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, South 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. A student-submitted program is aligned and composed with a reference solution in terms of control flow, and the variables of the two programs are automatically aligned via predicates describing the relationship between the variables. When verification attempt for the obtained aligned program fails, we turn a verification problem into a MaxSMT problem whose solution leads to a minimal repair. We have conducted experiments on student assignments curated from a widely deployed intelligent tutoring system. Our results show that generating verified repair without sacrificing the overall repair rate is possible. In fact, our implementation, Verifix, is shown to outperform Clara, a state-of-the-art tool, in terms of repair rate. This shows the promise of using verified repair to generate high confidence feedback in programming pedagogy settings.

CCS Concepts: • **Applied computing → Computer-assisted instruction**; • **Software and its engineering → Formal software verification**; **Software testing and debugging**.

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 (APR) techniques for providing automated feedback to student assignments [13, 16, 17, 30, 33, 34].

Existing approaches and their limitations. Table 1 provides a summary of state-of-the-art APR works for introductory programming assignment, and compares them with our approach Verifix. The repair rate of the state-of-the-art techniques [13, 16, 33] is astonishingly high, around 90%. However, different from general test-based APR technique, these works make certain assumptions such as the presence of multiple reference programs and a high quality tests.

Many student assignment feedback generation approaches [13, 16, 33] 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 test-based APR tools [11, 28, 35]. Prior studies have shown that trivial repairs such as functionality deletion alone can achieve ~50% repair success rate on buggy student programs given a weak oracle [6]. Generating complex incorrect feedback that merely passes all tests can potentially confuse novice students more than expert programmers.

*Joint first authors.
†Corresponding author.

Authors' addresses: Umair Z. Ahmed, umair@nus.edu.sg, National University of Singapore, Singapore; Zhiyu Fan, National University of Singapore, Singapore, zhiyufan@comp.nus.edu.sg; Jooyong Yi, Ulsan National Institute of Science and Technology, South Korea, jooyong@unist.ac.kr; Omar I. Al-Bataineh, National University of Singapore, Singapore, omerdep@yahoo.com; Abhik Roychoudhury, National University of Singapore, Singapore, abhik@comp.nus.edu.sg.

Tool	Completely Automated	Beyond Identical Reference CFG	Verified Repair	Target Language	Tool Availability	Dataset Availability
Clara [13]	✓	✗	✗	C, Python	✓	✗
SarfGen [33]	✓	✗	✗	C#	✗	✗
ITSP [34]	✓	✓	✗	C	✓	✓
Refactory [16]	✓	✓	✗	Python	✓	✓
CoderAssist [17]	✗	✓	✓	DP for C	✓	✗
Verifix	✓	✓	✓	C	✓	✓

Table 1. Programming assignment repair tools comparison. Most existing APR tools are completely automated and rely on test case evaluation (generate unverified repair).

Indeed, a prior study [34] shows that novice students when provided with incorrect/partial repair feedback that merely passes more tests, have been shown to struggle more, as compared to expert programmers given the same feedback. Hence, we suggest that the feedback given to novice students needs rigorous quality assurance, whenever possible.

In a related vein, some approaches, in particular recent ones [13, 33], 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] Using multiple reference programs can also diversify the solution space, and thereby a feedback can be made more customized to a student solution [12]. However, the problem is that the existing approaches collect multiple reference programs manually or based on testing (student submissions that pass all tests are considered correct), without formally verifying their correctness. Automatic equivalence checking remains challenging despite recent advances [7].

Insight. Many of the aforementioned problems of the existing APR techniques can be addressed with a verified repair. We assume the presence of at least one reference solution, which is always available in educational settings and can be given by an instructor. This setting is simpler than most existing approaches [13, 16, 33] requiring both multiple reference solutions and a test-suite. 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, 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. 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 [29]. We show that verified repair is feasible and achievable in a reasonable time scale (on average 29.5 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.[2] To the best of our knowledge, ours is the first work to espouse verified repair for general purpose

[1] SarfGen [33] and Clara [13] require that the control-flow structure of student and reference programs should be exactly the same. Clara also demands aligned variables to be evaluated into the same sequence of values at runtime.
[2] According to an earlier user study [34], students spend about 100s on average to resolve semantic errors.

```c
int check_prime(int n)
{
  if (n == 1)
    return 0;
  int j;
  for(j=2; j<n; j++)
  {
    if (n%j == 0)
      return 0;
  }
  return 1;
}
```

(a) A reference program

```c
int check_prime(int n)
{


  int i;
  for(i=1;i<=n-1;i++)
  {
    if (n%i == 0)
     break;
  }
  return 1;
}
```

(b) An incorrect student program

```c
int check_prime(int n)
{
  if (n == 1)
    return 0;
  int i;
  for(i=2;i<=n-1;i++)
  {
    if (n%i == 0)
      return 0;
  }
  return 1;
}
```

(c) The Verifix-generated repair

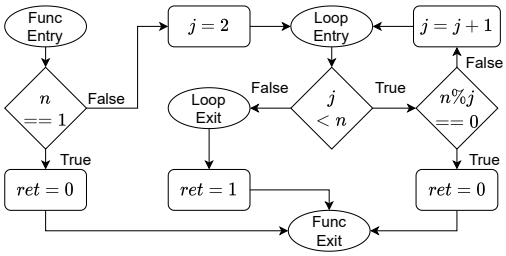
Fig. 1. Motivating example for the *Prime Number* programming assignment. Existing tools such as Clara [13] and Sarfgen [33] cannot repair the incorrect student program in Fig 1(b) since its Control-Flow Graph (CFG) differs from the CFG of instructor designed reference program in Fig 1(a). Our tool Verifix 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.

programming education. The only previous attempt on verified repair [17] is tightly tied to a specific structure of programs implementing dynamic programming.

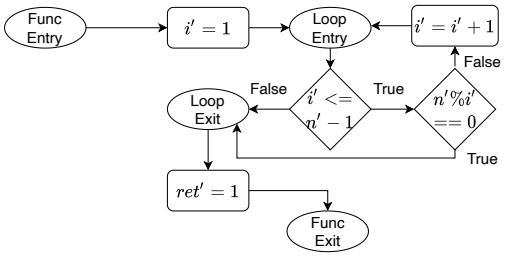
Repair tool: Verifix. We build our verified-repair technique by extending the existing program equivalence checking technique. Although automatically proving the equivalence between two programs remains challenging (mainly due to the difficulty of automatically finding loop invariants), we found that 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 [13, 33]. Exploiting this, Verifix produces a verified repair. Note that, Verifix performs 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 produces small-sized verified patches as feedback, which, whenever available, can be used by struggling students with high confidence. Our tool Verifix is available at https://github.com/zhiyufan/Verifix.

