fig. 1(a)) and $ENTRY \rightsquigarrow +_1$ on the right. Since the match edges are bi-directional, the four edges in the shaded region form a cycle. Similarly, (c) shows another ill-formed matching in which $F_1 \leftrightarrow I_1$ induces a cycle. The only legitimate matching is the one shown in figure (a), in which a-8_1 $\leftrightarrow$ b-5_1 and a-4_2 $\leftrightarrow$ b-5_1. □

Since one execution is buggy, total matching is impossible. Our goal is hence to maximize the number of matches. We reduce the problem to a *partial maximum satisfiability* (PMAX-SAT) problem. Given an UNSAT conjunction of clauses, the *maximum satisfiability* (MAX-SAT) problem aims to generate assignments to variables that maximizes the number of clauses that are satisfied. PMAX-SAT is an extension of MAX-SAT, which aims to ensure the satisfiability of a subset of clauses while maximizing the satisfiability of the remaining clauses. In our context, we want to maximize the number of assignment matchings while assuring the well-formedness constraints are satisfied. *We consider assignments more essential than predicates because the symbolic expressions of predicates are often quite different across programs even when they serve the same purpose.* Our problem statement is hence formulated as follows.

DEFINITION 1. *Given two executions, let $\ell_i \leftrightarrow t_j$ be a boolean function for each pair of assignment statement instances denoted by ℓ_i and t_j, with $\ell_i \leftrightarrow t_j = 1$ meaning they match.*

$$F = \boxed{\bigwedge_{\forall \neg ispred(\ell_i),\ \neg ispred(t_j)} \ell_i \leftrightarrow t_j}^{(1)} \wedge \boxed{C_{WF\text{-}SYM} \wedge C_{WF\text{-}CD} \wedge C_{WF\text{-}X}}^{(2)}$$

, with $C_{WF\text{-}SYM}$, $C_{WF\text{-}CD}$ and $C_{WF\text{-}X}$ the instantiations of well-formedness constraints using the relevant label instances. Our goal is to solve F while ensuring part (2) must be satisfied and maximizing the satisfiability of part (1).

PMAX-SAT is NP-hard. The formula has quantifiers and is cubic to the execution length. Solving it is prohibitively expensive.

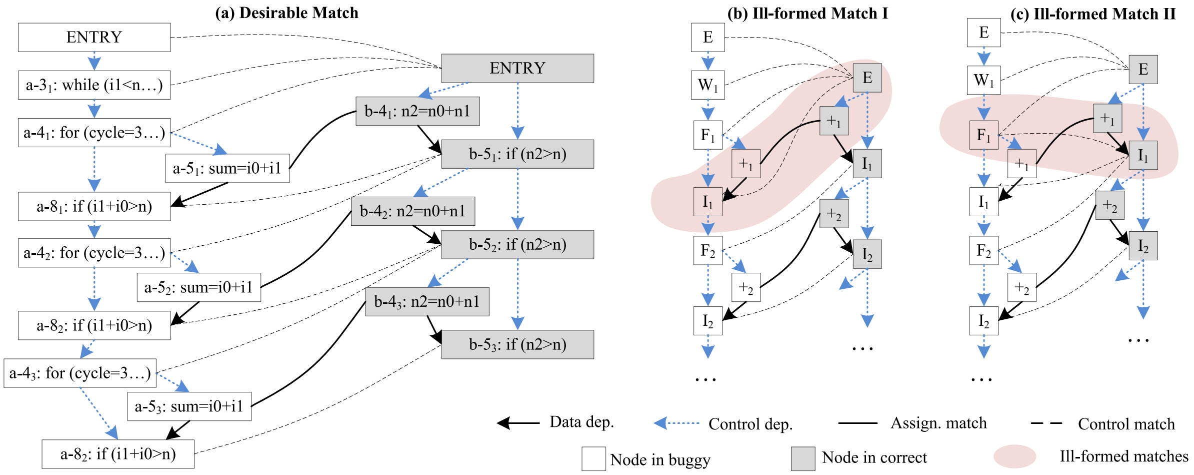


Figure 6: Instance Matching for the Example in Fig. 1. The nodes in grey are from the correct implementation. In (b) and (c), node ‘W$_1$’ stands for the first instance of the while loop in (a). Similarly, ‘F’, ‘I’, ‘+’ nodes in (b) an (c) stand for the for loop, if conditions, and the addition operations.

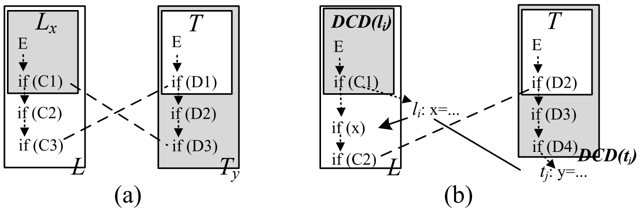


Figure 7: Cycles in Matchings. Boxes Denote Control Deps.

4. Design

The design of Apex consists of three phases and features an approximate solution to the statement instance matching (PMAX-SAT) problem.

In phase (1), an iterative matching algorithm is applied. In each iteration, sequence alignment is used to match the symbolic expression traces. Apex then matches the dynamic control dependences of the matched expressions and checks well-formedness. In the following iterations, Apex repeats the same procedure to match the residues, until no more matches can be identified. This is to handle statement reordering as exemplified in Fig. 2. In particular, in the first round, it matches the S1 sequences. Then in the second round, it matches the S2 sequences.

In phase (2), the (bug related) residues are further aligned (not matched) based solely on control structure, without requiring the symbolic expressions to be equivalent. Particularly, Apex summarizes all the matches identified in the previous phase to generate a matching at the statement level (not the instance level). Intuitively, this statement level matching identifies the common sub-program of the two versions. We then leverage the statement mapping to identify the entries that are supposed to match but their symbolic expressions are different. These entries are likely bug related. Note aligning these entries allows us to not only identify buggy behavior, but also suggest the corresponding correct behavior.

In phase (3), a dynamic comparative dependence graph is constructed and the comparative slice is computed to generate the bug report.

4.1 Phase (1): Iterative Instance Matching

The matching procedure in this phase is iterative. In each round, we first extend sequence alignment to match the symbolic expression sequences. Two expressions can be matched if they are equivalent. Apex then traverses the matched expression pairs in the generated common sub-sequence to match their dynamic control dependences (DCDs) by the well-formedness constraints (not by the symbolic equivalence of predicates). Due to Type II differences (e.g. control structure differences), it is often difficult to match a predicate uniquely to another predicate in the other version. We hence construct matchings between predicate sequences. Such matchings may be coarse-grained at the beginning (e.g. $E\text{-}3_1\text{-}4_1 \leftrightarrow E\text{-}5_1$). They are gradually refined (e.g. the previous matching becomes $E\text{-}3_1 \leftrightarrow E$ and $4_1 \leftrightarrow 5_1$).

Well-formed Matching of Control Dependences. Next we focus on explaining how Apex matches the DCDs and checks well-formedness. The algorithm traverses the longest common sub-sequence $\mathbb{C}$ produced by sequence alignment, trying to match the DCDs of each expression pair.

The traversal procedure is described by the two term rewriting rules on the bottom of Fig. 8. The symbols and functions used in the rules are defined on the top. The configuration of evaluation consists of the common sub-sequence $\mathbb{C}$ containing a sequence of label instance pairs, the DCD mappings $\mathbb{I}$ used to facilitate well-formedness checks, and the instance mappings $\mathbb{V}$.

$LabelInstSeq\ L,T := \overline{\ell_i}|\overline{t_j}$ $\quad InstMap := \mathcal{P}(LabelInstSeq \times LabelInstSeq)$ $\quad SeqAlignment\ \mathbb{C} := \overline{\langle \ell_i, t_j \rangle}$

$InstMap\ \mathbb{I}$: mappings between dynamic control dep. $\quad InstMap\ \mathbb{V}$:the resulting instance matchings $\quad LabelInstSeq\ DCD(\ell_i)$: dynamic control dep. of ℓ_i

$\textsc{wellformed}(L, T, \mathbb{I})$ determines if $L \leftrightarrow T$ is a well-formed matching.

$$\textsc{wellformed}(L,T,\mathbb{I}) = \begin{cases} false & \exists L_x, T_y,\ L_x \leftrightarrow T_y \in \mathbb{I} \wedge ((L_x \subset L \wedge T \subset T_y)\ \vee\ (L \subset L_x \wedge T_y \subset T)) \\ false & \exists \ell_i, t_j \neg ispred(\ell_i) \wedge \neg ispred(t_j) \wedge \ell_i \leftrightarrow t_j \in \mathbb{V} \wedge DCD(\ell_i) \leftrightarrow DCD(t_j) \in \mathbb{I} \wedge \\ & \quad DCD(\ell_i) \subset L \wedge T \subset DCD(t_j) \wedge \ell_i \rightsquigarrow \textsc{last}(L) \\ false & \exists \ell_i, t_j \neg ispred(\ell_i) \wedge \neg ispred(t_j) \wedge \ell_i \leftrightarrow t_j \in \mathbb{V} \wedge DCD(\ell_i) \leftrightarrow DCD(t_j) \in \mathbb{I} \wedge \\ & \quad DCD(t_j) \subset T \wedge L \subset DCD(\ell_i) \wedge t_j \rightsquigarrow \textsc{last}(T) \\ true & otherwise \end{cases}$$

