Verifix: Verified Repair of Programming Assignments

UMAIR Z. AHMED, National University of Singapore, Singapore

ZHIYU FAN, National University of Singapore, Singapore

JOOYONG YI, Ulsan National Institute of Science and Technology, Korea

OMAR I. AL-BATAINEH, National University of Singapore, Singapore

ABHIK ROYCHOUDHURY, National University of Singapore, Singapore

Automated feedback generation for introductory programming assignments is useful for programming education. Most works try to generate feedback to correct a student program by comparing its behavior with an instructor's reference program on selected tests. In this work, our aim is to generate verifiably correct program repairs as student feedback. The student assignment is aligned and composed with a reference solution in terms of control flow, and differences in data variables are automatically summarized via predicates to relate the variable names. Failed verification attempts for the equivalence of the two programs are exploited to obtain a collection of maxSMT queries, whose solutions point to repairs of the student assignment. We have conducted experiments on student assignments curated from a widely deployed intelligent tutoring system. Our results indicate that we can generate verified feedback in up to 58% of the assignments. More importantly, our system indicates when it is able to generate a verified feedback, which is then usable by novice students with high confidence.

1 INTRODUCTION

CS-1, the introductory programming course, is an undergraduate course offered by Universities and Massive Open Online Courses (MOOCs) across disciplines. Several programming assignments are typically attempted by the students as a part of this course, which are evaluated and graded against pre-defined test-cases. Given the importance of programming education and the difficulty of providing relevant feedback for the massive number of students, there has been increasing interest in automated program repair techniques for providing automated feedback to student assignments (APR) [9, 12, 13, 25, 28, 29].

Existing approaches and their limitations. Table 1 provides a summary of state-of-the-art APR works, and compares them with our approach Verifix. The repair rate of the state-of-the-art APR techniques [9, 12, 28] is astonishingly high, around 90%. However, these works make certain assumptions such as the presence of tests.

Many student assignment feedback generation approaches assume the existence of a complete set of high quality test-cases to validate their repairs. Over-fitting the repair to an incomplete specification is a well known problem of APR tools [8, 23]. Prior studies have shown that trivial repairs such as functionality deletion alone can achieve ~50% accuracy on buggy student programs given a weak oracle [5]. Generating complex incorrect feedback that merely passes all the tests can potentially confuse novice students more than expert programmers. Novice students when provided with incorrect/partial repair feedback that merely pass more tests, have been shown to struggle more, as compared to expert programmers given the same feedback [29]. Hence, we suggest that the feedback given to novice students needs rigorous quality assurance, whenever possible.

Tool	Completely Automated	Beyond Identical Reference CFG	Verified Repair
Clara [9]	✓	✗	✗
SarfGen [28]	✓	✗	✗
ITSP [29]	✓	✓	✗
Refactory [12]	✓	✓	✗
CoderAssist [13]	✗	✓	✓
Verifix	✓	✓	✓

Table 1. Programming assignment repair tools comparison.

```
 1      int check_prime(int n)
 2      {
 3        if (n == 1)
 4          return 0;
 5        int j;
 6        for(j=2; j<n; j++)
 7        {
 8          if (n%j == 0)
 9            return 0;
10
11
12        }
13        return 1;
14      }
```

```
 1      int check_prime(int n)
 2      {
 3
 4
 5        int i =1;
 6        while(i<=n−1)
 7        {
 8          if (n%i != 0)
 9            ; // No-Op
10          else
11            return 0;
12        }
13        return 1;
14      }
```

```
 1      int check_prime(int n)
 2      {
 3        if (n == 1)
 4          return 0;
 5        int i=2;
 6        while(i<=n−1)
 7        {
 8          if (n%i != 0)
 9            i++;
10          else
11            return 0;
12        }
13        return 1;
14      }
```

(a) The correct reference program   (b) The incorrect student program   (c) The repaired program by Verifix

Fig. 1. Motivating example for the *Prime Number* programming assignment. Existing tools such as Clara [9] and Sarfgen [28] cannot repair the incorrect student program in Fig 1(b) since its Control-Flow Graph (CFG) differs from the CFG of instructor designed reference program in Fig 1(a). Our tool Verifix generates the repaired program in Fig 1(c), which is verifiably equivalent to the reference implementation, due to superior Control-Flow Automata (CFA) based abstraction.

In a related vein, some approaches, in particular recent ones [9, 28], assume the existence of multiple reference programs. This assumption is made to overcome the difficulty of generating feedback when the Control-Flow Graph (CFG) structure of the student program is different from the instructor provided reference program.[1] The problem is that the existing approaches collect multiple reference programs from student submissions that pass all tests, again without verifying their correctness. It is well-known that verifying the equivalence of two program is challenging [6].

Insight. Many of the aforementioned problems of the existing APR techniques can be addressed with verified repair. We can assume only the presence of one reference assignment, which is always available in educational settings and can be given by an instructor. We then create a verifiably correct repair of the student assignment. In other words, the repaired student assignment will be semantically equivalent to the reference assignment given by the instructor. In terms of workflow,

---

[1]SarfGen [28] and Clara [9] require that the control-flow structure of student and reference programs should be exactly the same. Clara [9] demands aligned variables to be evaluated into the same sequence of values at runtime, which often does not hold for early stage programs.

the repair engine indicates when it can generate a verified repair as feedback, and when it does, the students can receive a feedback which is guaranteed to be correct. In other words, we can have greater confidence or trust on the feedback generated by the repair tool. Furthermore, student programs that are verified to be correct after repair can be used as additional trustworthy reference programs in future .

Contribution: Verified repair. In this work, we propose a general approach to verified repair, and show that verified repair can be performed with a single reference program. Verified repair engenders greater trust in the output of the automatic repair tool, which has been identified to be a key hindrance in deployment of automated program repair [24]. We show that verified repair is feasible and achievable in a reasonable time scale (less than 30 seconds) for student programming assignments of a large public university. This shows the promise of using verified repair to generate high confidence *live* feedback in programming pedagogy settings. To the best of our knowledge, ours is the first work to espouse verified repair for general purpose programming education. The only previous attempt on verified repair [13] is tightly tied to a specific structure of programs implementing dynamic programming.

Repair tool: Verifix. We build our verified-repair technique by extending the existing program equivalence checking technique. Although automatically proving the equivalence between two programs remains challenging (mainly due to the difficulty of automatically finding loop invariants), we found that student programs are in many cases amenable for equivalence checking. This is because there is usually a reference program whose structure is similar to the student program, as shown in earlier works [9, 28]. Exploiting this, Verifix produces a verified repair. Note that, Verifix peforms repair and equivalence checking at once. More concretely, Verifix aligns the incorrect student program with the reference program into an aligned automaton, derives alignment relation to relate the variable names of the two programs, and suggests repairs for the code captured by the edges of the aligned automaton via Maximum Satisfiability-Modulo-Theories (MaxSMT) solving. We use MaxSMT to find a minimal repair. Our approach can generate a program behaviourally equivalent to the reference program while preserving the original control-flow of the student program as much as possible. This leads to smaller patches/feedback which we believe are easier to comprehend, in general. We evaluate our approach on student programming submissions curated from a widely used intelligent tutoring system. Our approach Verifix produces small-sized verified patches as feedback, which, whenever available, can be used by struggling students with high confidence.
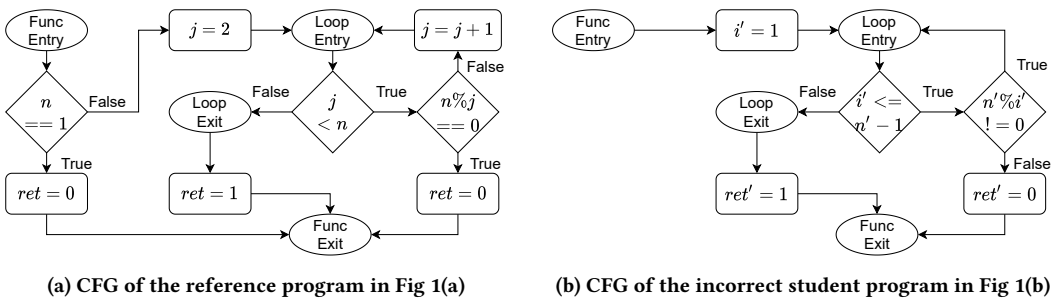

(a) CFG of the reference program in Fig 1(a)

(b) CFG of the incorrect student program in Fig 1(b)

Fig. 2. **Control Flow Graph (CFG) of the reference and incorrect program listed in Fig 1. Incorrect program CFG in Fig 2(b) differs from reference program CFG in Fig 2(a) due to a missing return node. Existing tools like Clara [9], Sarfgen [28] cannot repair the incorrect program.**
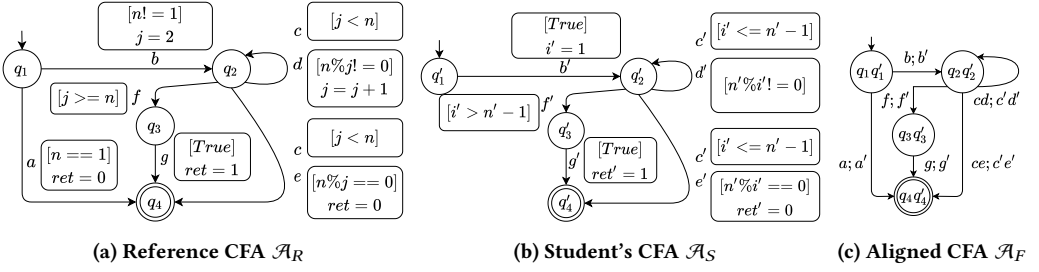
(a) Reference CFA $\mathcal{A}_R$ (b) Student's CFA $\mathcal{A}_S$ (c) Aligned CFA $\mathcal{A}_F$

Fig. 3. Control Flow Automata (CFA) of the reference and incorrect program listed in Fig 1. CFA $\mathcal{A}_R$ of reference program in Fig 3(a) is structurally aligned with CFA $\mathcal{A}_S$ of student program in Fig 3(b) to obtain an aligned CFA $\mathcal{A}_F$ in Fig 3(c).