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 a student.

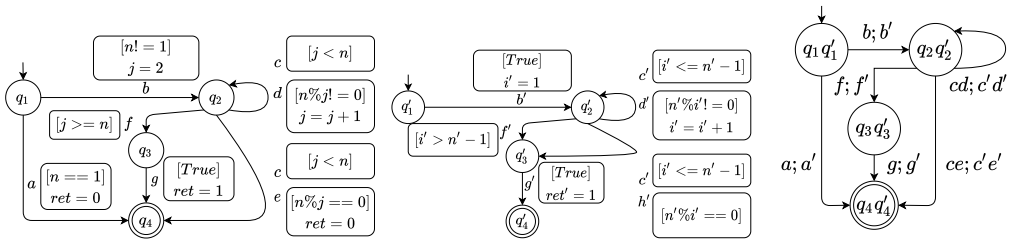


(a) CFG of the reference program in Fig 1(a)



(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 [13], Sarfgen [33] 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).

Limitations of the Existing Approaches. The state-of-the-art approaches such as Clara [13] and Sarfgen [33] make the same-control-flow assumption described as follows.

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

Clara fails to repair the incorrect program shown in Figure 1(b) when the reference implementation shown in Figure 1(a) is used, reporting that the structures of these two programs do not match. The CFGs of this incorrect program and its reference program are shown in Figure 2(b) and Figure 2(a), respectively. Notice that that in the reference CFG (Figure 2(a)), the LoopExit node has one incoming edge, whereas in the student program's CFG (Figure 2(b)) the matching LoopExit node has two incoming edges where the additional edge of Figure 2(b) comes from "$n'\%i' == 0$". The problem is that "$n'\%i' == 0$" does not match "n % j ==0" since the downward edge of node "n % j ==0" does not reach LoopExit, unlike in "$n'\%i' == 0$", and hence the structures of the two CFGs do not match. The fact that Clara treats a loop-free segment of the code as a single block does not help. In Clara, two adjacent nodes, "n % j ==0" and "ret = 0", of Figure 2(a) are grouped together, but the outgoing edge of this group still does not reach LoopExit.

A common approach that has been used to overcome this problem is to use multiple reference programs of diverse control-flow structures [13, 17, 33]. Since it would be labor-intensive for an instructor to prepare multiple reference implementations, recent works (e.g., [13, 33]) gets around

this problem by using student submissions. That is, student submissions that pass all tests are added into a pool of reference implementations. However, this approach exposes students to the risk of getting wrong feedback generated based on an incorrect program that happens to pass all tests.

Our Approach. We show how we address the aforementioned limitations. Essentially, we do not make the same control-flow-structure assumption. Instead, we conduct repair with Control Flow Automata (CFA) where its nodes represent program locations and its edges represent guarded actions. Figure 3 shows examples of CFAs, as will be described shortly in Section 2.1. Also, we extend the existing equivalence checking technique into a verified repair technique. We traverse each edge of the CFA obtained from a student submission and check its semantic equivalence with the corresponding edge of the CFA obtained from a reference program. Note that each edge represents a loop-free segment of a program. Equivalence checking is performed by encoding the problem into an SMT (Satisfiability Modulo Theories) formula. If equivalence checking fails, we reformulate the equivalence checking problem into a repair problem; we allow the expressions of the student submission to be replaced with the expressions of the reference program (after converting variable names). The number of replacements is minimized by encoding the repair problem into a MaxSMT (Maximum Satisfiability Modulo Theories) formula.

In the following, we show how 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 actions. Figure 3(a) and 3(b) show the CFA for the reference program ($\mathcal{A}_\mathcal{R}$) and the CFA for the student program ($\mathcal{A}_\mathcal{S}$), respectively. Notice that each edge of a CFA is annotated with a sequence of guarded actions. For example, in Figure 3(a), the edge between q_1 and q_2 is annotated with "$[n\,!=1]\,j=2$" where an assignment command $j = 2$ is guarded with the conditional expression $n\,!=1$. In the figure, we label this guarded action with "b". As another example, the self-edge of node q_2 is annotated with a sequence of two guarded actions, c and d, which indicates that c and d should be executed in sequence. As in the case of c, a guarded action can have only a conditional expression φ, which means that the NOP command is guarded with φ.

To perform verification/repair in the next phase, we build an aligned CFA $\mathcal{A}_\mathcal{F}$ by aligning the nodes and edges of $\mathcal{A}_\mathcal{R}$ and $\mathcal{A}_\mathcal{S}$. Figure 3(c) shows the aligned CFA for our running example. Notation q_1q_1' used in the entry node of Figure 3(c) denotes that node q_1 of $\mathcal{A}_\mathcal{R}$ and node q_1' of $\mathcal{A}_\mathcal{S}$ are aligned with each other. The other nodes of $\mathcal{A}_\mathcal{F}$ are interpreted similarly. Meanwhile, notation $cd; c'd'$ used in the edge between q_2q_2' and q_2q_2' denotes that guarded-command-sequence cd of $\mathcal{A}_\mathcal{R}$ is aligned with guarded-command-sequence $c'd'$ of $\mathcal{A}_\mathcal{S}$. To align nodes and edges, we use lightweight syntax-based approaches, as will be detailed in Section 4.1. Recall that the existing approaches [13, 33] fail to handle our running example, due to their same-CFG assumption. We relax this assumption by conducting node alignment and edge alignment separately. In our running example, after aligning node q_1 with q_1' and q_4 with q_4', we conduct edge alignment for the edge between q_1 and q_4 (annotated with guarded action a) by creating a fresh edge between q_1' and q_4' (annotated with a' in Figure 3(c)). Similarly, a new edge $c'e'$ is constructed between q_2' and q_4', corresponding to the edge ce between $q2$ and $q4$, during the alignment stage since no such edge exists in the student automata. Conversely, the edge $c'h'$ between q_2' and q_3' of the student's CFA is removed because no matching edge exists in the reference automata. Our experimental results

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'	$[n'\%i'! = 0]$	$i' = i' + 1$	$[n'\%i'! = 0]$	$i' = i' + 1$
e'	$\emptyset$		$[n'\%i' == 0]$	$ret' = 0$
f'	$[i' > n' - 1]$		$[i' > n' - 1]$	
g'	$[True]$	$ret' = 1$	$[True]$	$ret' = 1$
h'	$[n'\%i' == 0]$		$\emptyset$	

Table 2. Incorrect student blocks and their corresponding repairs generated by Verifix, after multiple rounds of edge verification-repair of Figure 3 aligned automaton. The blocks a' and e' are created in the automata, while the block h' is removed.

show that this simple extension alone reduces the structural alignment mismatch rate by 13% (see Table 4).