$\textsc{split}(\mathbb{V}, \ell_i \leftrightarrow t_j)$ splits all the statement instance matchings based on the single instance matching $\ell_i \leftrightarrow t_j$.

$$\textsc{split}(L \leftrightarrow T \cup \mathbb{V}, \ell_i \leftrightarrow t_j) = \begin{cases} \{L \leftrightarrow T\} \cup \textsc{split}(\mathbb{V}, \ell_i \leftrightarrow t_j) & \ell_i \notin L \vee t_j \notin T \\ \{L_1\text{-}\ell_i \leftrightarrow T_1\text{-}t_j, L_2 \leftrightarrow T_2\} \cup \textsc{split}(\mathbb{V}, \ell_i \leftrightarrow t_j) & L \equiv L_1\text{-}\ell_i\text{-}L_2 \wedge T \equiv T_1\text{-}t_j\text{-}T_2 \\ \{L_1\text{-}\ell_i \leftrightarrow T_1\text{-}t_j, L_2 \leftrightarrow t_j\} \cup \textsc{split}(\mathbb{V}, \ell_i \leftrightarrow t_j) & L \equiv L_1\text{-}\ell_i\text{-}L_2 \wedge T \equiv T_1\text{-}t_j \\ \{L_1\text{-}\ell_i \leftrightarrow T_1\text{-}t_j, \ell_i \leftrightarrow T_2\} \cup \textsc{split}(\mathbb{V}, \ell_i \leftrightarrow t_j) & L \equiv L_1\text{-}\ell_i \wedge T \equiv T_1\text{-}t_j\text{-}T_2 \end{cases}$$

$\textsc{maxpref}(L_1, L, T_1, T, \mathbb{I})$ determines if L_1 and T_1 are the maximum prefixes of L and T that are also shared by some previously matched pairs in $\mathbb{I}$.

$$\textsc{maxpref}(L_1, L, T_1, T, \mathbb{I}) = \begin{cases} true & L_1 \subset L \wedge T_1 \subset T\ \wedge\ \exists L', T',\ (L' \leftrightarrow T' \in \mathbb{I}\ \wedge\ L_1 \subset L'\ \wedge\ T_1 \subset T') \wedge \\ & \neg \exists L_x, T_y, L'', T'',\ (L_1 \subset L_x \subseteq L\ \wedge\ T_1 \subset T_y \subseteq T\ \wedge\ L_x \subset L''\ \wedge\ T_x \subset T''\ \wedge\ L'' \leftrightarrow T'' \in \mathbb{I}) \\ false & otherwise \end{cases}$$

$\mathbb{V} \otimes_{\mathbb{I}} L \leftrightarrow T$: the cross product of the current instance matching $\mathbb{V}$ with a new control dep. matching $L \leftrightarrow T$, which may introduce new matchings.

$$\begin{array}{lll} \mathbb{V} \otimes_{\mathbb{I}} L \leftrightarrow T = \mathbb{V} \cup \{L \leftrightarrow T\}, & \text{if } \neg \exists L_1 \neq nil, T_1 \neq nil,\ \textsc{maxpref}(L_1, L, T_1, T, \mathbb{I}) & \text{[C-NEW]} \\ \mathbb{V} \otimes_{\mathbb{I}} L \leftrightarrow T = \mathbb{V}, & \text{if } \textsc{maxpref}(L, L, T, T, \mathbb{I}) & \text{[C-DUP]} \\ \mathbb{V} \otimes_{\mathbb{I}} L_1\text{-}L_2 \leftrightarrow T_1\text{-}T_2 = \textsc{split}(\mathbb{V}, \textsc{last}(L_1) \leftrightarrow \textsc{last}(T_1)) \cup \{L_2 \leftrightarrow T_2\}, & \text{if } \textsc{maxpref}(L_1, L_1\text{-}L_2, T_1, T_1\text{-}T_2, \mathbb{I}) & \text{[C-SPLIT]} \\ \mathbb{V} \otimes_{\mathbb{I}} L_1\text{-}L_2 \leftrightarrow T = \textsc{split}(\mathbb{V}, \textsc{last}(L_1) \leftrightarrow \textsc{last}(T)) \cup \{L_2 \leftrightarrow \textsc{last}(T)\}, & \text{if } \textsc{maxpref}(L_1, L_1\text{-}L_2, T, T, \mathbb{I}) & \text{[C-TAILA]} \\ \mathbb{V} \otimes_{\mathbb{I}} L \leftrightarrow T_1\text{-}T_2 = \textsc{split}(\mathbb{V}, \textsc{last}(L) \leftrightarrow \textsc{last}(T_1)) \cup \{\textsc{last}(L) \leftrightarrow T_2\}, & \text{if } \textsc{maxpref}(L, L, T_1, T_1\text{-}T_2, \mathbb{I}) & \text{[C-TAILB]} \end{array}$$

$$\frac{\neg\textsc{wellformed}(DCD(\ell_i), DCD(t_j), \mathbb{I})}{\langle \ell_i, t_j \rangle \cdot \mathbb{C}, \mathbb{I}, \mathbb{V} \longrightarrow \mathbb{C}, \mathbb{I}, \mathbb{V}} \quad \text{[UNMATCHED-EXPR]}$$

$$\frac{\textsc{wellformed}(DCD(\ell_i), DCD(t_j), \mathbb{I}) \quad \mathbb{I}' = \mathbb{I} \cup DCD(\ell_i) \leftrightarrow DCD(t_j) \quad \mathbb{V}' = (\mathbb{V} \otimes_{\mathbb{I}} DCD(\ell_i) \leftrightarrow DCD(t_j)) \cup \ell_i \leftrightarrow t_j}{\langle \ell_i, t_j \rangle \cdot \mathbb{C}, \mathbb{I}, \mathbb{V} \longrightarrow \mathbb{C}, \mathbb{I}', \mathbb{V}'} \quad \text{[MATCHED-EXPR]}$$

Figure 8: Instance Matching Rules. Symbol '-' in L_1-L_2 means concatenation.

The traversal is driven by $\mathbb{C}$. Rule [UNMATCHED-EXPR] is to handle a pair of expressions in $\mathbb{C}$ that were matched by sequence alignment but violate the well-formedness constraints. Function *wellformed*() determines if matching two label instance sequences L and T (denoting DCDs) causes any well-formedness violations. According to its definition in Fig. 8, it detects three kinds of violations. In the first case, if there is already a mapping $L_x \leftrightarrow T_y$ admitted to $\mathbb{I}$ in the past, and L_x is a prefix of L and T a prefix of T_y (or vice versa), there must be a cycle similar to Fig. 7 (a). The $\subset$ operator means prefix here. In Fig. 7 (a), there must be some matching between a statement instance in $L \ominus L_x$ (i.e., in L but not L_x) and some statement instance in T (e.g. `if(C3)` $\leftrightarrow$ `if(D1)`). Similarly, there must be some matching between an instance in $T_y \ominus T$ and some instance in L_x (e.g. `if(C1)` $\leftrightarrow$ `if(D3)`). Also because $L \ominus L_x$ must be control dependent on L_x (all these are valid control dependences) and $T_y \ominus T$ control dependent on T. A cycle of dependence is formed, violating [WF-X] (Section 3).

As illustrated in Fig. 7 (b), the second case describes that there is an existing expression matching $\ell_i \leftrightarrow t_j \in \mathbb{V}$ (whose DCD matching is hence in $\mathbb{I}$ which we will explain later), and the dynamic control dependence of ℓ_i, $DCD(\ell_i)$, is a prefix of L, and T is a prefix of $DCD(t_j)$. If $L \leftrightarrow T$, the last entry of L (i.e. `if(C2)`) must match with some entry in T (e.g. `if(D2)`). However, the matching becomes illegal if some predicate in $L \ominus DCD(\ell_i)$ (e.g. `if(x)`) is dependent on ℓ_i (i.e. `x=...`), because a cycle ℓ_i-`if(x)`-`if(C2)`-`if(D2)`... -t_j-ℓ_i is formed. The third case is symmetric.

Rule [MATCHED-EXPR] handles the case that the matching of the DCDs of an expression pair is well-formed. The control dependence mapping set $\mathbb{I}$ is updated by adding the control dependences of the matched expressions. The instance mapping set $\mathbb{V}$ is also updated so that some previous matched statement sets can be broken down to smaller (matched) subsets. Note that smaller matched sets mean finer granularity in matching. This is done by a *cross-product* operation between the control dependence mapping (of the symbolic expressions) and the current instance mapping set. The expression matching is also added to the result set.