2   OVERVIEW

Consider a simple programming assignment for checking whether a given number n is a prime number. Figure 1(a) shows a reference implementation prepared by an instructor, and Figure 1(b) shows an incorrect program submitted by students.

Limitations of the Existing Approaches. The state-of-the-art approaches such as Clara [9] and Sarfgen [28] make the same-control-flow assumption described as follows.

Assumption 1. To perform a repair, a given incorrect program and its reference implementation should have the same control-flow structure.[2]

Notice that this assumption does not hold for the incorrect program shown in Figure 1(b). While the reference program has an if statement whose successors are either the for loop in lines 4–8 or the end of the program, the incorrect program starts with a while loop without having a preceding if statement. As a result, both Clara and Sarfgen fail to generate a repair.

A common approach that has been used to overcome this problem is to use multiple reference programs of diverse control-flow structures [9, 13, 28]. Since it would be labor-intensive for an instructor to prepare multiple reference implementations, recent works (e.g., [9, 28]) gets around this problem by using student submissions. That is, a student submissions that pass all tests are added into a pool of reference implementations.

Our Approach. We show how we address the aforementioned limitations. Essentially, we do not make the same control-structure assumption, by conducting repair with Control Flow Automata (CFA) where its nodes represent program locations and its edges represent guarded commands (see Figure 3 as an example). Also, we extend the existing equivalence checking technique into a verified repair technique. In the following, we show our repair algorithm works through the following three phases: the setup phase, the verification phase, and the repair phase. The last two steps occur simultaneously as explained in the following.

2.1   Setup Phase

In the setup phase, we model the given reference and student programs as Control Flow Automata (CFA) with the nodes representing control-flow locations and the edges representing guarded commands, as shown in Figure 3. It is possible for the guard to be true, in which case the command is executed unconditionally. Following which we prepare multiple aligned automata $\mathcal{A}_F$ using a

---

[2]For the definitions of control-flow structure used in Clara and Sarfgen, refer to Definition 4.1 of [9] and Definition 3.1 of [28]

syntax-based alignment technique (see Section 4.1). Figure 3 shows an example of $\mathcal{A}_F$ obtained on aligning the reference automata in Figure 3(a) and student automata in Figure 3(b).

The aligned automata $\mathcal{A}_F$ in Figure 3 consists of an aligned control entry node q_1q_1', control exit node q_4q_4', a loop entry node q_2q_2', and a loop exit node q_3q_3'. The notation

$$q_2q_2' \xrightarrow{cd;c'd'} q_2q_2'$$

denotes that $q_2 \xrightarrow{cd} q_2$ is aligned with $q_2' \xrightarrow{c'd'} q_2'$ where cd represents a sequence of edges c and d. Note that in the case of $q_1q_1' \xrightarrow{aa'} q_4q_4'$, a new edge a' is introduced to connect two aligned nodes, q_1q_1' and q_4q_4'. In general, it is possible that more than one syntactic aligned automaton can be created from two given assignments, in which case we enumerate over each aligned automaton to find out a minimal repair.

Given an aligned automaton, the verification of behavioral equivalence between two assignments can be performed by checking whether $q \sim q'$ (i.e., q is bisimilar to q') holds for each aligned nodes q and q', e.g. in Figure 3(c), q_i is aligned with q_i' for $i \in \{1, 2, 3, 4\}$.

So, the core task we have is to prove $q \sim q'$ for a pair of aligned nodes q and q', and also to generate a repair if the proof fails. We annotate each node with a variable alignment predicate. Variable alignment predicate provides a bijective mapping between the variables of two programs, which is necessary to prove their semantic equivalence. In our example, we obtain the following variable alignment predicate (automatically): $\{ret \leftrightarrow ret', n \leftrightarrow n', j \leftrightarrow i'\}$ where ret is a special variable holding the return value of the function under verification/repair.

2.2 Verification Phase

We perform verification for all aligned automata $\mathcal{A}_F$. If verification succeeds for $\mathcal{A}_F$ or its repaired variation, semantic equivalence between student and reference programs is guaranteed (see Theorem 1). Verification is performed inductively for individual edge, starting from the outgoing edges of the initial node of $\mathcal{A}_F$ (aa' and bb' for our Figure 3).

Consider the edge $q_1q_1' \xrightarrow{b;b'} q_2q_2'$. Given this edge, we should prove the following: when $q_1 \sim q_1'$ is assumed, $q_2 \sim q_2'$ holds after executing $b; b'$. We achieve this by checking

$$\varphi_{edge}^1 : \phi_{q_1q_1'} \wedge \psi_r \wedge \psi_s^1 \wedge \neg\phi_{q_2q_2'}$$

where $\phi_{q_1q_1'}$ and $\phi_{q_2q_2'}$ denote the variable alignment predicates at node q_1q_1' and q_2q_2', respectively.

$$\phi_{q_1q_1'}: \quad (ret_0 = ret_0') \wedge (n_0 = n_0') \wedge (j_0 = i_0')$$
$$\phi_{q_2q_2'}: \quad (ret_1 = ret_1') \wedge (n_1 = n_1') \wedge (j_1 = i_1')$$

Meanwhile, ψ_r and ψ_s^1 denote the guarded commands of b and b', respectively, in a Single Static Assignment (SSA) form, where

$$\psi_r: \quad (n_0 \neq 1 \implies j_1 = 2) \quad \wedge \quad (\neg(n_0 \neq 1) \implies j_1 = j_0)$$
$$\psi_s^1: \quad (True \implies i_1' = 1) \quad \wedge \quad (\neg True \implies i_1' = i_0')$$

If φ_{edge}^1 is satisfiable, then $q_2 \sim q_2'$ does not hold, indicating verification failure. We check the satisfiability of φ_{edge}^1 using an off-the-shelf SMT solver, Z3 [20].

2.3 Repair Phase

For our running example, the SMT solver Z3 finds that φ_{edge}^1 is satisfiable under a certain assignment ϕ_{ce}^1 which is

$$\phi_{ce}^1 : n_0 = n_0' = 1, j_0 = i_0' = 0$$

Block	Student Transition		Repaired Transition	
a'	$\emptyset$		$[n' == 1]$	$ret' = 0$
b'	$[True]$	$i' = 1$	$[n'! = 1]$	$i' = 2$
c'	$[i' <= n' - 1]$		$[i' <= n' - 1]$	
d'	$[n'\%i'! = 0]$		$[n'\%i'! = 0]$	$i' = i' + 1$
e'	$[n'\%i' == 0]$		$[n'\%i' == 0]$	$ret' = 0$
f'	$[i' > n' - 1]$		$[i' > n' - 1]$	
g'	$[True]$	$ret' = 1$	$[True]$	$ret' = 1$

Table 2. Incorrect student blocks and their corresponding repairs generated by Verifix, after multiple rounds of edge verification-repair of Figure 3 aligned automaton.

We call this generated satisfying assignment ϕ_{ce}^1 as a *counter-example*, since it is a counter-example to $q_2 \sim q_2'$ holding. Using this counter-example ϕ_{ce}^1, we perform a repair based on counter-example-guided inductive synthesis or CEGIS strategy [26]. Following CEGIS strategy, we look for a repair of ψ_s^1 which rules out the counter-example ϕ_{ce}^1. Verifix returns two potential repair candidates.

$$\psi_s^2: \quad (False \implies i_1' = 1) \quad \wedge \quad (\neg False \implies i_1' = i_0')$$
$$\psi_s^3: \quad (n_0' \neq 1 \implies i_1' = 1) \quad \wedge \quad (\neg(n_0' \neq 1) \implies i_1' = i_0')$$

Then, we repeat edge verification φ_{edge}^1 step with each of the obtained repair candidates. In our example, verification attempt fails again for both repair candidates, that is φ_{edge}^1 is satisfiable. and a new counter-example ϕ_{ce}^2 is obtained.

$$\phi_{ce}^2 : n_0 = n_0' = 2, i_0 = i_0' = 0$$

By considering both ϕ_{ce}^1 and ϕ_{ce}^2, Verifix returns a new repair candidate ψ_s^4,

$$\psi_s^4: \quad (n_0' \neq 1 \implies i_1' = 2) \quad \wedge \quad (\neg(n_0' \neq 1) \implies i_1' = i_0')$$

This repair candidate ψ_s^4 rules out both the counter-examples seen so far, and no further satisfying assignments of φ_{edge}^1 are found. This completes the verification and repair, thereby repairing the edge b' in Figure 3(b). Table 2 summarizes the buggy student automata $\mathcal{A}_S$ edges and the corresponding repairs generated by our repair tool Verifix.

Verifix searches for a minimal repair, for each aligned edge under consideration. Informally, a minimal repair modifies the minimum number of expressions, and a more formal description is available in Theorem 4. Minimal repair is achieved by reducing our repair search into a partial MaxSMT problem. Intuitively, a minimal repair should preserve the maximum number of the original expressions. Given the original guarded action of the student program, ψ_s^1, we first replace each of the original expressions of ψ_s^1 with a unique hole. For example, $[True] \implies (i_1' = 1)$ is transformed into $[h_1] \implies (i_1' = h_2)$ where each hole h_i can be filled in with a patch candidate expression chosen from the implementation space of h_i. Clearly, the implementation space of h_i also includes the original expression. Verifix retains the original expressions for as many holes as possible, by associating a higher weight penalty with the original expressions over other replacement expressions. Such a choice of weights ensures that the original expressions are favored by the partial MaxSMT solver. Further details are provided in Section 5.2.

3 PROGRAM MODEL

Prior to explaining our alignment and verification-repair procedures, we introduce the key structures used to model programs.