While in our example, only one aligned automaton can be constructed, there can be multiple ways to align $\mathcal{A}_{\mathcal{R}}$ and $\mathcal{A}_{\mathcal{S}}$ when multiple edges exist between two aligned nodes (Figure 5 shows an example). In such a case, we construct all possible aligned CFAs, and in the next phase (verification/repair phase), each aligned automaton is investigated to generate a minimal repair.

To conduct verification/repair, we also need to align variables used in $\mathcal{A}_{\mathcal{R}}$ and $\mathcal{A}_{\mathcal{S}}$. To align variables, we use a syntax-based approach similar to [33]. For each edge of $\mathcal{A}_{\mathcal{F}}$, we align variables whose usage patterns are similar to each other (see Section 4.2). For example, Verifix infers the following variable alignment predicate for the edge $q_1q_1' \rightarrow q_2q_2'$: $\{ret \leftrightarrow ret', n \leftrightarrow n', j \leftrightarrow i'\}$ 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 $\mathcal{A}_F$ ($a; a'$ and $b; b'$ for our Figure 3(c)). More specifically, we perform verification by checking whether $q \sim q'$ (i.e., q is bisimilar to q') holds for each aligned nodes q and q' of $\mathcal{A}_F$.

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

$$\varphi^1_{edge} : \phi_{q_1q_1'} \wedge \psi_r \wedge \psi^1_s \wedge \neg\phi_{q_2q_2'}$$

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.

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

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

$$\begin{aligned} \psi_r&: \quad (n_0 \neq 1 \implies j_1 = 2) \quad \wedge \quad (\neg(n_0 \neq 1) \implies j_1 = j_0) \\ \psi^1_s&: \quad (True \implies i_1' = 1) \quad \wedge \quad (\neg True \implies i_1' = i_0') \end{aligned}$$

If φ^1_{edge} is satisfiable, then $q_2 \sim q_2'$ does not hold, indicating verification failure. We check the satisfiability of φ^1_{edge} using an off-the-shelf SMT solver, Z3 [24].

2.3 Repair Phase

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

$$\phi^1_{ce} : n_0 = n'_0 = 1, j_0 = i'_0 = 0$$

where ϕ^1_{ce} can be viewed as a counter-example to the edge verification. When ϕ^1_{ce} holds, variable j_1 of the reference program has a value 0 (since $\neg(n_0 \neq 1) \implies j_1 = j_0$) by ψ_r), whereas variable i'_1 of the student program (aligned with j_1) has a different value 1 (since $True \implies i'_1 = 1$ by ψ^1_s), violating $\phi_{q_2 q'_2}$. Using this counter-example, we perform a repair based on counter-example-guided inductive synthesis or CEGIS strategy [31] (see Section 5.2). Following CEGIS strategy, we look for a repair of ψ^1_s which rules out the counter-example ϕ^1_{ce}. Verifix returns two potential repair candidates.

$$\begin{array}{lll} \psi^2_s\text{:} & (False \implies i'_1 = 1) & \wedge \quad (\neg False \implies i'_1 = i'_0) \\ \psi^3_s\text{:} & (n'_0 \neq 1 \implies i'_1 = 1) & \wedge \quad (\neg(n'_0 \neq 1) \implies i'_1 = i'_0) \end{array}$$

When ψ^2_s (or ψ^3_s) is substituted for ψ^1_s in $\varphi^1_{edge} \wedge \phi^1_{ce}$ (notice that the original formula φ^1_{edge} is conjoined with ϕ^1_{ce}), the modified formula is not satisfiable, indicating that under the context of the counterexample (i.e., ϕ^1_{ce}), ψ^2_s (or ψ^3_s) is a repair. Notice how the original formula ψ^1_s is repaired. In ψ^2_s and ψ^3_s, the original expression $True$ is replaced with $False$ and $n'_0 \neq 1$, respectively. To obtain $n'_0 \neq 1$, we use the expression $n_0 \neq 1$ appearing in ψ_r, the guarded action for the reference program. This copy mechanism that exploits the existence of a reference program is a de-facto standard technique in recent works [13, 33].

So far, we only showed that ψ^2_s (or ψ^3_s) is a repair only in the context of ϕ^1_{ce}. It is not known yet whether ψ^2_s (or ψ^3_s) is a repair in a general context. To check this, we retry edge verification for φ^1_{edge} after replacing ψ^1_s with ψ^2_s (or ψ^3_s) in φ^1_{edge}. In our example, verification attempt fails again for both ψ^2_s and ψ^3_s (that is, the repaired φ^1_{edge} is still satisfiable), and the following new counter-example ϕ^2_{ce} is obtained.

$$\phi^2_{ce} : n_0 = n'_0 = 2, i_0 = i'_0 = 0$$

By considering both ϕ^1_{ce} and ϕ^2_{ce}, Verifix returns a new repair candidate ψ^4_s,

$$\psi^4_s\text{:} \quad (n'_0 \neq 1 \implies i'_1 = 2) \quad \wedge \quad (\neg(n'_0 \neq 1) \implies i'_1 = i'_0)$$

As compared with ψ^1_s, two sub-expressions of ψ^1_s are repaired. As in ψ^3_s, $True$ is replaced with $n'_0 \neq 1$. Also, $i'_1 = 1$ is replaced with $i'_1 = 2$ based on $j_1 = 2$ appearing in ψ_r. This updated repair candidate ψ^4_s rules out all counter-examples seen so far, and no further satisfying assignments of φ^1_{edge} are found. This completes the verification and repair, thereby repairing the edge b' in Figure 3(b). The remaining edges are similarly verified/repaired, and Table 2 summarizes the buggy student automata $\mathcal{A}_S$ edges and their corresponding repairs generated by our repair tool Verifix.

We note that Verifix generates a minimal repair for each aligned edge under consideration. That is, a generated edge repair modifies the minimum number of expressions required to repair the edge (see Theorem 4). To obtain a minimal edge repair, we formulate a repair problem as a partial MaxSMT problem, as described in Section 5.2. Essentially, Verifix tries to preserve as many original expressions as possible, by assigning a higher weight penalty to the original expressions (hence, replacing an original expression increases the cost of repair). While combining minimal edge repairs does not necessarily lead to a globally minimal repair, our experimental results suggest that our greedy approach works well in practice. Verifix tends to generate smaller repairs than a state-of-the-art tool Clara (see Section 7.4).

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 [34] which uses GenProg [19], operate on program ASTs directly.