The cross-product operation $\mathbb{V} \otimes_{\mathbb{I}} L \leftrightarrow T$ may introduce new mappings and split an existing mapping into multiple mappings. If L and T do not share any common prefixes with any existing control dependence mapping, $L \leftrightarrow T$ is added to $\mathbb{V}$ (Rule [C-NEW]). If they do share common prefixes with some existing mappings, which suggests that the existing mappings are too coarse grained, the existing mappings are hence refined. The mapping with the maximum common prefixes is identified through the *maxpref*() primitive. Assume the maximum common prefixes are L_1

and T_1, the existing mappings are split if they include the mapping $\textsc{last}(L_1) \leftrightarrow \textsc{last}(T_1)$ through the *split*() primitive ([C-SPLIT]). For example, assume a new mapping E-3_1-$6_1 \leftrightarrow E$ shares common prefix with an existing mapping E-3_1-$4_1 \leftrightarrow E$-5_1. The existing mapping is split by the last entries of the common prefixes $3_1 \leftrightarrow E$, resulting in two smaller mappings E-$3_1 \leftrightarrow E$ and $4_1 \leftrightarrow 5_1$. The suffices of L and T are also added to $\mathbb{V}$ as a new mapping. Rules [C-TAILA] and [C-TAILB] handle the corner cases that the maximum common prefix is one of L and T, in which the non-empty suffix is matched with the last entry of the prefix. This is the only legal mapping without introducing cycles.

□ *Example.* Table. 1 shows how the algorithm works on the traces in Fig. 3. The sequence alignment generates the initial $\mathbb{C}$ that identifies the longest sequence of equivalent symbolic expression pairs, as shown in the first row. Each row of the table represents one step of the algorithm that processes and removes a pair from $\mathbb{C}$. Columns 3 and 4 show the DCD mappings and instance mappings, after the rules specified in the last column are applied. At step one, matching the DCDs of 5_1 and 4_1 is well-formed. As such, the DCDs are added to both $\mathbb{I}$ and $\mathbb{V}$, and $5_1 \leftrightarrow 4_1$ is added to $\mathbb{V}$. At step two, matching the DCDs of 5_2 and 4_2 is also well-formed. Since $DCD(5_2) = E$-3_1-4_1-8_1-4_2 and $DCD(4_2) = E$-5_1, the cross product of their matching with $\mathbb{V}$ identifies that an existing mapping E-3_1-$4_1 \leftrightarrow E$ has the maximum common prefix with the new mapping. Hence the suffix mapping 8_1-$4_2 \leftrightarrow 5_1$ is added. Step three is similar. At step four, the DCD matching of 14_1 and 8_1 is well-formed. The cross product of their DCD matching E-3_1-$13_1 \leftrightarrow E$-5_1-5_2-7_3 with $\mathbb{V}$ not only induces the addition of $13_1 \leftrightarrow 5_1$-5_2-7_3 to $\mathbb{V}$, but also splits E-3_1-$4_1 \leftrightarrow E$ to E-$3_1 \leftrightarrow E$ and $4_1 \leftrightarrow E$ by the *split*() primitive. □

To handle implementation differences such as statement reordering (e.g. Fig. 2), Apex applies the aforementioned procedure iteratively until no more matchings can be found. In particular, after each round, the trace entries corresponding to the matched symbolic expressions that pass the well-formedness checks (i.e., those admitted by Rule [MATCHED-EXPR]) are removed from the traces. Note that the predicate instances representing control dependences are never removed even they are matched. This is to support well-formedness checks for the matchings in the following rounds. The same matching algorithm is then applied to the remaining traces. For example in Fig. 2 (a), all the entries corresponding to S1 are removed after the first round but the loop predicate instances are retained, which allows us to perform well-formed matching of S2 entries in the next round. As a result, the loop predicate on the left is correctly matched with the two loop predicates on the right.

Finally, the results in $\mathbb{V}$ denote the matchings between statement instances in the two versions. They correspond to the common bug-free behavior.

4.2 Phase (2): Residue Alignment

There are statement instances that cannot be matched, which are likely bug related. They may belong to statements unique to an implementation, or statements with some but not all their instances matched. For the latter case, it is highly desirable to *align* the unmatched instances of those statements such that it becomes clear why they do not match while they should have. This is very important for bug explanation. Apex further aligns these unmatched instances. It does so by generating a statement level mapping $\mathbb{M}$ between the two versions, from the matching results in the previous phase. Particularly, relation $\mathbb{M} : \mathcal{P}(\mathcal{P}(Label_a) \times \mathcal{P}(Label_b))$ indicates a set of statements in program a matches with a set of statements in b. It is generated by the following equation.

$$\langle L, T \rangle \in \mathbb{V} \cup \mathbb{I} \implies \langle \textsc{set}(L), \textsc{set}(T) \rangle \in \mathbb{M}$$

Function $\textsc{set}()$ turns a sequence of label instances to a set of labels (e.g. $\textsc{set}(E$-3_1-$4_1) = \{E, 3, 4\}$).

For the example in Fig. 1, $\mathbb{M} = \{\langle\{3, 4, 8\}, \{5\}\rangle, \langle\{5\}, \{4\}\rangle, \langle\{14\}, \{8\}\rangle ...\}$. It essentially denotes a common sub-program of the two as shown in Fig. 1.

Then Apex traverses the residue traces that contain the remaining unmatched symbolic expressions and all the predicate instances, and aligns trace entries based on the common sub-program $\mathbb{M}$ and well-formedness.

The alignment algorithm takes as input the two residue traces $\mathbb{T}_a$ and $\mathbb{T}_b$. Each trace entry is a triple consisting of the label instance, the symbolic expression *se* and the concrete value *v*. It traverses the buggy trace and looks for alignment for each instance. Basically, two instances are aligned if such alignment is compatible with the statement mappings $\mathbb{M}$ and aligning their dynamic control dependences is well-formed. Note that they are aligned but not matched. They have different (symbolic) values. The green nodes in Fig. 4 are such examples.

The rules are presented in Fig. 9. In Rule [UNALIGN-PRED], a predicate instance ℓ_i is discarded if there is no alignment. Note that since ℓ_i and t_j are predicates, they are concatenated to the dynamic control dependences (e.g. $DCD(\ell_i)$) for the well-formedness check. If multiple well-formed alignments exist, the first one is selected and $\mathbb{I}$ and $\mathbb{V}$ are updated accordingly (Rule [ALIGN-PRED]). The alignment of assignment instances is similar (Rules [UNALIGN-ASSIGN] and [ALIGN-ASSIGN]).

□ *Example.* Table 2 presents an example. The first two columns show the residue traces for the fibonacci executions. In the buggy run, since the value in 8_5 is incorrect, 9_1 is incorrectly executed and the loop is terminated. Outside the loop, the false branch of 13_2 is taken and the outer loop is also terminated. In the correct run, the predicate at 5_6 takes the false branch and one more round of fibonacci computation is performed until the true branch of 5_7 is taken and the loop is terminated. The next two columns show the dynamic control dependences of the first trace entries.

#	$\mathbb{C}$	$\mathbb{I}$	$\mathbb{V}$	rules applied
1	$\{\langle 5_1, 4_1\rangle, \langle 5_2, 4_2\rangle, \langle 5_3, 4_3\rangle, \langle 14_1, 8_1\rangle\}$	E-3_1-$4_1 \leftrightarrow E$	E-3_1-$4_1 \leftrightarrow E, 5_1 \leftrightarrow 4_1$	[M-EXPR,C-NEW]
2	$\{\langle 5_2, 4_2\rangle, \langle 5_3, 4_3\rangle, \langle 14_1, 8_1\rangle\}$	E-3_1-$4_1 \leftrightarrow E,\ E$-$3_1$-$4_1$-$8_1$-$4_2 \leftrightarrow E$-$5_1$	E-3_1-$4_1 \leftrightarrow E,\ 8_1$-$4_2 \leftrightarrow 5_1, 5_1 \leftrightarrow 4_1, 5_2 \leftrightarrow 4_2$	[M-EXPR,C-SPLIT]
3	$\{\langle 5_3, 4_3\rangle, \langle 14_1, 8_1\rangle\}$	E-3_1-$4_1 \leftrightarrow E,\ E$-$3_1$-$4_1$-$8_1$-$4_2 \leftrightarrow E$-$5_1$, E-3_1-4_1-8_1-4_2-8_2-$4_3 \leftrightarrow E$-5_1-5_2	E-3_1-$4_1 \leftrightarrow E,\ 8_1$-$4_2 \leftrightarrow 5_1, 8_2$-$4_3 \leftrightarrow 5_2$, $5_1 \leftrightarrow 4_1, 5_2 \leftrightarrow 4_2, 5_3 \leftrightarrow 4_3$	[M-EXPR,C-SPLIT] [M-EXPR,C-SPLIT]
4	$\{\langle 14_1, 8_1\rangle\}$	E-3_1-$4_1 \leftrightarrow E,\ E$-$3_1$-$4_1$-$8_1$-$4_2 \leftrightarrow E$-$5_1$, E-3_1-4_1-8_1-4_2-8_2-$4_3 \leftrightarrow E$-5_1-5_2, E-3_1-$13_1 \leftrightarrow E$-5_1-5_2-7_3	E-$3_1 \leftrightarrow E,\ 4_1 \leftrightarrow E,\ 8_1$-$4_2 \leftrightarrow 5_1, 8_2$-$4_3 \leftrightarrow 5_2$, $5_1 \leftrightarrow 4_1, 5_2 \leftrightarrow 4_2, 5_3 \leftrightarrow 4_3, 13_1 \leftrightarrow 5_1$-$5_2$-$7_3, 14_1 \leftrightarrow 8_1$	[M-EXPR,C-SPLIT]