Abstract Syntax Tree (AST). An Abstract Syntax Tree (AST) consists of a set of nodes representing the abstract programming constructs. With the tree hierarchy, or edges, representing the relative ordering between the appearance of these constructs. We extend the standard AST with special labels

for two node types: *Func-Entry* and *Loop-Entry*. Each AST consists of a root node corresponding to a function definition, which is labelled as a function-entry node. Similarly, every loop construct in the AST is labelled as a loop-entry node.

The AST for motivating example shown in Figure 1 consists of two labelled nodes: a *Func-Entry* node q_1 which maps to the *check_prime* function definition and a *Loop-Entry* node q_2 which maps to the for-loop construct. We note that some existing APR techniques for programming assignments, like ITSP [29] which uses GenProg [15], operate on program ASTs directly.

Control Flow Graph (CFG). Existing state-of-art APR techniques like Clara [9] and SarfGen [28] operate at the level of CFG, whose nodes are basic blocks and edges denote control transfer. We extend the standard CFG by introducing 4 types of special labelled nodes: {*Func-Entry*, *Loop-Entry*, *Func-Exit*, and *Loop-Exit*}; denoting the program states when control enters a function or a loop, and when control exits a function or a loop, respectively. The *Func-Entry* and *Loop-Entry* CFG nodes correspond with control entering AST nodes of the same type. The *Func-Exit* and *Loop-Exit* CFG nodes correspond with the program state after control visits the last child of *Func-Entry* and *Loop-Entry* AST node, respectively. These *Func-Exit* and *Loop-Exit* program states can also be reached by altering the control-flow using *return* and *break* statements, respectively.

Figures 2(a) and 2(b) depict the CFG of the reference and student program in Figures 1(a) and 1(b), respectively. These CFGs contain four special nodes denoting *Func-Entry* (q_1/q_1'), *Loop-Entry* (q_2/q_2'), *Loop-Exit* (q_3/q_3'), and *Func-Exit* (q_4/q_4') program states.

Control Flow Automata (CFA). Our tool Verifix operates at the level of the control flow automaton (CFA), often used by model-checking and verification communities [11]. The CFA is essentially the CFG, with code statements labeling the edges of CFA, instead of code statements labeling nodes as in CFG. The nodes of our CFA are annotated with the node types mentioned earlier: *Func-Entry*, *Loop-Entry*, *Func-Exit*, and *Loop-Exit*. The edges of our CFA are constructed by choosing all possible code transitions between the program states in CFG. Depending on the reason for control-flow transition, these edges can be of three types: *normal*, *return* or *break*. Figures 3(a) and 3(b) depict the CFA modeled using the reference and student CFG in Figures 2(a) and 2(b), respectively.

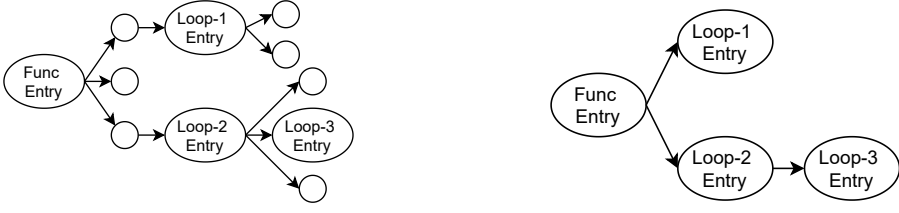We provide our precise definition of CFA in the following.

Definition 1 (Control Flow Automata). *A Control Flow Automata (CFA) is a tuple of the form* $\langle V, E, v^0, v^t, \Omega, \Psi, Var \rangle$, *where:*

- V : *is a finite set of vertices (or nodes) of the automata, representing function and loop entry/exit program states,*
- $E \subseteq V \times V$, *is a finite set of edges of the automata representing normal, break, and return transitions between program states,*
- v^0 : *is the initial node representing function entry state,*
- v^t : *is the terminal node representing function exit state,*
- $\Omega : \{u \leftrightarrow v \mid \forall u \in V, \exists v \in V\}$, *for each function/loop entry node, maintains a mapping to the corresponding exit node,*
- Ψ *is mapping from edge e to ψ_e for all edges e, where ψ_e is the set of guarded actions labeling e*
- Var, *is the set of variables used in $\bigcup_e \psi_e$*

For edge e in the CFA, ψ_e is thus the code statements labeling e.

4 ALIGNED AUTOMATA

Our methodology for repairing incorrect student programs relies on constructing an aligned automaton $\mathcal{A}_F$ from the given student automaton $\mathcal{A}_S$ and the reference automaton $\mathcal{A}_R$. The construction of the automaton $\mathcal{A}_F$ consists of following steps: (i) Model the student and reference

(a) Example *AST* with labelled and unlabelled nodes.   (b) Example AST^L after deletion of unlabelled nodes.

Fig. 4. **Example demonstrating Abstract Syntax Tree (AST) transformation to retain nodes labelled as function and loop entry.**
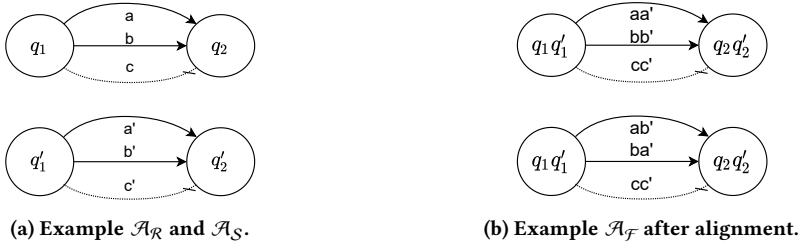

(a) Example $\mathcal{A}_{\mathcal{R}}$ and $\mathcal{A}_{\mathcal{S}}$.   (b) Example $\mathcal{A}_{\mathcal{F}}$ after alignment.

Fig. 5. **Example demonstrating edge alignment. Given node alignment** $V : \{q_1 q_1', q_2 q_2'\}$, **the edges are aligned based on type. The single** *break* **transitions** c **and** c' **are aligned with each other, while the multiple** *normal* **edges are aligned combinatorially to produce two unique aligned automata.**

programs as Control Flow Automata $\mathcal{A}_S$ and $\mathcal{A}_R$, (ii) the structural alignment of $\mathcal{A}_S$ and $\mathcal{A}_R$, (iii) the inference of the variable alignment predicates.

4.1 Structurally Aligning $\mathcal{A}_S$ and $\mathcal{A}_R$

For constructing aligned automata $\mathcal{A}_{\mathcal{F}}$, we first construct a node alignment between the program states of reference and student automata. Followed by aligning the transition edges between the reference and student program states.

Node Alignment. Given two Control Flow Automata $\mathcal{A}_S$ and $\mathcal{A}_{\mathcal{R}}$, and their corresponding Abstract Syntax Trees AST_S and AST_R for student and reference program, respectively, we construct node alignment $V : V_S \leftrightarrow V_R$ as follows.

(1) Delete all unlabelled nodes from AST_S and AST_R to obtain AST_S^L and AST_R^L, respectively. An AST^L consists of only *Func-Entry* and *Loop-Entry* labelled nodes.
(2) If the syntactic tree structure of AST_S^L and AST_R^L is same, align each node of AST_S^L with AST_R^L and add to V. This step aligns the *Func-Entry* and *Loop-Entry* nodes of $\mathcal{A}_S$ and $\mathcal{A}_{\mathcal{R}}$.
(3) For each pair of entry nodes (either *Func-Entry* or *Loop-Entry*) that are aligned with each other, their corresponding exit nodes (either *Func-Exit* or *Loop-Exit*) are aligned with each other.

For constructing node alignment V, we first align the labelled nodes of student and reference Abstract Syntax Tree (AST). The labelled AST nodes can be of two types: *Func-Entry* and *Loop-Entry*. These labels are same as those in $\mathcal{A}_S$ and $\mathcal{A}_R$, but we take advantage of the tree structure in the AST. Figure 4 demonstrates unlabelled *AST* node deletion in step-1 through an example, after which only the *Func-Entry* and *Loop-Entry* labelled nodes are retained. For the reference program (respectively

student program) listed in Figure 1, the labelled AST_R^L (resp. AST_S^L) consists of two nodes $q_1 \to q_2$ (resp. $q_1' \to q_2'$). Since both the AST^L trees are structurally similar, the node alignment V consists of $\{q_1 q_1', q_2 q_2'\}$ after step-2 of node alignment, denoting the *Func-Entry* and the *Loop-Entry* aligned nodes.

The step-3 of node-alignment finally aligns the function and loop exit nodes. Given the student and reference automata in Figure 3, q_4, which is the *Func-Exit* node corresponding to q_1, is aligned with q_4', which is the *Func-Exit* node corresponding to q_1'. Similary, the *Loop-Exit* nodes q_3 and q_3' are aligned, since their corresponding *Loop-Entry* nodes q_2 and q_2' were aligned in step-2.

The node alignment constructed thus, if successful, will lead to a bijective mapping from nodes of $\mathcal{A}_S$ to nodes of $\mathcal{A}_R$. Intuitively, we cannot repair student programs that have a different function/looping structure from the reference program. Note that the existing state-of-the-art approaches [9, 28] require identical number of edge transitions, in addition to an identical number of function/loop nodes, unlike in our approach. Referring to Figure 3, we obtain the nodes $\{q_1 q_1', q_2 q_2', q_3 q_3', q_4 q_4'\}$ of aligned automata $\mathcal{A}_{\mathcal{F}}$ after node-alignment on student automata $\mathcal{A}_S$ and reference automata $\mathcal{A}_R$.