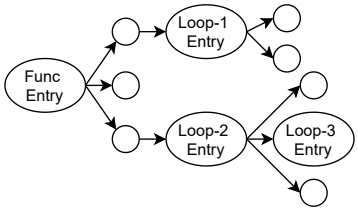
Control Flow Graph (CFG). Existing state-of-art APR techniques like Clara [13] and SarfGen [33] operate at the level of CFG, whose nodes are basic blocks and edges denote control transfer. We extend the standard CFG by introducing four types of special labelled nodes: {*Func-Entry*, *Loop-Entry*, *Func-Exit*, 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.

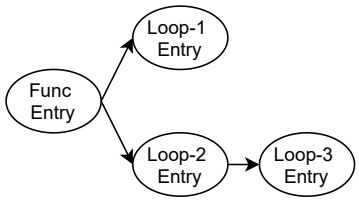
Control Flow Automaton (CFA). Our tool Verifix operates at the level of the control flow automaton (CFA), often used by model-checking and verification communities [15]. The CFA is essentially the CFG, with code statements labeling the edges of CFA, instead of code statements labeling nodes as in CFG. 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 Automaton). *A Control Flow Automaton (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,*

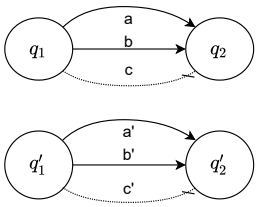


(a) Example AST with labelled and unlabelled nodes.

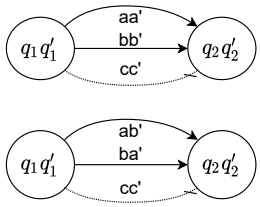


(b) Example AST^L after deletion of unlabelled nodes.

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



(a) Example $\mathcal{A}_R$ and $\mathcal{A}_S$.



(b) Example $\mathcal{A}_\mathcal{F}$ after alignment.

Fig. 5. Example demonstrating edge alignment. Given node alignment $V : \{q_1q'_1, q_2q'_2\}$, 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.

- Ψ *is a mapping from edge* e *to* ψ_e *for all edges* e*, where* ψ_e *is the set of guarded actions labeling* e*, and*
- 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. How we build a CFA is described in Section 4.

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

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

To construct an aligned automaton $\mathcal{A}_\mathcal{F}$, we first conduct node alignment between the nodes of $\mathcal{A}_S$ and $\mathcal{A}_R$. This step is followed by aligning the transition edges between $\mathcal{A}_S$ and $\mathcal{A}_R$. A more detailed description is provided below.

Node Alignment. Given two CFAs $\mathcal{A}_\mathcal{S}$ 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 structures of AST_S^L and AST_R^L are identical with each other, 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 $\mathcal{A}_R$.
(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 $q_1 \rightarrow q_2$ (resp. $q_1' \rightarrow q_2'$). Since both the AST^L trees are structurally the same, the node alignment V consists of $\{q_1q_1', q_2q_2'\}$ 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'. Similarly, 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 $\mathcal{A}_R$. Node alignment fails if the two programs have different different function/looping structure from each other. While limited, our approach can handle more diverse programs than the state-of-the-art approaches [13, 33] which require not only bijective mapping between nodes but also bijective mapping between edges. In these approaches, q_4 and q_4' of Figure 3 cannot be aligned with each other, since the edge $q_2 \rightarrow q_4$ of $\mathcal{A}_R$ does not have a corresponding edge in $\mathcal{A}_S$.

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 four 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) Only $\mathcal{A}_S$ has an edge from u_S to v_S of type t, while $\mathcal{A}_R$ has no edge from u_R to v_R of type t.
(4) 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_1q_1' and q_4q_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. The third case is the opposite of the second one. In this case, we remove the edge between u_S and v_S since there is no matching edge in the reference automata.

Lastly, in the fourth case, there exist several possible edge alignments of the order of $\binom{M}{N} \times N!$, where M is the number of edges from $u_R \rightarrow v_R$ and N is the number of edges from $u_S \rightarrow v_S$.

Figure 5 demonstrates this case through an example, resulting in two possible edge alignments. The single *break* transitions c and c' are aligned with each other, while the remaining *normal* edges (i.e., a, b, a' and b') are aligned combinatorially to produce two unique aligned automata. When multiple aligned automata can be constructed, we choose the edge alignment which maximizes the number of verified-equivalent edges in the resultant aligned automaton $\mathcal{A}_{\mathcal{F}}$. The formal structure of aligned automaton is described 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,*
- *$v^0 : v^0_S \leftrightarrow v^0_R$, where v^0_S and v^0_R are the initial function entry nodes of the automata $\mathcal{A}_S$ and $\mathcal{A}_R$ respectively,*
- *$v^t : v^t_S \leftrightarrow v^t_R$, where v^t_S and v^t_R 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 a 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, and*
- *$Pred : Var_S \leftrightarrow Var_R$, denoting variable alignment, is a bijective mapping between variables of $\mathcal{A}_S$ and $\mathcal{A}_R$.*

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 [33]. Our approach for computing a 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 $usage(x', e_i)$. If 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 [13, 33] 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 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, $((x_1 > 1) \implies x_2 = x_1 + 1) \wedge (\neg(x_1 > 1) \implies x_2 = x_1)$. 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.

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

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:

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

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 $\phi_u \wedge \psi_r \wedge \psi_s$ (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 user-defined function calls such as *check_prime*, 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 [31]. 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 the case 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 [13, 33] 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 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.

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

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 $h^{(i)}$ 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 $h^{(i)} \in S[\![h^{(i)}]\!] \backslash \{e^{(i)}\}$.

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.