Table 1: Applying the Algorithm in Fig. 8 to Traces in Fig. 3.

$$\frac{ispred(\ell_i) \quad \neg\exists\langle t_j, se1, v1\rangle \in \mathbb{T}_b,\ \langle \textsc{set}(DCD(\ell_i)\text{-}\ell_i), \textsc{set}(DCD(t_j)\text{-}t_j)\rangle \in \mathbb{M}\ \wedge\ \textsc{wellformed}(DCD(\ell_i)\text{-}\ell_i, DCD(t_j)\text{-}t_j, \mathbb{I})}{\langle \ell_i, se, v\rangle \cdot \mathbb{T}_a, \mathbb{T}_b, \mathbb{I}, \mathbb{V} \ \longrightarrow\ \mathbb{T}_a, \mathbb{T}_b, \mathbb{I}, \mathbb{V}} \quad \text{[UNALIGN-PRED]}$$

$$\frac{\begin{array}{c} ispred(\ell_i) \quad \mathbb{T}_b = \mathbb{T}'_b \cdot \langle t_j, se1, v1\rangle \cdot \mathbb{T}''_b \quad \langle \textsc{set}(DCD(\ell_i)\text{-}\ell_i), \textsc{set}(DCD(t_j)\text{-}t_j)\rangle \in \mathbb{M} \quad \textsc{wellformed}(DCD(\ell_i)\text{-}\ell_i, DCD(t_j)\text{-}t_j, \mathbb{I}) \\ \neg\exists\langle t'_k, se2, v2\rangle \in \mathbb{T}'_b,\ \langle \textsc{set}(DCD(\ell_i)\text{-}\ell_i), \textsc{set}(DCD(t'_k)\text{-}t'_k)\rangle \in \mathbb{M}\ \wedge\ \textsc{wellformed}(DCD(\ell_i)\text{-}\ell_i, DCD(t'_k)\text{-}t'_k, \mathbb{I}) \\ \mathbb{I}' = \mathbb{I} \cup DCD(\ell_i)\text{-}\ell_i \leftrightarrow DCD(t_j)\text{-}t_j \quad \mathbb{V}' = \mathbb{V} \otimes_{\mathbb{I}} DCD(\ell_i)\text{-}\ell_i \leftrightarrow DCD(t_j)\text{-}t_j \end{array}}{\langle \ell_i, se, v\rangle \cdot \mathbb{T}_a, \mathbb{T}_b, \mathbb{I}, \mathbb{V} \ \longrightarrow\ \mathbb{T}_a, \mathbb{T}''_b, \mathbb{I}', \mathbb{V}'} \quad \text{[ALIGN-PRED]}$$

$$\frac{\neg ispred(\ell_i) \quad \neg\exists\langle t_j, se1, v1\rangle \in \mathbb{T}_b,\ \langle \ell, t\rangle \in \mathbb{M} \wedge \textsc{wellformed}(DCD(\ell_i), DCD(t_j), \mathbb{I})}{\langle \ell_i, se, v\rangle \cdot \mathbb{T}_a, \mathbb{T}_b, \mathbb{I}, \mathbb{V} \ \longrightarrow\ \mathbb{T}_a, \mathbb{T}_b, \mathbb{I}, \mathbb{V}} \quad \text{[UNALIGN-ASSIGN]}$$

$$\frac{\begin{array}{c} \neg ispred(\ell_i) \quad \mathbb{T}_b = \mathbb{T}'_b \cdot \langle t_j, se1, v1\rangle \cdot \mathbb{T}''_b \quad \langle \ell, t\rangle \in \mathbb{M} \quad \textsc{wellformed}(DCD(\ell_i), DCD(t_j), \mathbb{I}) \\ \neg\exists\langle t'_k, se2, v2\rangle \in \mathbb{T}'_b,\ \langle \ell, t'\rangle \in \mathbb{M} \wedge \textsc{wellformed}(DCD(\ell_i), DCD(t'_k), \mathbb{I}) \\ \mathbb{I}' = \mathbb{I} \cup DCD(\ell_i) \leftrightarrow DCD(t_j) \quad \mathbb{V}' = (\mathbb{V} \otimes_{\mathbb{I}} DCD(\ell_i) \leftrightarrow DCD(t_j)) \cup \ell_i \leftrightarrow t_j \end{array}}{\langle \ell_i, se, v\rangle \cdot \mathbb{T}_a, \mathbb{T}_b, \mathbb{I}, \mathbb{V} \ \longrightarrow\ \mathbb{T}_a, \mathbb{T}''_b, \mathbb{I}', \mathbb{V}'} \quad \text{[ALIGN-ASSIGN]}$$

Figure 9: Residue Alignment.

$\mathbb{T}_a$	$\mathbb{T}_b$	DCD$_a$	DCD$_b$	$\mathbb{I}$	$\mathbb{V}$	rules applied
$8_5 \cdot 9_1 \cdot 13_2 \cdot 3_3 \cdot 16_1$	$5_6 \cdot 7_6 \cdot 8_2 \cdot 4_7 \cdot 5_7 \cdot 11_1$	L	T	$\mathbb{I}_0 \cup \{L$-$8_5 \leftrightarrow T$-$5_6\}$	$\mathbb{V}_0 \cup \{8_5 \leftrightarrow 5_6\}$	[A-P]
$9_1 \cdot 13_2 \cdot 3_3 \cdot 16_1$	$7_6 \cdot 8_2 \cdot 4_7 \cdot 5_7 \cdot 11_1$	L-8_5	T-5_6	$\mathbb{I}_0 \cup \{L$-$8_5 \leftrightarrow T$-$5_6\}$	$\mathbb{V}_0 \cup \{8_5 \leftrightarrow 5_6\}$	[U-A]
$13_2 \cdot 3_3 \cdot 16_1$	$7_6 \cdot 8_2 \cdot 4_7 \cdot 5_7 \cdot 11_1$	E-3_1-3_2	T-5_6	$\mathbb{I}_0 \cup \{L$-$8_5 \leftrightarrow T$-5_6, E-3_1-3_2-$13_2 \leftrightarrow T$-5_6-$7_6\}$	$\mathbb{V}_0 \cup \{8_5 \leftrightarrow 5_6, 13_2 \leftrightarrow 5_4$-$5_5$-$5_6$-$7_6\}$	[A-P]
$3_3 \cdot 16_1$	$8_2 \cdot 4_7 \cdot 5_7 \cdot 11_1$	E-3_1-3_2	T-5_6-7_6	$\mathbb{I}_0 \cup \{L$-$8_5 \leftrightarrow T$-5_6, E-3_1-3_2-$13_2 \leftrightarrow T$-5_6-$7_6\}$	$\mathbb{V}_0 \cup \{8_5 \leftrightarrow 5_6, 13_2 \leftrightarrow 5_4$-$5_5$-$5_6$-$7_6\}$	[U-A]
...	...	...	...	...	...	

Table 2: Applying the Algorithm in Fig. 9 to the Residue Traces in the Fibonacci Executions. Let $L = E$-3_1-3_2-4_4-8_3-4_5-8_4-4_6, $T = E$-5_1-5_2-5_3-5_4-5_5, $\mathbb{I}_0 = \{...E$-3_1-$13_1 \leftrightarrow E$-5_1-5_2-$7_3, ..., L \leftrightarrow T\}, \mathbb{V}_0 = \{E$-$3_1 \leftrightarrow E, 3_2 \leftrightarrow 5_1$-$5_2$-$5_3, ..., 8_4$-$4_6 \leftrightarrow 5_5, 5_5 \leftrightarrow 4_5\}$. [A-P] stands for [ALIGN-PRED].