Edge Alignment. Given two CFAs $\mathcal{A}_S$ and $\mathcal{A}_R$, and their corresponding node alignment $V :$ $V_S \leftrightarrow V_R$, we construct an aligned CFA $\mathcal{A}_{\mathcal{F}}$ by aligning the edges of $\mathcal{A}_S$ and $\mathcal{A}_R$. Suppose that $u_S \leftrightarrow u_R$ (i.e., node u_S in $\mathcal{A}_S$ is aligned with u_R in $\mathcal{A}_R$) and $v_S \leftrightarrow v_R$. For each edge of type $t \in \{break, return, normal\}$, we treat the following three cases differently.

(1) $\mathcal{A}_S$ has only one edge from u_S to v_S of type t, and $\mathcal{A}_R$ has only one edge from u_R to v_R of the same type t.
(2) Only $\mathcal{A}_R$ has an edge from u_R to v_R of type t, while $\mathcal{A}_S$ has no edge from u_S to v_S of type t.
(3) None of the above matches, and $\mathcal{A}_S$ (or $\mathcal{A}_R$) has multiple edges from u_S to v_S (or from u_R to v_R) of type t.

In the first case, we simply align the matching edges. For example, in Figure 3, $\mathcal{A}_R$ contains only one *normal* edge b between q_1 and q_2 and $\mathcal{A}_S$ contains only one *normal* edge b' between q_1' and q_2'. Hence, the aligned CFA $\mathcal{A}_{\mathcal{F}}$ has an edge $b; b'$ as shown in Figure 3(c). An example of the second case is shown with the two nodes, $q_1 q_1'$ and $q_4 q_4'$, of $\mathcal{A}_{\mathcal{F}}$. While $\mathcal{A}_R$ has one edge a between q_1 and q_4, $\mathcal{A}_S$ has no edge between q_1' and q_4'. In this case, we insert an edge $a; a'$ to $\mathcal{A}_{\mathcal{F}}$ where a' has an empty guarded action.

Lastly, in the third case, there exist several possible edge alignments, of the order of $\binom{M}{N} \times N!$, where M is the number of edges from $u_R \to v_R$ and N is the number of edges from $u_S \to v_S$. Fig 5 demonstrate this case through an example, resulting in two possible edge alignments. We choose the edge alignment which maximizes the number of verified-equivalent edges in the resultant aligned automaton $\mathcal{A}_{\mathcal{F}}$. The resultant aligned automaton $\mathcal{A}_{\mathcal{F}}$ of our running example is shown Fig. 3. The formal structure of aligned automaton appears in the following.

Definition 2 (Aligned automaton). *The automaton $\mathcal{A}_{\mathcal{F}}$ that results from aligning the automata $\mathcal{A}_S$ and $\mathcal{A}_R$ is a tuple of the form $\langle V, E, v^0, v^t, \Omega, \Psi, Pred \rangle$, where:*

- *$V : V_S \leftrightarrow V_R$, is a finite set of one-to-one bijective mappings between the nodes of the automata $\mathcal{A}_S$ and $\mathcal{A}_R$,*
- *$E \subseteq V \times V$, is a finite set of edges representing normal, break, and return transitions between the aligned nodes,*
- *$v^0 : v_S^0 \leftrightarrow v_R^0$, where v_S^0 and v_R^0 are the initial function entry nodes of the automata $\mathcal{A}_S$ and $\mathcal{A}_R$ respectively,*
- *$v^t : v_S^t \leftrightarrow v_R^t$, where v_S^t and v_R^t are the final function exit nodes of the automata $\mathcal{A}_S$ and $\mathcal{A}_R$ respectively*

- $\Omega : \{u \leftrightarrow v \mid \forall u \in V, \exists v \in V\}$, *for each function/loop entry node, maintains a mapping to the corresponding exit node,*
- Ψ *is the mapping from edge e to ψ_e for all edges e, where $\psi_e = \psi_s \cup \psi_r$, and ψ_s, ψ_r are the set of guarded actions at the aligned edges e_s and e_r of the automata $\mathcal{A}_S$ and $\mathcal{A}_R$ respectively.*
- $Pred : Var_S \leftrightarrow Var_R$, *denoting variable alignment, is a bijective mapping between variables of $\mathcal{A}_S$ and $\mathcal{A}_R$,*

Aligned automata augment the classical Control Flow Automata with variable alignment predicates for capturing mismatches (semantic variations) between the behavior of the student automaton and the corresponding reference automaton.

4.2 Inferring Variable Alignment Predicates

To infer alignment predicates of $\mathcal{A}_F$, we use a syntactic approach based on variable-usage patterns similar to that of SarfGen[28]. Our approach for computing the mapping between two sets of variables proceeds as follows.

For each edge e_i in $\mathcal{A}_F$ we collect the usage set for each variable x/x' in the reference/student program, namely the sets $usage(x, e_i)$ and $usage(x', e_i)$. In case the student automaton has fewer variables than reference automaton ($|Var_S| < |Var_R|$), then fresh variables are defined in Var_S. The goal is to find a variable alignment, a bijective mapping between Var_R and Var_S, which minimizes the average distance between $usage(x, e_i)$ and $usage(x', e_i)$ for each $i \in [1, n]$, where n is the number of edges in $\mathcal{A}_F$. This is done by constructing a distance matrix $\mathcal{M}_{e_i}$ for each edge e_i of size $|Var_R| \times |Var_S|$, where

$$\mathcal{M}_{e_i}(x, x') = \Delta \left(usage(x, e_i), usage(x', e_i) \right)$$

Using the matrices $\mathcal{M}_{e_1}, \ldots, \mathcal{M}_{e_n}$, we construct a global distance matrix $\mathcal{M}_g$ for the entire set of edges in $\mathcal{A}_F$, where

$$\mathcal{M}_g(x, x') = \sum_{i=1}^{n} \frac{\mathcal{M}_{e_i}(x, x')}{n}$$

We then choose to align the variable x in R to the variable x' in S, denoted as $x \leftrightarrow x'$, if the pair (x, x') has the minimum average distance among all possible variable y aligned with x', that is among all variable alignment pairs (y, x').

5 VERIFICATION AND REPAIR ALGORITHM

Once the aligned automaton $\mathcal{A}_F$ is constructed, we can initiate the repair process of the incorrect student program. Note that a repaired version of the incorrect student program produced by our algorithm is guaranteed to be semantically equivalent to the given structurally matched reference program. Our algorithm traverses the edges of the automaton $\mathcal{A}_F$ to perform edge verification which basically checks the semantic equivalence between an edge of the student automaton and its corresponding edge of the reference automaton.[3] In case the edge verification fails, we perform edge repair after which edge verification succeeds. While the existing approaches [9, 28] also similarly perform repair for aligned statements/expressions, the correctness of repair is not guaranteed unlike in our algorithm.

We combine the edge verification and repair into a single step by extending the well-known SyGuS (syntax-guided synthesis) approach [3] which can be defined as follows:

Definition 3 (SyGuS). *SyGuS consists of $\langle \varphi, T, S \rangle$ where φ represents a correctness specification expressed assuming background theory T and S represents the space of possible implementations (S*

---

[3]Our implementation performs a breadth-first search, while our algorithm is not restricted to a particular search strategy.

is typically defined through a grammar). The goal of SyGuS is to find out an implementation that satisfies φ.

While in principle SyGuS can be directly used to perform repair, we have an additional non-functional requirement not considered in SyGuS—that is, we want to preserve the student program as much as possible for pedagogical purposes. To accommodate this additional requirement, we introduce our approach, SyGuR (syntax-guided repair), formulated as follows:

Definition 4 (SyGuR). *Syntax-guided Repair or SyGuR consists of* $\langle \varphi, T, S, impl_o \rangle$ *where the first three components are identical with those of SyGuS, and* $impl_o \in S$ *represents the original implementation that should be repaired. The goal of SyGuR is to find out a repaired implementation* $impl_r \in S$ *that satisfies* φ. *In addition, differences between* $impl_o$ *and* $impl_r$ *should be minimal under a certain minimality criterion.*

We realize SyGuR in the context of automated feedback generation for student programs. In this section, we present the two algorithmic steps we perform to conduct SyGuR: edge verification and edge repair.

5.1 Edge Verification

In this section, we describe how we detect faulty expressions in the given incorrect student program. Recall that the edges of the automaton $\mathcal{A}_F$ are constructed by aligning the edges of the student automaton $\mathcal{A}_S$ with the edges of the reference automaton $\mathcal{A}_R$. Recall also that the edges of $\mathcal{A}_S$ can be faulty while the edges of $\mathcal{A}_R$ are considered always as non-faulty.

Each edge $e : u \xrightarrow{\psi_s; \psi_r} v$ of $\mathcal{A}_F$ between nodes u and v asserts the following property:

$$\{\phi_u\}\psi_s; \psi_r\{\phi_v\} \tag{1}$$

where ϕ_u and ϕ_v are the variable alignment predicates at the source node u and target node v of the edge e respectively, and ψ_r and ψ_s represent a list of guarded actions of the reference implementation and student implementation, respectively, expressed in a Single Static Assignment (SSA) form. For example, an original guarded action, if (x>1) x++, is converted into its SSA form, $((x_1 > 1) \implies x_2 = x_1 + 1) \land (\neg(x_1 > 1) \implies x_2 = x_1)$. Note that ψ_s and ψ_r do not interfere with each other, since the variables used in ψ_s and ψ_r are disjoint from each other. Also note that ψ_r and ψ_s do not contain a loop (that is, a single edge does not form a loop), and thus an infinite loop does not occur in the edge.

Edge verification succeeds if and only if property (1) holds. In SyGuR, property (1) expresses a correctness specification φ_e for edge e. To check property (1), we use an SMT solver by checking the satisfiability of the following formula:

$$\varphi_e = \phi_u \land \psi_s \land \psi_r \land \neg\phi_v \tag{2}$$

The satisfiability of φ_e indicates verification failure, or showing non-equivalence of two implementations along edge e. Conversely, the unsatisfiability of φ_e indicates verification success. Note that there always exists a model m that satisfies $\phi_u \land \psi_r \land \psi_s$ (this is because ϕ_u is not false, and the SSA forms of ψ_r and ψ_s are defined over disjoint variables), and verification succeeds only when for all such m, $\neg\phi_v$ does not hold. Intuitively, verification succeeds if and only if it is impossible for the post-condition ϕ_v to be false after executing ψ_r and ψ_s under the pre-condition ϕ_u.