Algorithm 1 Edge verification-repair

Input: Aligned *edge*
Output: Verified/Repaired *edge*

Let $\phi_u \equiv$ edge.sourceNode.invariants
Let $\phi_v \equiv$ edge.targetNode.invariants
Let $\psi_r \equiv$ edge.label.reference
Let $\psi_s \equiv$ edge.label.student
$CEs \leftarrow []$ // List of counter-examples
$candidates \leftarrow [\psi_s]$
repeat
    // Step 1: attempt for edge verification
    for each ψ_s^i in $candidates$ **do**
        Let $\varphi_{edge}^i \equiv \neg(\phi_u \wedge \psi_r \wedge \psi_s^i \implies \phi_v)$
        if $\not\models \varphi_{edge}^i$ **then** // UNSAT
            $edge.label.student \leftarrow \psi_s^i$ // Update edge
            return ✓ // Verifiably correct
        else
            $\phi_{ce}^i \models \varphi_{edge}^i$ // SAT
            $CEs \leftarrow CEs \cdot \phi_{ce}^i$
        end if
    end for
    // Step 2: make holes in ψ_s
    Let $\psi_s^+ \equiv Extend(\psi_s, \psi_r)$
    Let $\psi_f \equiv RepairSketch(\psi_s^+)$
    // Step 3: define implementation space
    $\varphi_{hard} \leftarrow []$; $\varphi_{soft} \leftarrow []$
    for each ϕ_{ce}^i in CEs **do**
        $\varphi_{hard} \leftarrow \varphi_{hard} \cdot (\phi_{ce}^i \wedge \psi_r \wedge \psi_f \wedge \phi_v)$
    end for
    for each $hole, expr, weight$ in $RepairSpace(\psi_f, \psi_r, \psi_s)$ **do**
        $\varphi_{soft} \leftarrow \varphi_{soft} \cdot (hole = expr, weight)$
    end for
    // Step 4: search for a repair
    if $\not\models (\varphi_{hard}, \varphi_{soft})$ **then** // UNSAT or UNKNOWN
        return ✗ // Repair Failure
    else
        // Update $candidates$ using a pMaxSMT solver
        // There can be multiple candidates
        $candidates \models (\varphi_{hard}, \varphi_{soft})$
    end if
until timeout

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.

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

Proof. Recall that for each repaired/verified edge $u \xrightarrow{\psi_s;\psi_r} v$ of a repaired automaton $\mathcal{A}'_F$, $\{\phi_u\}\psi_s;\psi_r\{\phi_v\}$ holds. By structural induction on the edges of $\mathcal{A}_F$, the post-condition of $\mathcal{A}'_F$'s final node holds true, and hence $out = out'$ holds for the outputs aligned between $\mathcal{A}_S$ and $\mathcal{A}_R$. Note that for introductory programming assignments, output is clearly known (such as the return value of the program), and we enforce the post-condition of $\mathcal{A}'_F$'s final node to contain $out = out'$. □

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 [9]:

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

Proof. Whenever edge verification fails, Algorithm 1 performs repair in step 4 of the algorithm. In case a repair exists in the repair space, Algorithm 1 reaches line 36, and a pMaxSMT solver is fed with formula (3) to find out a repair. Thus, if the MaxSMT/pMaxSMT solver is complete, a repair is always generated. □

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.

Proof. Assume the three assumptions are met. Since Verifix traverses all edges of $\mathcal{A}_F$ one by one without backtracking, it suffices to show that each edge is repaired by Algorithm 1 which at a high level consists of the following two parts: verification (step 1 of the algorithm) and repair (step 2, 3, and 4).

First, consider the verification part. Verification is performed via a MaxSMT solver which returns either (a) UNSAT (line 11) or (b) SAT (line 15) for φ^i_{edge} (see line 10). Note that the UNKNOWN case is excluded by the third assumption. In case (a), edge verification is done. In case (b), the algorithm moves to the repair part which we now consider.

In the repair part, a pMaxSMT solver is invoked at line 31 and 36 of Algorithm 1 and returns either (i) UNSAT or (ii) SAT. The UNKNOWN case is excluded by the third assumption. Case (i) happens only when the second assumption is violated (that is, a repair is not in the implementation space), and we exclude this case from consideration. In case (ii), repair candidates are obtained (line 35), and verification is re-attempted to see if one of the obtained candidates ψ'_s satisfies $\{\phi_u\}\psi'_s;\psi_r\{\phi_v\}$.

The repetition between repair and verification is guaranteed to terminate, since the implementation space is finite. This concludes the proof. □

Lastly, we consider the minimality of repair. In Verifix, use of MaxSMT guarantees the minimality of edge repair.

Theorem 4 (Minimality of edge repair). *Suppose that our algorithm repairs edge $e : u \rightarrow v$ 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 $C_r \cup U_r$.*

Proof. Recall that we pass formula (3) to a pMaxSMT solver. In the formula, the number placeholder variables $h^{(i)}$ defines the maximum size of edge repair, and a minimal edge repair is obtained when the minimum number of placeholder variables $h^{(i)}$ are equated with expressions different from their original expression $e^{(i)}$, which happens when expression $h^{(i)} = e^{(i)}$ in formula (3) is ignored by the pMaxSMT solver. Since a pMaxSMT solver ignores the minimum number of $h^{(i)} = e^{(i)}$, the stated theorem holds. □

Theorem 4 does not necessarily guarantee the global minimality of a generated repair. In the following theorem, we identify the conditions that should be additionally satisfied to guarantee global minimality.

Theorem 5 (Global minimality). *A repaired program generated by our algorithm is minimal if the following conditions hold:*

(1) Node alignment made in $\mathcal{A}_F$ is optimal in the sense that there is no alternative node alignment (other than the one generated by Verifix) that can lead to a smaller repair.
(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.

Proof. Once node alignment and invariants of $\mathcal{A}_F$ are fixed, repairing a student program amounts to repairing each edge of $\mathcal{A}_F$ for which edge verification fails. Since each edge is repaired minimally (Theorem 4), the stated theorem holds. □

Verifix currently does not guarantee the global minimality of repair. Node alignment and variable alignment made by Verifix are not necessarily optimal. Instead of considering all possible alignments, we use a heuristics-based approach for the sake of efficiency. Nonetheless, our experimental results show that Verifix tends to find smaller repairs than Clara. Note that the existing approaches designed to generate minimal repairs [13, 33] 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 all possible edge alignments between aligned nodes.

6 EXPERIMENTAL SETUP

6.1 Research Questions

We address the following research questions in this work.

(1) RQ1: How does Verifix perform in terms of the repair success rate, as compared to state-of-the-art approaches? While Verifix generates verifiably correct repair, is the repairability comparable to the existing approaches?
(2) RQ2: How does Verifix perform in terms of running time? Given that Verifix uses heavy-weight SMT techniques to conduct verification, slowdown in running time as compared to non-verification approaches is expected. How severely is the time performance affected?

(3) RQ3: What are the reasons for repair failure in Verifix? The answers to this question can be used to identify where to improve in the future work.
(4) RQ4: Does Verifix generate small sized repair? In a pedagogical setting, small repairs are usually desired. While Verifix generates a minimal edge repair, it does not guarantee to generate a globally minimal repair. What is the practical consequence of this greedy approach?
(5) RQ5: What is the effect of test-suite quality on repair when a test-based approach is used? We ask this question to compare the existing test-based approaches with Verifix which does not require a test.
(6) RQ6: How is the repair success rate of Verifix affected by the number of reference solutions? We ask this question to assess the performance of Verifix when multiple reference implementations are available.

6.2 Dataset

Evaluation of a programming assignment feedback tool requires a dataset of incorrect student assignments. For our dataset, we chose a publicly released dataset curated by ITSP [4] [34] 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. Other datasets used in previous work are either not publicly available [13, 17, 33] or use different programming languages than C [16].