In the first step, the alignment $8_5 \leftrightarrow 5_6$ is added to $\mathbb{V}$ as the statement mapping $\langle\{E, 3, 4, 8\}, \{E, 5\}\rangle \in \mathbb{M}$ and the alignment of control dependences is well-formed. It corresponds to the first green node in Fig. 4. Next, no alignment is found for 9_1 (i.e. red node in Fig. 4). In the third step, 13_2 and 7_6 are aligned (despite their different branch outcomes), corresponding to the second green node. □

4.3 Phase (3): Comparative Dependence Graph Construction, Slicing, and Feedback Generation

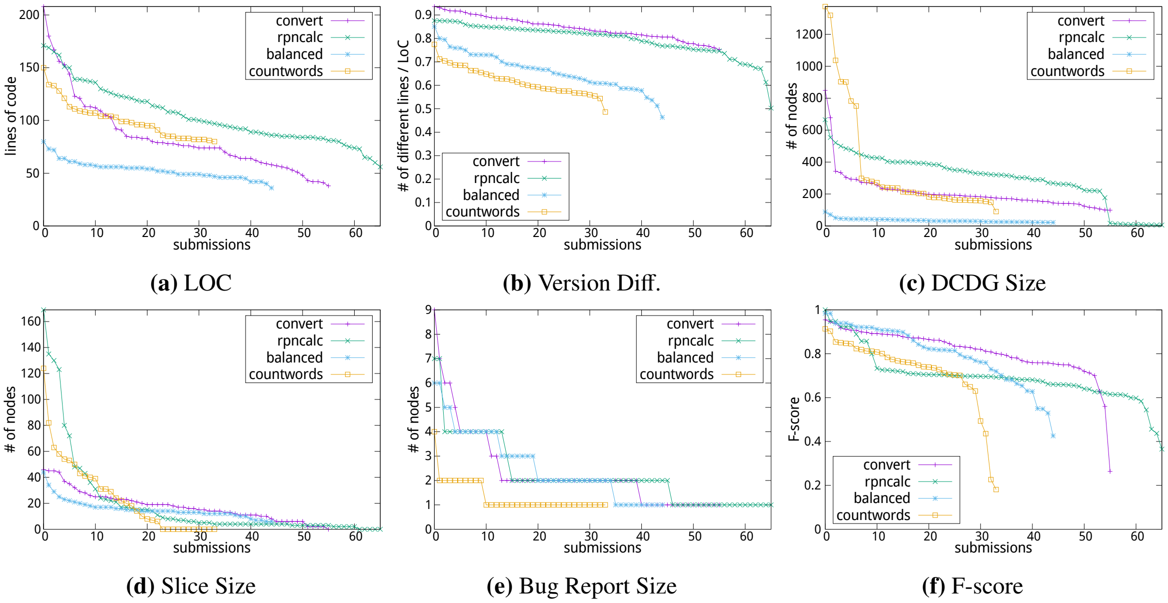
Apex generates the DCDG from the matching and alignment results. Fig. 4 represents an example graph. It further computes a *comparative slice* from the graph. The slicing criterion consists of the instances that emit the different outputs. The slice captures the internal differences that caused the output differences. It is computed by graph traversal, which starts from the criterion, going backward along dependence edges. If a plain node (for matched instances) is reached, no traversal is beyond the node. Our tool follows a set of rules to generate the bug report from a slice. For example, a pair of aligned but unmatched assignment instances $\ell_i \leftrightarrow t_j$, is translated to "*the value at ℓ_i should have been $v(t_j)$ instead of $v(\ell_i)$*". The report for the fibonacci bug can be found at the end of Section 2. Details are elided.

5. Implementation and Evaluation

The tracing component of Apex that collects symbolic and concrete traces is implemented using LLVM. The SMT solver used is Z3 [24]. The rest is implemented in Python. The experiments were conducted on an Intel Core i7 machine running Arch Linux 3.17.1 with 16GB RAM. All the benchmarks, the failure inducing inputs, and the bug reports by Apex are available on the project site [4].

5.1 Experiment with Real Student Submissions

We have acquired 4 programming assignments from a recent programming course at the authors' institute: `convert` turns a number with one radix into another radix; `rpncalc` evaluates a postfix expression using a stack; `balanced` checks if the input string has a valid nesting of parentheses and brackets; `countwords` counts the frequency of words. The number of buggy versions ranges from 33-65 for each submission. The total number of buggy submissions is 205.



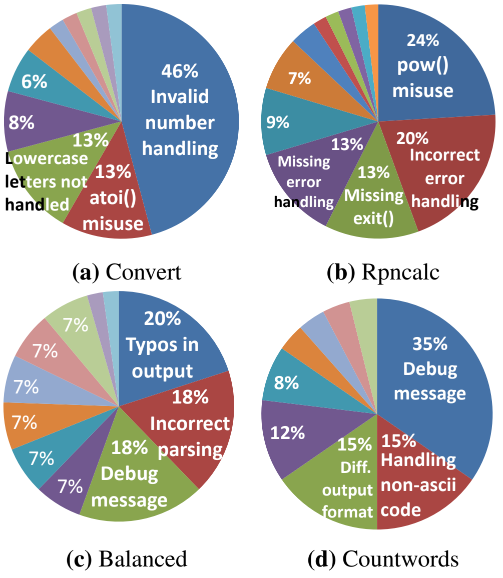
(a) LOC (b) Version Diff. (c) DCDG Size (d) Slice Size (e) Bug Report Size (f) F-score

Figure 10: Student Submission Results. On each figure, the submissions are sorted by the Y-axis values.

For each buggy version, we have the failure inducing input and the patched version (submitted later by the students). For each assignments, we have the instructor's solution. We applied APEX to each failing run. The results are presented in Fig. 10. The execution time ranges from 1 to 20 seconds with most finishing in a few seconds.

From Fig. 10a, the submission LOC ranges from 60-210. Fig. 10b measures the syntactic differences between the submissions and the instructor's version (i.e. edit distance over LOC sum). Observe that they are substantially different. From Fig. 10c, the computed DCDG has 10-1300 nodes. Some have a small DCDG because of the simplicity of the test case (e.g., testing input validation). From Fig. 10d, the comparative slices have 2-160 nodes. The large slices usually correspond to cases in which the buggy program has substantially different states from the correct program. However, as shown in Fig. 10e, the bug reports are very small. According to our experience with students, succinct bug reports without too much low level details are important for usability. We have a number of methods to reduce bug report size, including coalescing the repetitive instances (from loops), and avoiding presenting detailed causality in large unmatched code regions, which often occur when the correct program terminates quickly due to invalid inputs but the buggy program goes on as normal, or vice versa. We cross-checked the root causes reported by APEX with the patched versions and found that APEX identifies the correct root causes for 195 out of 205 cases. Here, when we say APEX identifies a root cause, we mean that the root cause is reported as the first entry in the causal explanation just like the example in Section 2.

Fig. 10f shows the F-score [41] of execution matching, including both assignment and predicate matchings. F-score is a weighted average of precision and recall that reflects the percentage of matching. Here, *precision/recall means the percentage of the statement instances in the failing/passing run that have matches in the other party*, and F-score $F = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$. Observe that APEX is able to match a lot of instances for many cases. Some have almost everything matched. These bugs are usually due to typos in outputs.



(a) Convert (b) Rpncalc

(c) Balanced (d) Countwords

Figure 11: Student Bug Classifications.

Student Bugs. To better demonstrate the effectiveness of APEX in identifying root causes and matching executions, we further generate a grading report for each assignment by classifying the bugs based on the root causes. However, the root causes in the bug reports by Apex only contain artifacts from the buggy programs, which are very different from each other. It is hence difficult to classify based on bug reports directly. Fortunately, Apex has the matchings to the correct version, which is stable across all bugs. We hence classify bugs based on the projection of the root cause in the correct version. Intuitively, with Apex we are able to classify by *the part that is not correctly implemented by the student*. The results are shown in Fig. 11. We have the following observations. **(1)** Most bugs fall into a few main categories. For example in `rpncalc`, 24% of bugs are due to the incorrect parameter order of the `pow()` function. The reason is that parameters are stored in the stack order so that they need to be flipped before calling `pow()`. In `convert`, almost half of the students forgot to check if an input character is legal for the radix. Such information is very useful for the instructor as they indicate where to improve. **(2)** Typos in final outputs are very common (e.g. `printf("String not balanced.\n")` versus `printf("String is not balanced.\n")` in `balanced`). For these cases, a simple automatic grading policy that counts the number of passing runs by comparing outputs would give 0 credit. In contrast, Apex would allow partial credit by execution matching (e.g., the F-score). In these cases, the students will get almost full credit. **(3)** Apex missed the root cause for 10 out of 205 cases. We further inspected these cases. Most of them are because the buggy run is so wrong that there are very few matched symbolic expressions to begin with. For example in `rpncalc`, the instructor and most students used predicates whereas two students used table look-up to drive execution like in a compiler frontend. However, the table indexing is wrong. As such, almost the entire sequence of (symbolic) values are wrong. **(4)** Although from the reports many bugs have simple root causes, it does not mean they are easier for Apex as identifying them requires matching the substantially different program structures. There are also subtle bugs. But their number is relatively small.