As for background theories in the SMT solver, we use: LIA (linear integer arithmetic) for integer expressions, the theory of strings for modeling input/output stream, theory of uninterpreted functions to deal with function calls such as scanf/printf, and LRA (linear real arithmetic) to approximate floating-point expressions.

5.2 Edge repair

Once φ_e is found to be satisfiable for an edge e (which indicates that the edge verification fails), our goal is to repair edge e by modifying the student implementation encoded in ψ_s. Algorithm 1 shows our edge repair algorithm based on the CEGIS (counter-example-guided inductive synthesis) strategy [26]. In step 1, edge verification is attempted, and verification failure results in a counter-example ϕ_{ce} that witnesses verification failure. In the remaining part of the algorithm, we modify ψ_s to ψ_s' in a way that $\{\phi_{ce}\}\psi_s'; \psi_r\{\phi_v\}$ holds. If $\{\phi_u\}\psi_s'; \psi_r\{\phi_v\}$ also happens to hold, edge repair is deemed as completed. Otherwise, an SMT solver generates a new counter-example ϕ_{ce}', and our algorithm searches for ψ_s'' satisfying both $\{\phi_{ce}\}\psi_s''; \psi_r\{\phi_v\}$ and $\{\phi_{ce}'\}\psi_s''; \psi_r\{\phi_v\}$. This process is repeated until either edge repair is successfully done or it fails. Edge repair can fail either because the search space is exhausted or timeout occurs.

Let us first consider cases where ψ_s and ψ_r have the same number of guarded actions and all guarded actions have the same number of assignments. To ensure this requirement is met, we call function *Extend* (see line 20 of Algorithm 1) which will be described later. Under the current assumption that ψ_s and ψ_r have the same number of guarded actions, $Extend(\psi_s, \psi_r)$ returns ψ_s, and thus, its return value ψ_s^+ equals ψ_s.

To repair guarded actions ψ_s^+, we replace each of the conditional expressions and the update expressions (RHS expressions) with a unique placeholder variable h. This makes an effect of making holes in ψ_s^+, and filling in a hole for repair amounts to equating h with a repair expression. Function *RepairSketch* of the algorithm performs this task of making holes in ψ_s^+ and returns ψ_f defined as $\psi_s^+[e^{(i)} \mapsto h^{(i)}]$. In this definition, notation $\mapsto$ denotes a substitution operator defined over all expressions $e^{(i)}$ appearing in ψ_s^+ and their corresponding placeholder variables $h^{(i)}$. In the following, we use "hole" to refer to a placeholder variable.

In SyGuR (see Definition 4), the expression space of the holes is defined by implementation space S. Previous state-of-the-art works [9, 28] use the expressions of the reference program for repair (generated repairs are not verified in these works unlike in our approach), and we similarly define the implementation space of each hole as follows:

Definition 5 (Implementation space of a hole). *Let C_s (C_r) and U_s (U_r) be respectively the set of conditional and update expressions of ψ_s^+ (ψ_r). Recall that ψ_s^+ (ψ_r) represents guarded actions of the student (reference) program. When a conditional expression e_c is replaced by a hole h_c, the implementation space of h_c is defined as $C_s \mid C_r' \mid true \mid false$, where C_r' represents the set of conditional expressions appearing in ψ_r with all variables of C_r replaced with their aligned variables of the student program (see Section 4.2 for variable alignment). Similarly, given an assignment $x = h_u$ where h_u represents a hole for an update expression e_u, the implementation space of h_u is defined as $U_s \mid U_r' \mid x$, where U_r' represents the set of update expressions of ψ_r with all variables of U_r replaced with their aligned variables of the student program. The inclusion of an lhs variable x in the implementation space is to allow assignment deletion—replacing $x = e_u$ with $x = x$ simulates assignment deletion.*

The repair synthesis process for some faulty expression on the edge e_s relies on four factors: the discovered counter-examples, the set of suspicious expressions in ψ_s^+, the set of reference expressions in ψ_r, and the inferred alignment predicates. These factors collectively determine the set of expressions on the edge e_r that can be exploited to repair the buggy expressions on e_s.

Recall that given a list of counter-examples CE, we search for a repair ψ_s' that satisfies $\forall \phi_{ce} \in CE : \{\phi_{ce}\}\psi_s'; \psi_r\{\phi_v\}$. When searching for a repair, we preserve the expressions of the student program as much as possible for pedagogical reasons. We achieve this by conducting a search for a repair using a pMaxSMT (Partial MaxSMT) solver. Note that an input to a pMaxSMT solver consists of (1) hard constraints which must be satisfied and (2) soft constraints all of which may not be satisfied. Whenever a soft constraint C is not satisfied, cost is increased by the weight associated with C, and

---

Algorithm 1 Edge verification-repair

Input: Aligned $edge$
Output: Verified/Repaired $edge$
1: Let $\phi_u \equiv$ edge.sourceNode.invariants
2: Let $\phi_v \equiv$ edge.targetNode.invariants
3: Let $\psi_r \equiv$ edge.label.reference
4: Let $\psi_s \equiv$ edge.label.student
5: $CEs \leftarrow []$ // List of counter-examples
6: $candidates \leftarrow [\psi_s]$
7: **repeat**
8:      // Step 1: attempt for edge verification
9:      **for** each ψ_s^i in $candidates$ **do**
10:         Let $\varphi_{edge}^i \equiv \neg(\phi_u \wedge \psi_r \wedge \psi_s^i \implies \phi_v)$
11:         **if** $\not\models \varphi_{edge}^i$ **then** // UNSAT
12:            $edge.label.student \leftarrow \psi_s^i$ // Update edge
13:            **return** ✓   // Verifiably correct
14:         **else**
15:            $\phi_{ce}^i \models \varphi_{edge}^i$ // SAT
16:            $CEs \leftarrow CEs \cdot \phi_{ce}^i$
17:         **end if**
18:      **end for**
19:      // Step 2: make holes in ψ_s
20:      Let $\psi_s^+ \equiv Extend(\psi_s, \psi_r)$
21:      Let $\psi_f \equiv RepairSketch(\psi_s^+)$
22:      // Step 3: define implementation space
23:      $\varphi_{hard} \leftarrow []; \varphi_{soft} \leftarrow []$
24:      **for** each ϕ_{ce}^i in CEs **do**
25:         $\varphi_{hard} \leftarrow \varphi_{hard} \cdot (\phi_{ce}^i \wedge \psi_r \wedge \psi_f \wedge \phi_v)$
26:      **end for**
27:      **for** each $hole, expr, weight$ in $RepairSpace(\psi_f, \psi_r, \psi_s)$ **do**
28:         $\varphi_{soft} \leftarrow \varphi_{soft} \cdot (hole = expr, weight)$
29:      **end for**
30:      // Step 4: search for a repair
31:      **if** $\not\models (\varphi_{hard}, \varphi_{soft})$ **then** // UNSAT or UNKNOWN
32:         **return** ✗ // Repair Failure
33:      **else**
34:         // Update $candidates$ using a pMaxSMT solver
35:         // There can be multiple candidates
36:         $candidates \models (\varphi_{hard}, \varphi_{soft})$
37:      **end if**
38: **until** timeout

---

a pMaxSMT solver searches for a model that minimizes the overall cost. We pass the following formula to a pMaxSMT solver where hard constraints are underlined.

$$\forall \phi_{ce} \in CE : (\underline{\phi_{ce}} \wedge \underline{\psi_r} \wedge \underline{\psi_f} \wedge \underline{\phi_v} \wedge$$
$$\bigwedge_{(h^{(i)}, e^{(i)}, S[\![h^{(i)}]\!]) \in holes(\psi_f)} (h^{(i)} = e^{(i)} \wedge h^{(i)} \in S[\![h^{(i)}]\!] \backslash \{e^{(i)}\})) \tag{3}$$

where function $holes(\psi_f)$ returns a set of $(h^{(i)}, e^{(i)}, S[\![h^{(i)}]\!])$ in which $h^{(i)}$ represents the placeholder variable appearing in ψ_f (recall that ψ_f is prepared by making holes in ψ_s), $e^{(i)}$ denotes the original expression of h extracted from student program, and $S[\![h^{(i)}]\!]$ represents the implementation space of $h^{(i)}$. Our soft constraints encode the property that each of the original expressions can be either preserved or replaced with an alternative expression in the implementation space. To preserve as many original expressions as possible, we assign a higher weight to $h^{(i)} = e^{(i)}$ than $h^{(i)} \in S[\![h^{(i)}]\!] \setminus \{e^{(i)}\}$.

The Extend function. Previously, we consider only the cases where ψ_s and ψ_r have the same number of guarded actions and all guarded actions have the same number of assignments. To ensure this requirement, we invoke the *Extend* function which performs the following. First, if ψ_s has a smaller number of guarded actions than ψ_r, then ψ_s^+ (the return value of *Extend*) should contain additional guarded actions, each of which uses the following template: $[False] \implies x = x$, where x is constrained to be the variables of the student program. Notice that these additional guards are initially deactivated to preserve the original semantics of the student program, but they can be activated whenever necessary during repair, since $False$ is replaced with a hole by *RepairSktech*. After this step, *Extend* finds the guarded action of ψ_r that has the maximum number of assignments. Given this maximum number M, we check whether all guarded actions of ψ_s (including additional guarded actions with the $False$ guard) also have M assignments. Any guarded action that has a smaller number of assignments than M is appended with additional assignments, $x = x$ where x is constrained to be the variables of the student program. This process makes sure that for each guarded action, the student program can have as many assignments as the reference program.