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. Our implementation currently does not support all programming language constructs such as pointers, multi-dimensional arrays, and struct, which are necessary to support the remaining labs in the ITSP benchmark. Note that support for more programming language constructs is orthogonal to our verified-repair generation algorithm. As more programming language constructs are supported, our repair algorithm can be used without modification to repair more diverse programs, these are left as future work.

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 [13] 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] [13] 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 repair. We do not directly compare our results against CoderAssist [17] tool since it does not work with our dataset (CoderAssist targets dynamic programming assignments), while Refactory [16] implementation targets Python programming assignments. About SarfGen, we could not obtain access to the tool from its authors due to a copyright issue (SarfGen is commercialized). We instead

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

Lab-ID	Topics	# Assign-ments	# Prog-rams	Repair (%)		Avg. Time (sec)	
				Clara	Verifix	Clara	Verifix
Lab-3	Floating point, printf, scanf	4	63	54.0%	92.1%	2.0	39.7
Lab-4	Conditionals, Simple Loops	8	117	71.8%	74.4%	32.9	34.2
Lab-5	Nested Loops, Procedures	8	82	22.0%	45.1%	10.2	12.5
Lab-6	Integer Arrays	8	79	12.7%	21.5%	14.2	8.1
Overall	-	28	341	42.8%	58.4%	21.3	29.5

Table 3. Lab-wise repair success rate (shown in the Repair column) of our tool Verifix and Clara [13]. Time column represents the average runtime in seconds for all successfully repaired programs. The number of assignments in each lab is shown in the *#Assignments* column, and the number of incorrect student submissions in each lab is shown in the *#Programs* column.

address these comparisons in our related work Section 10. Our tool Verifix [6] is publicly released to aid further research.

Our experiments were carried out on a machine with Intel® Xeon® E5-2660 v4 @ 2.00 GHz processor and 64 GB of RAM.

6.3 Implementation

Verifix supports repairing compilable incorrect C programs, given a reference C program and optional test cases. Verifix implementation is composed of three components: (1) Setup, (2) Verification, and (3) Repair generation.

For the setup phase, we build on top of Clara [5] [13] parser to convert incorrect and reference C programs into a Control-Flow Graph (CFG) representation. We then convert the obtained CFGs into its dual Control-Flow Automata (CFA), and align the reference CFA with incorrect CFA.

In the verification phase, the reference and student program labels on each aligned edge are converted into a Single Static Assignment (SSA) format using our custom Verification Condition Generator (VCGen) implementation. We make use of Z3 [24] SMT solver to verify if the aligned edges are equivalent.

In the repair phase, we encode each repair candidate using Boolean selectors. Z3 pMaxSMT solver is used to select the repair with minimal cost. The final repaired CFA/CFG internal representation is converted back into a program using a custom concretization module (reverse VCGen). After which, we make use of Zhang-Shasha [7] tree-edit distance algorithm [37] to compute the patch size between incorrect student program and the repaired student program.

7 EVALUATION

7.1 RQ1: Repair success rate

Table 3 compares the repair success rate of our tool Verifix against the state-of-art tool Clara [13] on the common dataset of student submissions. Given a single reference implementation per assignment, Verifix achieves an overall repair success rate of 58.4% on the 348 incorrect programs across 28 unique assignments. In comparison, the baseline tool Clara achieves a lower overall repair success rate of 42.8% 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. That is, Verifix generates a verifiably correct feedback for 58.4% of student submissions in diverse assignments, which is not possible using existing test-based approaches.

[6] https://github.com/zhiyufan/Verifix
[7] https://github.com/timtadh/zhang-shasha

Lab-ID	# Prog-	Repair (%)		Struct. Mismatch (%)		Timeout (%)		Unsupported (%)		SMT issues (%)	
	rams	Clara	Verifix	Clara	Verifix	Clara	Verifix	Clara	Verifix	Clara	Verifix
Lab-3	63	54.0%	92.1%	0.0%	0.0%	42.9%	0%	3.2%	3.1%	0%	4.8%
Lab-4	117	71.8%	74.4%	7.7%	7.7%	19.6%	10.3%	0.9%	0.9%	0%	6.8%
Lab-5	82	22.0%	45.1%	75.6%	35.4%	1.2%	11.0%	1.2%	1.2%	0%	7.3%
Lab-6	79	12.7%	21.5%	83.5%	69.6%	2.5%	0%	1.3%	1.3%	0%	7.6%
Overall	341	42.8%	58.4%	40.2%	27.2%	15.5%	6.2%	1.5%	1.5%	0%	6.7%

Table 4. The distribution of the four reasons for repair failure, i.e., structural mismatch (4th column), timeout (5th column), unsupported language constructs (6th column), and SMT issues (7th column). The first three columns are copied from Table 3.

The improvement in repair success rate of Verifix over Clara is partly due to the more flexible structural alignment of Verifix than that of Clara. Recall that Verifix uses a more relaxed structural alignment, as compared to the stricter structural alignment used by existing state-of-the-art approaches including Clara, as described in Section 4.1. Verifix requires the reference and incorrect Control-Flow Automata (CFA) to have the same number of program states or nodes, denoting functions and loops. While Clara additionally requires the reference and incorrect CFA to have the same number of edges, denoting return/break/continue transitions. In Section 7.3, we investigate the common reasons for repair failure.

7.2 RQ2: Running time

The *time* column of Table 3 shows the average running time of Verifix and Clara, in seconds. Verifix on average takes 29.5s to successfully repair an incorrect program, as compared to 21.3s on average by Clara. The running time of Verifix is particularly high in Lab-3 (39.7s) and Lab-4 (34.2s), whereas in Lab-6, Verifix runs significantly faster than Clara (8.1s vs 14.2s). The high running time of Verifix in Lab-3 and Lab-4 seems due to the fact that Lab-3 and Lab-4 programming assignments involve non-linear arithmetic expressions. For example, one of the Lab-4 assignments is on *computing the distance between two co-ordinate points*, which involves square-root computation. Note that SMT solvers generally run slow when non-linear arithmetic expressions are used in the input formula. There has been an effort to handle non-linear arithmetic more efficiently [10], and Verifix can be benefited from the improvement of the SMT techniques.

We also note that while Clara runs faster than Verifix across the labs except for in Lab-6, its repair success rate is always lower than that of Verifix across all labs. For example, in Lab-3, Clara's average running time is only 2.0s, but its repair success rate is only 54.0%, which is 38.1% lower than that of Verifix (92.1%). Overall, while Verifix, which uses heavy-weight SMT techniques, tends to require more running time than Clara, the overall results are nuanced by the other facts such as repair success rate and correctness guarantee.