5.2 Experiment with `stackoverflow.com` Programs

To better evaluate applicability, we have collected 15 pairs (buggy vs. correct) of implementations from `stackoverflow.com`. They were mainly posted in 2014. The benchmarks and their urls are presented in the first two columns of Table 3. The benchmarks are named after the algorithms they implement. Each row represents two programs. The sizes of each pair are presented in the third column. The programs are by different programmers. The fourth column presents the size of the symbolic trace, i.e. the number of symbolic expressions for assignments, excluding all simple copies and the assignments that are not data dependent on inputs. The `time` column shows the time taken to match all symbolic expression pairs. This is to prepare for the iterative instance matching algorithm. The last column shows the number of equivalent pairs. Observe that the number of pairs may be much larger than the number of expressions in the individual versions because one expression may be symbolically equivalent to many. Also observe that the time taken is not substantial as Apex uses concrete value comparison to prune the candidate pairs. That is, we only compare symbolic equivalence when two expressions have the same concrete value.

benchmark	url	LOC	# sym expr	time	# matches
knapsack-1	[9]	56 / 78	69 / 210	4.65s	401
matrix-mult	[11]	53 / 53	421 / 421	9.56s	1534
fibonacci-sum	[14]	35 / 29	22 / 28	0.79s	22
kadane	[8]	43 / 29	22 / 26	0.25s	34
euclid	[5]	23 / 22	17 / 10	0.66s	5
dijkstra	[2]	57 / 64	79 / 76	1.85s	219
mergesort	[12]	47 / 70	135 / 231	8.66s	150
span-tree	[13]	71 / 75	153 / 135	7.60s	1499
floyd	[7]	46 / 47	154 / 173	8.51s	1892
dijkstra-2	[3]	61 / 64	121 / 76	2.00s	303
euler	[15]	44 / 27	110 / 81	21.74s	167
gt_product	[16]	27 / 27	315 / 330	164.05s	866
binarysearch	[1]	25 / 27	32 / 37	1.60s	27
euclid-2	[6]	31 / 21	8 / 10	0.52s	5
knapsack-2	[10]	33 / 42	60 / 109	1.29s	41

Table 3: Benchmarks and Symbolic Expression Matching.

Table 4 shows the instance matching results. The `size` column shows the number of LLVM IR statement instances in the execution traces. We have excluded all the copy operations and short-cut the corresponding dependences for brevity. The “Matched" column shows the number of instances that are matched, and the percentage (e.g. for the `knapsack` case, 80/221 = 36% whereas 80/417 = 19%). The A&U column shows those aligned but not matched. The U&U column shows those neither aligned nor matched. The next two columns show the graph and the slice sizes (in nodes). The last column shows the root causes reported by Apex. Symbol ‘-’ means that Apex misses the real root cause. Observe that Apex can align and match roughly half of the instances. It can also align part of the unmatched instances. Those instances are usually closely related to bugs. Depending on the semantic differences, the unaligned and unmatched parts may still be large. For example, 81% of the instances in the correct version of `knapsack` cannot be matched or aligned. This is because the correct execution is much longer. Also observe that most of comparative slices are small, much smaller than the graph sizes. More importantly, in most cases, the root of the slice precisely identifies the real root cause as mentioned in the online bug report. Recall that the root of a slice is the A&U or U&U instances whose parents are matched. Benchmark `matmult` has an exceptionally large slice. That is because the buggy execution has largely corrupted state. An array index computation is wrong such that most values are wrong from the beginning. All such faulty values are part of the slice. Interestingly, Apex can still match and align most of the control structures and part of the computation and it also precisely pinpoints the root cause. Since these unmatched instances belong to a few statements (in loops), the bug report is still very small.

benchmark	size	time	# Matched (%/%)	# A&U (%/%)	# U&U(%)		G.size	S.size	root cause
					buggy	correct			
knapsack	221 / 417	0.51s	80(36/19)	0(0/0)	141(64)	337(81)	558	51	s.h. `if (weight<w2 && w1>=weight)` ...
matmult	514 / 514	0.62s	247(48/48)	33(6/6)	179(35)	179(35)	638	312	s.n. `FIRST[c*M+k]` s.h. `FIRST[c*N+k]` at line 25
fibonacci-sum	38 / 42	0.34s	21(55/50)	2(5/4)	15(40)	19(46)	59	19	`if (i1+i0>n)` s.h. `false` s.n. `true` at line 26
kadane	40 / 49	0.34s	26(65/53)	4(10/8)	10(25)	19(39)	59	5	s.h. `if (0<=cumulativeSum)` inside line 15.
euclid	21 / 15	0.32s	6(29/40)	1(5/7)	9(43)	5(33)	21	8	s.n. `b*(a/b)` s.h. `a%b` at line 7.
dijkstra	195 / 150	0.40s	68(35/45)	0(0/0)	125(64)	77(51)	270	18	s.n. `visited[vert]=1` at line 43.
mergesort	178 / 294	0.53s	118(66/40)	17(10/6)	21(12)	117(40)	273	31	s.n. `arr[mid]=arr[start]` s.h. `new_array[i] = arr[high]` at line 31.
span-tree	389 / 324	0.51s	210(54/65)	2(1/1)	177(46)	112(35)	501	6	-
floyd	286 / 351	0.51s	210(73/60)	2(1/1)	177(62)	112(32)	501	16	s.h. `if (path[i][k] != INT_MAX...)` at line 30
dijkstra-2	192 / 150	0.42s	39(20/26)	0(0/0)	82(43)	67(45)	188	22	s.n. `for (...;i<nr_airport;i++)` s.h. `while (cur_vertex != END)` at line 24.
euler1	133 / 91	0.38s	71(53/78)	0(0/0)	50(38)	11(12)	132	61	s.h. `return 0` at line 43.
gt_product	546 / 473	0.80s	318(58/67)	5(1/1)	117(21)	29(6)	469	88	s.n. `product = product * ((int) NUM[j] - 48)` at line 16.
binarysearch	46 / 54	0.36s	21(46/39)	2(4/4)	13(28)	18(33)	54	6	s.n. `retval=0` at line 22.
euclid-2	11 / 15	0.33s	2(18/13)	0(0/0)	4(36)	5(33)	11	2	s.h. `q = (r[0] % r[1])` at line 15
knapsack-2	87 / 174	0.44s	11(13/6)	1(1/1)	53(61)	80(46)	145	45	-

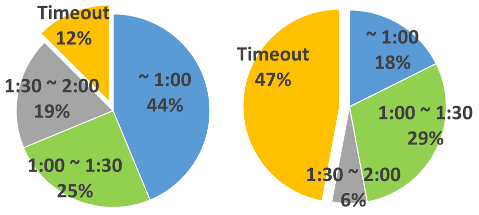
Table 4: Instance Matching. "s.h." stands for "should have", "s.n." for "should not"

`Mergesort` is an interesting case. The buggy code compares values l from the lower half and h from the higher half of an array and directly swaps the values if h is smaller than l. It uses one loop while the correct code uses four loops. Apex was able to match the control structures and recognize that the buggy code needs an additional array instead of direct swapping. In particular, Apex identifies an unmatched additional array assignment within a matched branch in the correct run. In `dijkstra`, the two implementations are substantially different. They use different values to denote if a node has been visited. Moreover, a loop in the correct version corresponds to two separate loops in the buggy version. Apex was able to match the control structures and correctly explain the bug. In `dijkstra-2`, a nesting loop in one version corresponds to a few consecutive loops in the other. Details can be found at [4].

Apex misses the root cause for two cases: `span-tree` that computes the minimal spanning tree and `knapsack-2`. The reason is that the buggy programs used algorithms different from that used by the correct version. Apex currently does not support matching across algorithms (e.g., bubble sort vs. merge sort). If different algorithms are allowed, we plan to follow the same strategy as in [26], which is to let the instructor provide a set of possible algorithms beforehand. We can also use Apex to match *passing runs*. Algorithmic differences will yield poor matchings in passing runs. We will leave it to future work.

5.3 User Study

We have evaluated Apex with 34 undergraduate students who take the C programming course at the authors' institute. We have partitioned the students into 2 groups, one using Apex and the other not. We requested them to implement `convert` in Sec. 5.1 in a two-hour lab. Our research questions are: 1. Can Apex help the overall productivity of the students? 2. Can Apex help understand bugs?



(a) w/ Apex **(b)** wo/ Apex

Figure 12: Time taken by students to finish the task

	(a) w/ Apex	(b) wo/ Apex
Avg	4417.6s	5742.4s
SD	1670.7	1606

Figure 13: Average and standard deviation of time took by each group in seconds

We have implemented a script that records the students' activities, including each compilation, each test run, each revision, and each invocation of our tool. The completion time is what a student took to pass all the test cases (11 in total). Fig. 12 shows the results of the two groups. In group (a) (with Apex), only 12% of the students could not finish the task in time. On the other hand, in group (b), 47% of the students could not finish. This supports that Apex can help the students' productivity in programming assignments with 99% certainty (following the A/B significance test). In group (a), 44% finished within an hour while only 18% in group (b). We have also inspected the suggestions Apex generated. Most of them have only 2-3 lines, which imply that Apex did not disclose too much about the correct version. Fig. 13 presents the average and standard deviation of time took by each group considering timeouts as 2 hours. On average students in group (a) took about 67 minutes and students in group (b) took 74 minutes. On average group (a) can complete the task about

23% faster than group (b). We also performed the t-test with our data. The p-value is 0.0316 and with 95% confidence our system can help students' productivity.

A. Suggestions are easy to understand.
B. Suggestions are useful to locate error.
C. Suggestions are useful to understand errors.
D. Suggestions are useful to understand correct algorithm.
E. Suggestions are useful to fix errors.
F. Suggestions are useful for overall productivity.