5.3 Properties preserved by Verifix

Once all the edges of the aligned automaton $\mathcal{A}_F$ are repaired and verified, it is straightforward to produce a repaired student automaton $\mathcal{A}'_S$ by copying repaired expressions from the automaton $\mathcal{A}_F$ to the automaton $\mathcal{A}_S$. In this section, we discuss several interesting properties of our repair algorithm, namely soundness, completeness, and minimality of generated repairs. Some of their proofs are available in the supplementary material.

Theorem 1 (Soundness). *For all program inputs, $\mathcal{A}'_S$ and $\mathcal{A}_R$ return the same program output.*

Our edge repair algorithm (Algorithm 1) always returns a repaired edge as long as the underlying MaxSMT/pMaxSMT solver used in the algorithm is complete (that is, UNKNOWN is not returned). This can be stated as follows, using the concept of relative completeness [7]:

Theorem 2 (Relative completeness of edge repair). *The completeness of Algorithm 1 is relative to the completeness of the MaxSMT/pMaxSMT solver.*

Meanwhile, the overall repair algorithm of Verifix is not complete. If $\mathcal{A}_F$ is failed to be constructed, the repair process cannot be started. Theorem 3 identifies the conditions under which Verifix succeeds to generate a repair. In Theorem 3, we use the following definition of alignment consistency:

Definition 6 (Alignment Consistency). *For each edge e of $\mathcal{A}_F$ $\{\phi_u\}\psi_s; \psi_r\{\phi_v\}$, modify ψ_s into the ψ'_s as follows: $\psi'_s \equiv \psi_r[x_r^{(i)} \mapsto x_s^{(i)}]$ where $x_r^{(i)}$ denotes all reference-program variables appearing in ψ_r and $x_s^{(i)}$ denotes student-program variables aligned with $x_r^{(i)}$. Repeat this for all edges of $\mathcal{A}_F$. Then, we say that $\mathcal{A}_F$ is alignment consistent when $\{\phi_u\}\psi'_s; \psi_r\{\phi_v\}$ for all modified edges.*

$\mathcal{A}_F$ is alignment consistent only when the variable alignment predicates are such that a given student program can be verifiably repaired by edge-to-edge copy of the reference program (patch minimality is not considered).

Theorem 3 (Relative completeness). *Our repair algorithm succeeds to generate a repair, under the following assumptions:*

(1) *$\mathcal{A}_F$ is constructed,*
(2) *$\mathcal{A}_F$ is alignment-consistent, and*
(3) *The MaxSMT/pMaxSMT solver used for repair/verification is complete.*

Use of MaxSMT guarantees the minimality of edge repair.

Theorem 4 (Minimality of edge repair). *Suppose that our algorithm repairs edge $e : u \longrightarrow v$ of $\mathcal{A}_F$ by changing $F \subseteq C_s \cup U_s$ (C_s and U_s are defined in Definition 5). There does not exist F' s.t. $|F'| < |F|$ and the pre-/post-conditions of e are satisfied by replacing the expressions of F' with the expressions in $C_r \cup U_r$.*

In the following theorem we state the conditions under which the global minimality of our generated repair is guaranteed.

Theorem 5 (Relative minimality). *A repaired program generated by our algorithm is minimal under the following assumptions:*

(1) *Node alignment made in $\mathcal{A}_F$ is optimal in the sense that there is no alternative alignment that can lead to a smaller repair. Note that edge alignment of $\mathcal{A}_F$ is optimal (see Section 4.1).*
(2) *The variable alignment predicates of $\mathcal{A}_F$ are optimal in the sense that there is no alternative variable alignment that can lead to a smaller repair. As shown in Section 4.2, we use a heuristics-based approach for the sake of efficiency.*

Despite these limitations, our experimental results show that Verifix tends to find smaller repairs than Clara. Note that the existing approaches designed to generate minimal repairs [9, 28] also do not consider node/edge alignment in the calculation of the minimality of a repair. Instead, a minimal repair is searched for only after node/edge alignment is made. In fact, unlike those existing approaches that do not consider alignment at all, we consider edge alignment by enumerating over all possible edge alignments between aligned nodes.

6 EXPERIMENTAL SETUP

Evaluation of a programming assignment feedback tool requires a data-set of incorrect student assignments. For our data-set, we chose a publicly released dataset curated by ITSP [4] [29] for evaluating feasibility of APR techniques on introductory programming assignments. This benchmark consists of incorrect programming assignment submissions by 400+ first year undergraduate students crediting a *CS-1: Introduction to C Programming* course at a large public university. Note that the datasets used by Clara [9], Sarfgen [28], and CoderAssist [13] are not publicly released, while the dataset released by Refactory [12] targets Python programming assignments.

We take students' incorrect attempts from four basic weekly programming labs in ITSP benchmark, where each lab consists of several programming assignments that cover different programming topics. For example, the lab in week 3 (Lab-3 in Table 3) consists of four programming assignments which teach students about floating-point expressions, printf, and scanf. Table 3 lists the four programming labs partitioned by different programming topics. Students had, on average, a time limit of one hour duration for completing each individual assignment. Advanced labs dealing with topics such as multi-dimensional arrays (matrices) and pointers were excluded from our benchmark due to lack of Verification Condition Generation (VCGen) implementation support of converting code to SMT logical formulae.

---

[4]https://github.com/jyi/ITSP#dataset-student-programs

Lab-ID	Topics	# Assignments	# Programs	SM (%)		Repair (%)		Time (avg)	
				Clara	Verifix	Clara	Verifix	Clara	Verifix
Lab-3	Floating point, printf, scanf	4	63	100%	100%	54.0%	92.1%	2.1	58
Lab-4	Conditionals, Simple Loops	8	117	92.3%	92.6%	71.8%	82.9%	21.0	41.5
Lab-5	Nested Loops, Procedures	8	82	24.4%	64.6%	22.0%	45.1%	11.1	9.7
Lab-6	Integer Arrays	8	79	15.2%	31.6%	12.7%	21.5%	17.9	9.4
Total	-	28	341	59.5%	71.2%	42.8%	58.4%	15.2	39

Table 3. Repair accuracy (Repair) and Structural Match (SM) rate of our tool Verifix and Clara [9]. Time column represents the average runtime for all successfully repaired programs. Number of assignments in each lab is shown as *#Assignments*. Number of incorrect student submissions in each lab is shown as *#Programs*.

Clara		Verifix	
Reason	Perc (%)	Reason	Perc (%)
Structural Mismatch (SM)	71%	Structural Mismatch (SM)	62%
Timeout	27%	Timeout	35%
Unsupported feature	2%	SMT issue	2%
		Unsupported feature	1%

Table 4. Reasons for Clara and Verifix repair failure in Table 3

Finally, we use 341 compilable incorrect students' submissions from 28 various unique programming assignments as our subject. In addition to the incorrect student submissions, each programming assignment in the ITSP benchmark contains a single reference implementation and a set of test cases designed by the course instructor. Both Verifix and baseline Clara [9] have access to the reference implementation and test cases to repair the incorrect student programs.

Baseline comparison. We compare our tool Verifix's performance against the publicly released state-of-art repair tool Clara [5] [9] on the common dataset of 341 incorrect student assignments. A timeout of 5 minutes per incorrect student program was set for both Verifix and Clara to generate the repair. We do not directly compare our results against CoderAssist [13] tool since it requires additional manual effort in the form of instructor validated submissions, while Refactory [12] implementation targets Python programming assignments. The SarfGen [28] tool has not been publicly released and hence cannot be directly compared against. We instead address these comparisons in our related work Section 9. We will publicly release both our Verifix tool (open-source) and subjects to aid further research.

In principle, Verifix is applicable to any programming language with Verification Condition Generation (VCGen) support; our implementation targets *C* programming assignments. Our experiments were carried out on a machine with Intel® Xeon® E5-2660 v4 @ 2.00 GHz processor and 64 GB of RAM.

7 EVALUATION

We address the following research questions in our work.

(1) RQ1: What is the repair accuracy and reasons for failure of Verifix on the diverse programming assignments dataset?
(2) RQ2: Does Verifix generate small sized meaningful repair?
(3) RQ3: What is the effect of test-suite quality on repair (overfitting)

---

[5]https://github.com/iradicek/clara

7.1 RQ1: Repair accuracy and failure analysis

Table 3 compares the repair accuracy of our tool Verifix against the state-of-art tool Clara [9] on our dataset of student submissions. Given a single reference implementation per assignment, Verifix achieves an overall repair accuracy of 58.4% on the 348 incorrect programs across 28 unique assignments. In comparison, the baseline tool Clara achieves a lower overall accuracy of 42.5% on the same assignments, a difference of more than 15%. Note that *the repairs generated by Verifix are verifiably equivalent to the reference implementation*, in addition to passing all the instructor provided test cases. This means that in close to 60% of the student assignments, we can use Verifix to generate a verifiably correct feedback with confidence, failing which traditional manual feedback generation by human tutors can be used.

This substantial improvement in repair accuracy of Verifix over Clara is primarily due to the simpler *Structural Match* (SM) requirement of Verifix than that of Clara. Verifix requires the nodes in Control-Flow Automata (CFA) structures of both reference and student program to match, in order to generate a repair (refer Section 4.1). While Clara imposes an additional requirement for the edges of the reference and student CFA to match. That is, in addition to the function and loop structure matching, Clara additionally imposes return/break/continue control-flow matching between the student and reference program.

This difference in repair accuracy due to structural match failure can be observed from our Table 3. The reference and student programs for the initial Lab-3 assignments (covering floating-point, printf and scanf topic) are simple; single-procedural without loops. Hence the control flow structure of all the 63 incorrect student programs matches with the CFG of instructor designed reference program. As a result, Clara's CFG based SM algorithm performs similar to Verifix's CFA based SM algorithm, with both achieving 100% structural match rate.