7.3 RQ3: Reasons for repair failure

Table 4 shows the distribution of the repair failure reasons for Verifix and Clara. *Structural Mismatch* (shown in the 4th column) is the primary reason for repair failure of Verifix and Clara, accounting for 27.2% and 40.2% of all the 341 incorrect student programs, respectively. Recall that a single reference solution is used for each assignment in the labs. For simpler programs such as those in Lab-3 and Lab-4, both tools achieve low structural mismatch rate. That is, almost all the incorrect student programs can be structurally aligned with the reference program. As the complexity of the programs increases (in our dataset, as the lab ID increases, the students submissions tend to be more complex), the structural mismatch rate tends to increase in both tools. However, the

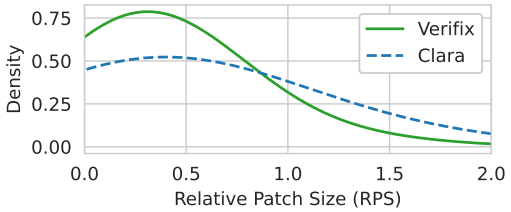


Fig. 6. Kernel Density Estimate (KDE) plot of Relative Patch Size (RPS) by Verifix and Clara on 132 common successful repairs.

rate increases more gently in Verifix than in Clara. For example, in Lab-5, while the structural mismatch rate of Clara drastically increases to 75.6%, Verifix maintains a much lower mismatch rate of 35.4%. This difference in structural match rate results in a overall higher repair success rate in Verifix as compared to Clara. For example, 45.1% of Verifix vs 22.0% of Clara for Lab-5. The high structural mismatch rates in Lab-6 are related to the following: many incorrect students' programs use function calls, but the reference programs often do not have functions with matching function signatures.

The second biggest failure reason is *Timeout* (5 minutes), accounting for 6.2% and 15.5% of the dataset for Verifix and Clara, respectively. In Verifix, most of the running time is spent on SMT and pMaxSMT solving by Z3 solver during verification and repair stage, respectively. In 1.5% of student programs, repair failure occurs since our current implementation does not support all programming language constructs used in our datasets. For example, both Verifix and Clara currently do not support the *GOTO* statement. Lastly, in 6.7% of the incorrect programs, Verifix fails to generate a repair due to the incompleteness of SMT solving. Common cases of this kind are when the SMT solver returns UNKNOWN result, instead of SAT or UNSAT, during the verification or repair phase.

7.4 RQ4: Minimal repair

To investigate this research question, we compare the sizes of repairs generated from Verifix and our baseline state-of-art tool Clara [13]. 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.

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 132 common programs. Out of these 132 commonly repaired programs, Verifix generates a patch with smaller RPS in 67 of the cases, Clara generates a patch with smaller RPS in 47 of the cases, and both tools generate a patch of the exact same relative patch size in 18 cases. Note that in the case of Clara, a smaller repair does not necessarily imply better quality repair since these repairs can overfit the test cases (see Section 7.5).