Figure 14: Questions

Question	Agree	Disagree	Neutral
A	56%	6%	38%
B	61%	11%	28%
C	72%	6%	22%
B+C	78%	11%	11%
D	56%	22%	22%
E	83%	0%	17%
D+E	94%	0%	6%
F	78%	0%	22%

Figure 15: Students' response to the questions

In order to evaluate the quality of our suggestions, we surveyed the participants in group (a). We asked them 6 questions as in Fig. 14. We classify the questions into 4 groups. First, question A asks if the suggestions in pseudo code can be easily apprehended. Second, questions B and C ask if the student can understand their problems more easily with our system. Third, questions D and E ask whether Apex can provide hints on how to fix the problems. Last, question F asks the overall effectiveness of our tool. Fig. 15 shows the responses. We have the following observations. First, while 6% of the students complained about difficulties in understanding the suggestion pseudo code, the presentation of the suggestions could be improved. More details are disclosed in Section 5.3.1.

Second, 78% of the students agreed that our tool is useful in either locating or understanding errors. Oral communication with the students discloses that they seem to have very diverse understanding about where the root causes are. Third, 94% of the students agreed that they can get hints on fixing the problems. It was very much appreciated by the students that Apex can present correction suggestions in the context of their code (e.g., using their variables). Last, 78% of the students agreed that our tool can help the overall productivity. This is consistent with the results in Fig. 12.

5.3.1 Limitations

One student had a very interesting comment that although she got her bug fixed by copying a constant value in the correction suggestion, she did not understand why she should use the constant. We inspected her case. The buggy code is shown in Fig. 16. This code is for converting an integer digit into an alphanumeric digit: converting 10 into A, 11 into B, and so on. The operation at line 4 should be "`digit + 'A' - 10`". Apex precisely reported the root cause and suggested the proper correction. However, the suggestion is simply a line of pseudo code "`digit + 55`". This is because Apex internally operates on the IR level so that letters are all represented as constant values which lack semantic meanings and operations on constants are unfolded.

```
1 // Convert an integer digit into a character digit.
2 if ('A' <= digit && digit <= 'Z')
3   // SUGGESTIONS
4   ??? digit = digit + 'A';  // (BUGGY)
5   // Instead,
6   +++ digit = digit + 55    // (CORRECT)
7
8   // The constant 55 means 'A' - 10
```

Figure 16: Student Buggy Code and the Suggestion.

We plan to address the problem by adding annotations or textual debugging hints to the instructor's version. In the former example, line 3 could be commented with a debugging hint such as "*It is likely that the constant you use to transform a value to a letter is wrong*". Instead of showing the pseudo code, the instructor can configure the tool to emit the textual hint. Together with the (faulty) variable values emitted by Apex, the student should be able to quickly understand the bug. Note that in order to provide high quality textual hints, internally Apex should capture the precise bug causality and identify the corresponding correct code.

5.4 Comparison with PMaxSat

We have also implemented a version of Apex directly based on the PMaxSat formulation. We used Z3 as the PMaxSat solver. We compare the performance and the quality of execution alignment of the two versions. We set the timeout of PMaxSat to 5 minutes and ran it for the stackoverflow cases and the convert cases. The results are shown in Table 5. In most of the cases, PMaxSat is much slower than Apex. In 3 out of the 15 stackoverflow cases, PMaxSat could not find the solution in 5 minutes. On average our system can find the alignment in less than 2 seconds, whereas PMaxSat requires more than 90 seconds. The results for the convert cases (the last row) are similar. Note that the high overhead of PMaxSat may not be acceptable for the students, especially during labs.

In terms of execution alignment, the two versions generate similar results. On average the precision of the approximate version of Apex is more than 79% and the recall is more than 74%, compared to the PMaxSat version. This indicates that most alignments discovered by the approximate version are identical to those by PMaxSat. The most common cases of alignment differences are constant operations that do not depend on the inputs such as initializing variables with 0 and increasing loop variables by 1. In our observation, these operations have very little effect on the generated suggestions.

We compared the performance with WPM3-2015-in [20], a state-of-art incomplete partial maxsat solver. The incomplete solvers can find the solution incrementally and hence they can produce intermediate results as soon as possible. We measured the time took by the WPM3 solver until it finds a solution that can satisfy the same number of the clauses as the solution found by Apex. Table 6 presents the comparison result. Though the incomplete solver can reach the similar solution faster than Z3 which is a complete solver, Apex is more than 10 times faster on average.

Program	Run time (s)		Precison	Recall	F-score
	Apex	PMaxSat			
binarysearch	1.00	14.19	0.91	0.99	0.95
dijkstra	1.58	30.46	0.77	0.77	0.77
dijkstra-2	1.18	6.64	0.70	0.74	0.72
euclid	0.30	.16	0.80	0.80	0.80
euclid-2	0.25	.14	0.71	0.83	0.77
euler1	0.93	1.34	0.97	0.94	0.95
fibonacci-sum	1.06	1.44	0.96	0.88	0.92
floyd	5.04	> 300	-	-	-
gt_product	1.72	> 300	-	-	-
kadane	1.33	4.25	0.60	0.64	0.62
knapsack	1.52	14.28	0.69	0.80	0.74
knapsack-2	1.32	162.58	0.67	0.72	0.69
matmult	0.93	53.75	0.88	1.00	0.93
mergesort	1.48	7.42	0.77	0.91	0.84
span-tree	1.37	> 300	-	-	-
average	1.40	90.21	0.79	0.84	0.81
Convert (average)	2.30	46.02	0.88	0.74	0.79

Table 5: Comparison between Apex and PMaxSat

Program	Run time (s)	
	Apex	WPM3
binarysearch	1.00	2.54
dijkstra	1.58	5.50
dijkstra-2	1.18	1.77
euclid	0.30	0.14
euclid-2	0.25	0.11
euler1	0.93	0.57
fibonacci-sum	1.06	0.74
floyd	5.04	71.68
gt_product	1.72	103.62
kadane	1.33	2.31
knapsack	1.52	2.09
knapsack-2	1.32	2.26
matmult	0.93	2.62
mergesort	1.48	2.78
span-tree	1.37	19.1
average	1.40	14.52
Convert (average)	2.30	20.63
Rpncalc (average)	1.53	52.86
Balanced (average)	0.88	24.15

Table 6: Comparison between Apex and incomplete solver

5.5 Experiment with IntroClass Benchmarks

We have also evaluated Apex with the IntroClass Benchmarks [35]. The benchmark is designed for evaluation of automated software repair techniques. Out of the 6 projects, we have selected 5 projects with 710 buggy implementations: `checksum` computes the checksum of input string; `digits` prints each digit of the input number; `grade` computes a letter grade for the input score; `median` finds a median number among the 3 input numbers; `syllables` counts the frequencies of vowels in the input string. We did not select `smallest` because it is too small (usually a few lines).

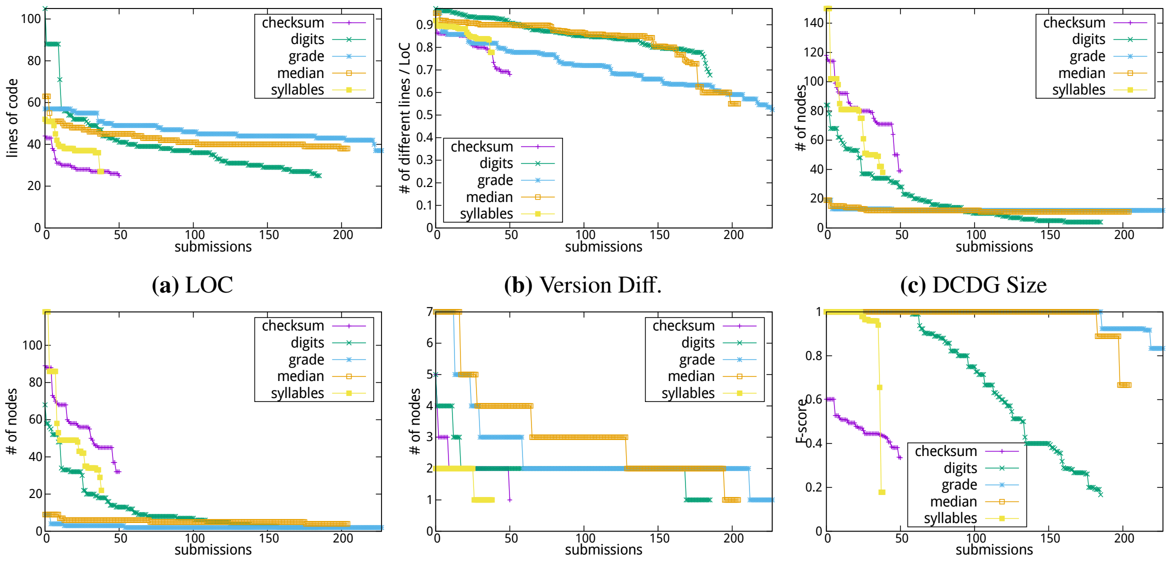
The results are shown in Fig. 17. From Fig. 17a, the program sizes are mostly 40-60 LOC. Fig. 17b suggests that the programs are very different from the solution version syntactically. Fig. 17c shows that the DCDGs have 10-150 nodes and some projects such as `median` and `grade` have mostly less than 20 nodes. This is because these programs have no loop and their executions are very short. Fig. 17e presents that our suggestions are very small.