For comparison, consider the *Lab-5* assignment, involving nested loop and multi-procedures. The difference between Verifix's Structural Match (SM) rate of 64.6% for Lab-5 and Clara's Structural Match (SM) rate of 24.4% is ~50%. Due to Verifix's structural matching involving CFA based abstraction, Verifix is able to successfully align 64.6% of incorrect student programs with reference program, while Clara's stricter CFG based structural matching can align only 24.4% of Lab-5 incorrect student programs. This allows Verifix to achieve more than twice the repair accuracy of 45.1% for Lab-5 as compared to Clara's accuracy of 22.0%.

Verifix successfully generates a repair within 39 seconds on average. A majority of this time is spent in the invocation of Z3 SMT and pMaxSMT solver for verification and repair purposes, respectively. In comparison, the average running time of Clara is 15.2 seconds. To generate a verifiably correct repair, Verifix uses more time than Clara, suggesting that there is a trade-off to make between correctness guarantee and repair time. We note that according to an earlier user study [29], students spend 100s on average to resolve semantic errors, and the average running time of Verifix is significantly less than 100s, indicating its possible use in live feedback.

Table 4 lists the reasons for repair generation failure by Verifix, along with the weightage of each reason. *Structural Mismatch* (SM) is the primary reason for repair failure, accounting for 62% of all the failure cases. A structural mismatch occurs when the loop nesting or number of function definitions do not match between the incorrect program and the single reference solution. Since the current version of Verifix does not have the capability to insert/delete loops or functions, we are unable to verify and repair such incorrect student programs.

The second biggest failure reason is due to *Timeout*, when Verifix exhausts our 5 minutes runtime limit in verifying/repairing the incorrect student submission. This case can occur when the repair space involving reference/incorrect expressions is large. Note that while our algorithm is sound and
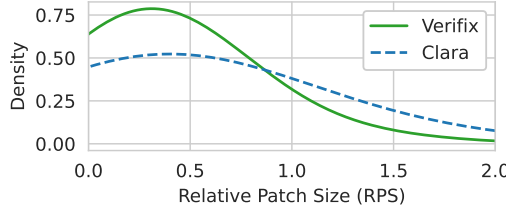
Fig. 6. Kernel Density Estimate (KDE) plot of Relative Patch Size (RPS) by Verifix and Clara on 104 common successful repairs.

```
 1   void main(){
 2     int n1, n2, i;
 3     scanf("%d %d", &n1, &n2);
 4     if(n2 <= 2)          // Repair #1: Delete spurious print
 5       printf("%d ", n2); //              Verifix ✓, Clara ✗
 6     for(i=n1; i<=n2; i++){
 7       if(check_prime(i)==0) // Repair #2: Delete ==0
 8         printf("%d ", i);    //   Verifix ✓, Clara ✓
 9     }
10   }
```

Fig. 7. Example from a Lab-5 *Prime Number* assignment. The main function contains two errors, both of which are fixed by Verifix, while Clara's repair overfits given test-suite by ignoring first error.

complete, the z3 solver could struggle with finding a SAT or UNSAT result within our 5 minutes time limit.

In ~2% of the failure cases, Verifix is unable to generate a repair due to SMT issues. These SMT issues mainly occur due to the incomplete theories of integers and reals for non-linear expressions in SMT solver. Leading to an UNKNOWN query results by the SMT solver. The final issue accounting for ~1% of all failure cases occurs due to lack of support for few programming feature that occur in the dataset. For example, we currently do not support the use of *GOTO* control-flow transitions since they are considered bad programming practice, especially in a pedagogy setting. From Table 4 we note that the failure reasons of Clara follow a similar pattern as Verifix. With Structural Mismatch (SM) and timeout accounting for majority of the failures. Unsupported programs in the dataset accounts for a distant third, at 2% of all failure cases.

7.2 RQ2: Minimal repair

In a pedagogy setting, large changes to the incorrect solution can confuse the student. Hence, apart from correctness measures, we also compare the patch (or repair) size of Verifix and our baseline state-of-art tool Clara [9]. Since the size of the student programs vary significantly, we normalize patch size with the size of original incorrect program to obtain Relative Patch Size (RPS), given by: $RPS = Dist(AST_s, AST_f)/Size(AST_s)$. Where, AST_s and AST_f represents the Abstract Syntax Tree (AST) of incorrect student program and fixed/repaired program generated by tool, the *Dist* function computes a *tree-edit-distance* between these ASTs, and the *Size* function computes the #nodes in the AST.
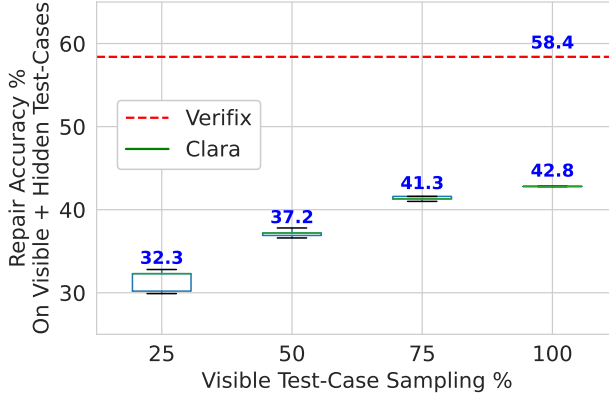
Fig. 8. Repair accuracy of Clara and Verifix on various test case samplings.

In our benchmark of 341 incorrect programs, Verifix can successfully repair 199 student programs, Clara can successfully repair 146 programs, while Verifix and Clara both can successfully repair 104 common programs. Out of these 104 commonly repaired programs, Verifix generates a patch with smaller RPS in 56 of the cases, Clara generates a patch with smaller RPS in 33 of the cases, and both tools generate a patch of the exact same relative patch size in 15 cases. Note that a smaller repair does not necessarily imply better quality repair since these repairs can overfit the test cases, as we demonstrate in our RQ3 through an example (Section 7.3).

Figure 6 plots the Kernel Density Estimate (KDE) of Relative Patch Size (RPS) for these 104 common programs that both Verifix and Clara can successfully repair, in order to visualize the RPS distribution for these large number of data points. KDE is an estimated Probability Density Function (PDF) of a random variable, often used as a continuous smooth curve replacement for a discrete histogram. From the Figure 6 plot we observe that the density of patch-sizes (y-axis) produced by Verifix is greater than that of Clara when $RPS < 0.8$ (x-axis). On the other hand, the density of patch-sizes generated by Clara is greater than that of Verifix when $RPS \geq 0.8$. That is, a large proportion of repairs generated by Verifix have a small relative patch-size, since the density concentration of repairs is towards lower RPS (x-axis). In comparison, a significantly larger proportion of Clara's repairs have $RPS \geq 0.8$, as compared to Verifix.

7.3 RQ3: Overfitting

Majority of the programming assignment repair tools [9, 12, 28, 29] generate repairs that satisfy a given test suite (incomplete specification). Unlike these existing test based approaches, Verifix uses SMT based verification to find a minimal repaired program that behaves similar to a given reference implementation on all possible inputs (complete specification). Figure 7 demonstrates an example from a Lab-5 *Prime Number* assignment, where Clara's [9] repair overfits the test cases. With the help of a reference implementation, Verifix is able to detect a new counter-example where the student program deviates from correct behavior, when input stream is "1 2" ($n1 = 1$, $n2 = 2$). Given this new unseen test case, the repair suggested by Clara results in an incorrect output "2 2 ", while the repair suggested by Verifix results in the correct behaviour producing output "2 ".

In order to measure the degree of overfitting repairs generated by each tool, we compare the affect of test case quality on repair accuracy. This is done by running Clara and Verifix on our common benchmark of 341 incorrect programs under four different settings, where a percentage of

test cases were hidden from tool during repair generation. For each of the 28 unique assignments, with 6 number of instructor designed test cases on average, we randomly sampled $X\%$ as "visible" test cases. Once the repair was successfully generated by a tool on the limited visible test case sample, we re-evaluated the repaired program on all test cases, including hidden ones. We carried out this experiment under four different settings, with a random sampling rate of 25%, 50%, 75%, and 100% of the available test cases. This entire experiment was repeated 5 times, where we randomly sampled test cases each time, and we report on the distribution of repair accuracy achieved by each tool.

Figure 8 displays the result of our overfitting experiment, with the X-axis representing the visible test case sampling %, and Y-axis representing the repair accuracy % obtained by APR tool on the entire test-suite (visible and hidden test cases). Each box plot displays the distribution of repair accuracy per test case sampling, by showing the minimum, maximum, upper-quartile, lower-quartile and median values. The median value of each box-plot is shown as text above the box-plot.

From the Figure 8 we observe that Verifix's repair accuracy is constant. That is, Verifix's repair does not change based on the percentage of visible test cases provided, since it does not use the available test cases for repair generation or evaluation/verification. On the other hand, Clara's repair accuracy varies from a median value of 42.8% (when all test cases are made visible) to a median value of 32.3% (when only 25% of test cases are made available to Clara). In other words, Clara overfits on $42.8 - 32.3 = 10.5\%$ of our benchmark of 341 incorrect programs, when 25% of test cases are randomly chosen. Similarly, overfitting of $42.8 - 41.3 = 1.5\%$ is observed when visible test case sampling rate is 75%, or 5 visible test cases ($\lceil 75\% \times 6 \rceil = 5$) on average. In other words, when even a single test case on average is hidden from Clara, its generated repair can overfit the test cases.

Moreover, the choice of test case sampling has a large effect on Clara's accuracy, as evident from the variation in box-plot distribution. In the case of 25% visible test case sampling, Clara's repair accuracy ranges from a minimum value of 29.9 to maximum of 32.8; depending on which two test cases ($\lceil 25\% \times 6 \rceil = 2$) were made available.