Regarding the bugs and the quality of suggestions, we have the following observations. **(1)** For 57 out of 710 cases, Apex missed the root cause. This happens mostly in `median`. The programs in this project have neither loops nor arithmatic operations. They have at most 6 comparisons. In buggy executions, there are usually insufficient evidence for Apex to achieve good alignment. **(2)** Most bugs are due to missing/incorrect conditions, missing computation, or typos in output messages. For example, in `checksum`, 76% of the submissions failed because of missing a modulo operation. Detailed breakdown for all projects can be found at [4]. Note that such information is very useful for the instructors.

6. Related Work

In [43], program synthesis was used to automatically correct a buggy program according to the correct version. The technique requires the instructor to provide a set of correction rules. In [26], a technique was proposed to detect algorithms used in functionally correct student submissions by comparing the value sequences. Then it provides feedback prepared by the instructor for each type of algorithm. It is complementary to ours as we could use it to detect cases in which the students are using a different algorithm. Equivalence checking [33, 34] determines if two programs are equivalent. If not, it generates counter-examples. In [32], equivalence checking is extended to look for a single value replacement that can partially fix the faulty state. The value replacement is reported as the root cause. In [22], a technique was proposed to identify the weakest precondition that triggers the behavioral differences between two program versions. These techniques do not align/match the intermediate states, which is critical to understanding failure causality.

There have been works on debugging using the passing and failing executions of *the same program*, by mutating states [23, 44, 46] or slicing two executions [45]. In contrast, Apex assumes different programs. There are also works on comparing traces from program versions to understand regression bugs [28, 40]. They perform sequence alignment in traces without using symbolic analysis. There are satisfiability based techniques that also strive to explain failures within a single program [25, 29, 30, 39], and even fix the bugs through templates [31]. They do not leverage the correct version. Fault localization [21, 36, 42] leverages a large number of passing and failing runs to identify root causes. In our experience, many buggy student submissions fail on all inputs. Besides, they usually do not explain causality or



(a) LOC

(b) Version Diff.

(c) DCDG Size

(d) Slice Size

(e) Bug Report Size

(f) F-score

Figure 17: IntroClass Benchmark Results. On each figure, the submissions are sorted by the Y-axis values.

provide fix suggestions. LAURA [19] statically matches the buggy and correct implementations and reports mismatches. TALUS [37] recognizes the algorithm from the students' submissions and projects the correct implementation to the submissions to generate feedback.

7. Conclusion

We present Apex, a system that explains programming assignment bugs. It leverages both symbolic and dynamic analysis to match and align statement instances from the executions of the buggy code and the correct code. A comparative slice is computed from the unmatched parts of the two runs, starting from the different outputs. The algorithm is an approximate solution to the underlying PMAX-SAT problem. The experiments show that Apex can accurately identify the root causes and explain causality for 94.5% of the 205 student bugs and 15 bugs collected from Internet. The evaluation on a standard benchmark set with over 700 student bugs shows similar results. A user study in the classroom shows that Apex has substantially improved student productivity.

Acknowledgments

We thank the anonymous reviewers for their constructive comments. This research was supported, in part, by DARPA under contract FA8650-15-C-7562, NSF under awards 1409668 , 1320444, and 1320306, ONR under contract N000141410468, and Cisco Systems under an unrestricted gift. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

References

[1] What is wrong with my binary search implementation? `http://stackoverflow.com/questions/21709124`.

[2] Dijkstra's algorithm not working. `http://stackoverflow.com/questions/14135999`, .

[3] Logical error in my implementation of dijkstra's algorithm. `http://stackoverflow.com/questions/10432682`, .

[4] Apex benchmarks. `http://apexpub.altervista.org/`.

[5] Euclid algorithm incorrect results. `http://stackoverflow.com/questions/16567505`, .

[6] Inverse function works properly, but if works after while loops it produces wrong answers. `http://stackoverflow.com/questions/22921661`, .

[7] Bug in my floyd-warshall c++ implementation. `http://stackoverflow.com/questions/3027216`.

[8] Is this an incorrect implementation of kadane's algorithm? `http://stackoverflow.com/questions/22927720`.

[9] Knapsack algorithm for two bags. `http://stackoverflow.com/questions/20255319`, .

[10] Is there something wrong with my knapsack. `http://stackoverflow.com/questions/21360767`, .

[11] Incorrect result in matrix multiplication in c. `http://stackoverflow.com/questions/15512963`.

[12] Merge sort implementation. `http://stackoverflow.com/questions/18141065`.

[13] Prims alghoritm. `http://stackoverflow.com/questions/24145687`.

[14] What is wrong with this algorithm? `http://stackoverflow.com/questions/18794190`, .

[15] Project euler problem 4. `http://stackoverflow.com/questions/7000168`, .

[16] Project euler 8, i don't understand where i'm going wrong. http://stackoverflow.com/questions/23824570, .

[17] Stackoverflow. http://www.stackoverflow.com.

[18] Analysis: The exploding demand for computer science education, and why america needs to keep up. http://www.geekwire.com/2014/analysis-examining-computer-science-education-explosion/, 2014.

[19] A. Adam and J.-P. Laurent. Laura, a system to debug student programs. *Artificial Intelligence*, 15(1):75–122, 1980.

[20] C. Ansótegui, F. Didier, and J. Gabàs. Exploiting the structure of unsatisfiable cores in maxsat. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI '15, pages 283–289. AAAI Press, 2015. ISBN 978-1-57735-738-4.

[21] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Directed test generation for effective fault localization. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 49–60, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-823-0.

[22] A. Banerjee, A. Roychoudhury, J. A. Harlie, and Z. Liang. Golden implementation driven software debugging. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 177–186, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-791-2.

[23] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 342–351, New York, NY, USA, 2005. ACM. ISBN 1-58113-963-2.

[24] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-78799-2, 978-3-540-78799-0.

[25] A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. *International Journal on Software Tools for Technology Transfer*, 8(3):229–247, June 2006. ISSN 1433-2779.

[26] S. Gulwani, I. Radiček, and F. Zuleger. Feedback generation for performance problems in introductory programming assignments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '14, pages 41–51, New York, NY, USA, 2014. ACM.

[27] D. S. Hirschberg. Algorithms for the longest common subsequence problem. *Journal of ACM*, 24(4):664–675, Oct. 1977. ISSN 0004-5411.

[28] K. J. Hoffman, P. Eugster, and S. Jagannathan. Semantics-aware trace analysis. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 453–464, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1.

[29] M. Jose and R. Majumdar. Cause clue clauses: Error localization using maximum satisfiability. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 437–446, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8.

[30] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso. Minthint: Automated synthesis of repair hints. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE '14, pages 266–276, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5.

[31] R. Könighofer and R. Bloem. Automated error localization and correction for imperative programs. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, FMCAD '11, pages 91–100, Austin, TX, 2011. FMCAD Inc. ISBN 978-0-9835678-1-3.

[32] S. Lahiri, R. Sinha, and C. Hawblitzel. Automatic rootcausing for program equivalence failures in binaries. In *Proceedings of the 27th International Conference on Computer Aided Verification*, CAV'15, pages 362–379, Berlin, Heidelberg, 2015. Springer-Verlag. ISBN 978-3-319-21689-8.

[33] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 712–717, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-31423-0.

[34] A. Lakhotia, M. D. Preda, and R. Giacobazzi. Fast location of similar code fragments using semantic 'juice'. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, PPREW '13, pages 5:1–5:6, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1857-0.

[35] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering (TSE)*, 41(12):1236–1256, December 2015. ISSN 0098-5589.

[36] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI'03*, 2003.

[37] W. R. Murray. Automatic program debugging for intelligent tutoring systems. *Computational Intelligence*, 3(1):1–16, 1987.

[38] G. C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 83–94, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2.

[39] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 772–781, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3.

[40] M. K. Ramanathan, A. Grama, and S. Jagannathan. Sieve: A tool for automatically detecting variations across program versions. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 241–252, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2579-2.

[41] C. J. V. Rijsbergen. *Information Retrieval*. Butterworth-Heinemann, Newton, MA, USA, 2nd edition, 1979. ISBN 0408709294.

[42] S. K. Sahoo, J. Criswell, C. Geigle, and V. Adve. Using likely invariants for automated software fault localization. In *Pro-*

ceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, pages 139–152, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1870-9.

[43] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 15–26, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6.

[44] W. N. Sumner and X. Zhang. Comparative causality: Explaining the differences between executions. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 272–281, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3.

[45] D. Weeratunge, X. Zhang, W. N. Sumner, and S. Jagannathan. Analyzing concurrency bugs using dual slicing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 253–264, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-823-0.

[46] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '02/FSE-10, pages 1–10, New York, NY, USA, 2002. ACM. ISBN 1-58113-514-9.