Hence, APR tools such as Clara [9] which rely on availability of good quality test cases for their repair generation and evaluation can suffer from overfitting. Even when the instructor misses out on a single important test case coverage during assignment design. Thereby generating incomplete feedback to students struggling with their incorrect programs. Verifix on the other hand does not suffer from overfitting limitation, due to its sole reliance on reference implementation for repair generation and evaluation/verification.

8 THREATS TO VALIDITY

Our aligned automata setup phase consists of a syntactic procedure to obtain a unique edge and variable alignment between the reference and student automata. Producing an incorrect alignment does not affect our soundness or relative completeness guarantees, but can instead increase our patch-size. This however occurs rarely in practice, as demonstrated by our RQ2 (Section 7.2).

The arithmetic theory of SMT solvers is incomplete for non-linear expressions, which can affect our relative completeness. However, this issue affects $< 2\%$ of our dataset of incorrect student programs in practice, as demonstrated by our results in Table 4.

Evaluating repair tools using multiple correct student submissions, instead of restricting to a single instructor reference solution, could help improve the repair accuracy. We mitigate this risk by noting that such an evaluation has been undertaken in the literature earlier [12, 28], and would benefit both Verifix and our baseline tool Clara in terms of additional Structural Match (SM) rate.

Furthermore, we cannot always assume the availability of a large number of reference solutions, in general.

9 RELATED WORK

9.1 General Purpose Program Repair

Automated Program Repair (APR) [8, 19] is an enabling technology which allows for the automated fixing of observable program errors thereby relieving the burden of the programmers. General purpose APR techniques such as GenProg [15], SemFix [21], Prophet [16] and Angelix [18], require an incomplete correctness specification typically in the form of a test-suite. These techniques achieve low accuracy on student programs that suffer from multiple mistakes, since they can scale to large programs but not necessarily to large repair search spaces [29]. As student programs are substantially incorrect, the search space of repairs is typically large.

ITSP [29] reports positive results on deploying general APR tools for grading purpose by expert programmers, and negative result when used by novice programmers for feedback. Their low repair accuracy and reliance on test-cases (overfitting) can be seen as a motivator for our work.

S3 [14] synthesizes a program using a generic grammar and user-defined test-cases. Semgraft [17] uses simultaneous symbolic execution on a buggy program and a reference program to find a repair, which makes the two program equivalent for a group of test inputs; this class of test inputs is captured by a user-provided input condition. Our work shifts away from test inputs and instead constructs verification guided repair. Furthermore, for our application domain of pedagogy, we seek to build minimal repairs by retaining as much of the buggy program as possible.

Churchill et al [6] use trace based alignment to produce a unique aligned automaton, to demonstrate equivalence between two correct programs. Verifix constructs aligned automata using a syntax-based alignment, with the goal of repairing a given incorrect program to become equivalent to a given correct program.

9.2 Repair of Programming Assignments

Autograder [25] is one of the early approaches in this domain. In Autograder, the correctness of generated patches are verified only in bounded domains (e.g., the size of a list in the program is bounded to a constant number), and thus the verification result is generally unsound. Autograder also requires instructors to manually provide an error model that specifies common correction patterns of student mistakes, which is not needed in Verifix.

Clara [9] too performs bounded unsound verification. Clara checks whether each concrete execution trace of the student program matches that of the reference program, and performs a repair on mismatch. Since a concrete execution trace is obtained from test execution, the correctness of a generated patch cannot be guaranteed. We have provided a detailed experimental comparison with Clara. Clara assumes the availability of multiple correct student submissions with matching control-flow to the incorrect submissions, limiting their applicability unlike Verifix. Sarfgen [28] generates patches based on a lightweight syntax-based approach, and assumes the availability of previous student submissions. Both Clara and SarfGen require strict Control-Flow Graph (CFG) similarity between the student and reference program. In comparison, Verifix requires matching function and loop structure between student and reference program. Unlike Clara and SarfGen, Verifix can recover from differences in return/break/continue edge transitions due to its usage of Control-Flow Automata (CFA) based abstraction.

Refactory [12] handles the CFG differences by mutating the CFG of the student program to that of the reference program by using a limited set of semantics-preserving refactoring rules, designed manually. For example, refactoring a while-loop by replacing it with a for-loop structure. Note that

Verifix, unlike Refactory, keeps the original CFG of the student program as much as possible, as shown in Fig 1. Our goal is to produce small feedback of high quality. We cannot experimentally compare with Refactory since its implementation targets Python programming assignments.

CoderAssist [13], to the best of our knowledge, is the only APR approach that can generate verified feedback. CoderAssist clusters submissions based on their solution strategy followed by manual identification (or creation) of correct reference solutions in each cluster. After the clustering phase, CoderAssist undertakes repair at the contract granularity rather than expression granularity — that is, while CoderAssist can suggest which pre-/post-condition should be met for a code block, CoderAssist does not have the capacity to suggest a concrete expression-level patch. CoderAssist repair algorithm and evaluation results focus on dynamic programming assignments. In contrast, Verifix is designed and evaluated as a general-purpose APR.

There have been several attempts to use neural networks [1, 4, 5, 10, 22, 27] for program repair. These approaches typically target syntactic/compilation errors, and the repair rate for semantic/logical errors is low [22]. Such machine learning based techniques do not offer any relative completeness guarantees, and their repair soundness is evaluated against an incomplete specification (such as tests).

There has been prior work on live deployment of APR tools for repairing student programs [2, 29]. The work of ITSP [29] shows negative results on providing semantic repair feedback for student programs. At the same time, the work of Tracer [2] demonstrates positive results for repair based feedback, albeit on simpler (compilation) errors. In this work, we present an approach for repairing complex logical errors in student programs. Our tool Verifix can generate verified feedback for 58.4% of incorrect student submissions for 28 diverse assignments, collected from an actual CS-1 course offering. The human acceptability of our verified feedback can be further investigated via future user-studies.

10 CONCLUSION

In this paper, we have presented an approach and tool Verifix, for providing verified repair as feedback to students undertaking introductory programming assignments. The verified repair is generated via relational analysis of the student program and a reference program. Verifix is able to achieve better repair accuracy than existing approaches on our common benchmark. The repairs produced by Verifix are of better quality than state-of-art techniques like Clara [9], since they are often smaller in size, while being verifiably equivalent to the instructor provided reference implementation.

REFERENCES

[1] Umair Z Ahmed, Pawan Kumar, Amey Karkare, Purushottam Kar, and Sumit Gulwani. 2018. Compilation error repair: for the student programs, from the student programs. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*. 78–87.

[2] Umair Z. Ahmed, Nisheeth Srivastava, Renuka Sindhgatta, and Amey Karkare. 2020. Characterizing the pedagogical benefits of adaptive feedback for compilation errors by novice programmers. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering Education and Training (ICSE)*. 139–150.

[3] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-Guided Synthesis. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. 1–17.

[4] Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. 2018. Neuro-symbolic program corrector for introductory programming assignments. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 60–70.

[5] Darshak Chhatbar, Umair Z. Ahmed, and Purushottam Kar. 2020. MACER: A Modular Framework for Accelerated Compilation Error Repair. In *International Conference on Artificial Intelligence in Education*. Springer, 106–117.

[6] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*

(PLDI). 1027–1040.

[7] Stephen A Cook. 1978. Soundness and completeness of an axiom system for program verification. *SIAM J. Comput.* 7, 1 (1978), 70–90.

[8] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* 62 (2019). Issue 12.

[9] Sumit Gulwani, Ivan Radicek, and Florian Zuleger. 2018. Automated clustering and program repair for introductory programming assignments. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* 465–480.

[10] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. Deepfix: Fixing common c language errors by deep learning. In *Conference on Artificial Intelligence (AAAI).* 1345–1351.

[11] T.A. Henzinger, R Jhala, R Majumdar, and G Sutre. 2002. Lazy Abstraction. In *ACM SIGPLAN SIGACT Symposium on Principles of Programming Languages (POPL).*

[12] Yang Hu, Umair Z Ahmed, Sergey Mechtaev, Ben Leong, and Abhik Roychoudhury. 2019. Re-factoring based program repair applied to programming assignments. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 388–398.

[13] Shalini Kaleeswaran, Anirudh Santhiar, Aditya Kanade, and Sumit Gulwani. 2016. Semi-supervised verified feedback generation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE).* 739–750.

[14] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE).* 593–604.

[15] C. Le Goues, ThanhVu Nguyen, S. Forrest, and W. Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (Jan 2012), 54–72.

[16] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL).* 298–312.

[17] Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2018. Semantic Program Repair Using a Reference Implementation. In *Proceedings of the 40th International Conference on Software Engineering (ICSE).*

[18] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE).* 691–701.

[19] Martin Monperrus. 2018. Automatic software repair: a bibliography. *Comput. Surveys* 51 (2018). Issue 1.

[20] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *TACAS.* 337–340.

[21] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: program repair via semantic analysis. In *Proceedings of the 35th International Conference on Software Engineering (ICSE).* 772–781.

[22] Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. 2016. sk_p: a neural program corrector for MOOCs. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity.* 39–40.

[23] Z Qi, F Long, S Achour, and M Rinard. 2015. An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA).*

[24] Tyler J Ryan, Gene M Alarcon, Charles Walter, Rose Gamble, Sarah A Jessup, August Capiola, and Marc D Pfahler. 2019. Trust in automated software repair. In *International Conference on Human-Computer Interaction.* Springer, 452–470.

[25] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation (PLDI).* 15–26.

[26] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems.* 404–415.

[27] Ke Wang, Rishabh Singh, and Zhendong Su. 2017. Dynamic neural program embedding for program repair. *arXiv preprint arXiv:1711.07163* (2017).

[28] Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Search, align, and repair: data-driven feedback generation for introductory programming exercises. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).* 481–495.

[29] Jooyong Yi, Umair Z. Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. 2017. A feasibility study of using automated program repair for introductory programming assignments. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE).* 740–751.