Figure 6 plots the Kernel Density Estimate (KDE) of Relative Patch Size (RPS) for these 132 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)

```
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.

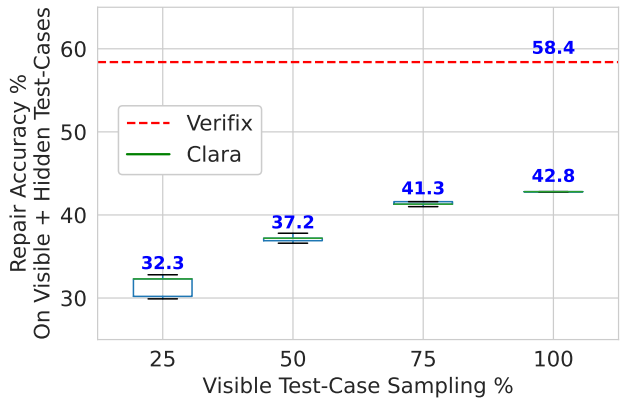


Fig. 8. Repair accuracy of Clara and Verifix on various test case samplings.

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.

7.5 RQ5: Overfitting

Majority of the programming assignment repair tools [13, 16, 33, 34] generate repairs that satisfy a given test suite (incomplete specification). Verifix is distinguished from these existing test based approaches in that it generates a verifiably correct repair. Figure 7 demonstrates an example from a Lab-5 *Prime Number* assignment, where Clara's [13] 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 impact 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 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 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.

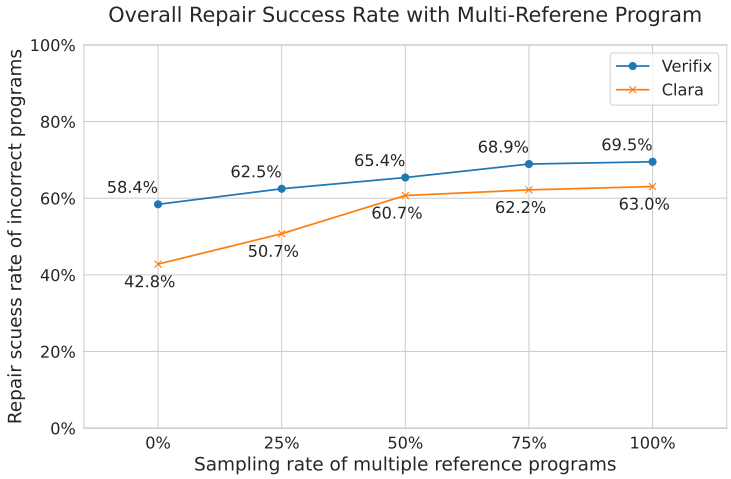
Moreover, the choice of test case sampling has a large effect on Clara's repair accuracy, as evident from the variation in box-plot distribution. In the case of 25% visible test case sampling, Clara's repair accuracy ranges from a minimum value of 29.9 to maximum of 32.8; depending on which two test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

Hence, APR tools such as Clara [13] 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.

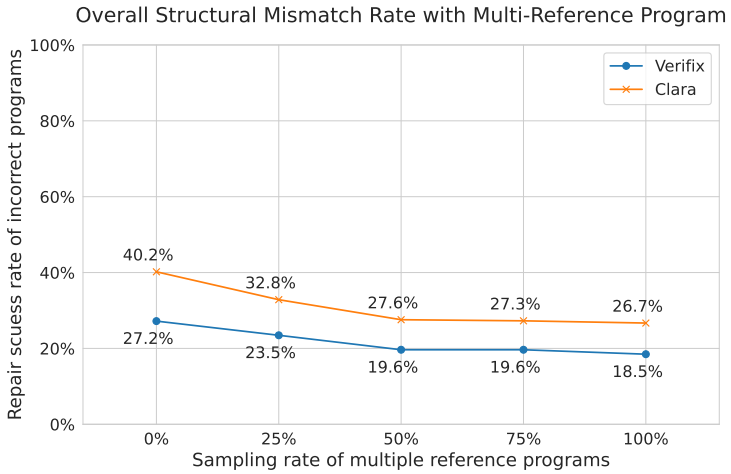
7.6 RQ6: Repair success rate with multiple reference implementations

In the previous sections, we conducted experiments with a single reference implementation for each assignment. Several previous works, including Clara [13] and SarfGen [33], assume the prevalence of multiple reference solutions to help alleviate structural matching issues. In this section, we compare the repair success rate of both Verifix and our baseline tool Clara [13], on being provided access to multiple reference solutions. As additional reference implementations, we use 341 student submissions in the ITSP dataset [34] that pass all test-cases. While passing all tests does not guarantee the correctness of a program, previous works [13, 33] used similar approaches.

To evaluate the change in repair success rate on providing access to multiple reference implementations, we run Verifix and Clara with diverse sampling rates of 0%, 25%, 50%, 75%, and 100%; for each sampling rate of $N\%$, we randomly sample $N\%$ of all available reference implementations,



(a) Overall repair success rates for all labs



(b) Overall structural mismatch rates for all labs

Fig. 9. Repair success rates and structural mismatch rates across different sampling rates of multiple reference solutions. The X and Y axes represent the sampling rate of the reference solutions and the observed repair success rate, respectively.

in addition to the instructor-provided reference program. For example, 0% sampling rate indicates only the instructor provided reference solution was used (single-reference program). While 100% indicates that all reference programs were made available for the repair tool, in addition to the instructor provided reference program. To prevent a student's incorrect program P being repaired by his/her own final submission P' that passes all test-cases, we exclude P' from the sampled set of multi-reference programs (if it exists) when P is being repaired. We run our baseline tool Clara [13] in its default mode for multi-reference programs; its clustering algorithm is first executed on the set of sampled reference implementations, followed by running its repair algorithm on each incorrect program using the obtained clusters.

The results of multi-reference experiments are shown in Figure 9. Figure 9(a) shows how repair success rate changes as more reference programs are used, while Figure 9(b) shows how structural mismatch rate changes. A student submission S is considered structurally mismatched with a sampled group of reference programs G when no program in G structurally matches S. From Figure 9(a) we observe that the repair success rate increases for both Verifix and Clara, as more reference implementations are made available for repair. From Figure 9(b) we note that this is primarily due to a reduction in structural alignment mismatch between the set of multiple reference implementations (with more diverse program structures) and the given incorrect program.[8] We note that similar observations have been made regarding the effect of multi reference programs on repair success rate in prior work [16].

From Figure 9(a) we observe that Verifix achieves a higher success rate over Clara across all sampling rates. The gap between the repair rate of both tools reduces as more reference programs are provided, indicating that Clara's repair success rate could eventually match that of Verifix's on being provided a large number of reference solutions. From Figure 9(b) we observe that Verifix maintains a lower structural mismatch rate over Clara across all sampling rates. When all reference solutions are used, structural mismatch rate of Verifix and Clara drops down to 18.5% and 26.7%, respectively. This result demonstrates the benefit of using Verifix's CFA (Control-Flow-Automata) based structural alignment algorithm over Clara's CFG (Control-Flow-Graph) based alignment algorithm, even in the case of multi-reference solutions.

8 USER STUDY

In order to evaluate the usefulness of the repair generated by Verifix, we conducted a user study of tutors of introductory programming courses. Note that students have expressed positive feedback about using feedback generation systems such as Clara [13] and SarfGen [33]. Verifix uses the same copy mechanism for repair as these tools (i.e., parts of a reference implementation are copied) and can generate the same style of feedback. The main difference between Verifix and the existing tools lies in that Verifix generates verifiably correct repairs. We believe that tutors can better appreciate the quality of repairs than novice students, and our user study sheds helpful light on understanding its pedagogical value. A user study with students is left as future work.

8.1 User Study Questionnaire

In this user-study, we explored the practical value of Verifix in aiding tutors in the task of grading and providing feedback on incorrect student submissions. This was explored using the following questions:

(1) Rate the quality of the generated repair (in terms of semantic correctness, size, etc).
(2) Rate the possibility that you would like to use the repair (either complete or partial) as feedback to the student.
(3) Rate the possibility that you would like to use the repair indirectly: to help formulate your own custom feedback to student.
(4) Rate the possibility that these repairs can help you in grading?
(5) Will examples of student incorrect submissions and repairs like these help you in improving the grading policy?
(6) If the repair is known to be verifiably (provably) correct, does it give you more confidence in using it?

[8]Repair failures may also occur due to reasons other than structural mismatch, as discussed in Section 7.3.

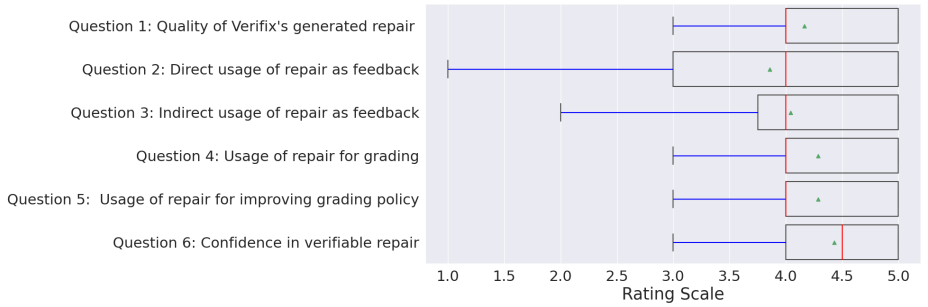


Fig. 10. Boxplot of the responses–with the scales from 1 (very low) to 5 (very high)–collected from 14 tutors. Red line represents the median value and green triangle represents the mean value. The whiskers denote the minimum/maximum value, and the rectangle denotes the first/third quartile.

8.2 User Study Setup

To answer the above questions, we circulated a Google-Form survey among the tutors of introductory programming courses at NUS (National University of Singapore) and UNIST (Ulsan National Institute of Science and Technology). After which, 14 tutors in total volunteered for this survey and completed their responses. For this survey, we randomly selected 10 incorrect student submissions from our benchmark of 341 programs, on which Verifix could successfully generate a repair. For each incorrect student program, the tutors were shown the assignment title, assignment description, a sample testcase, and the differences between a student-written buggy program and its repair generated by Verifix. All the volunteered tutors were shown the same 10 incorrect student submissions in the same order. One of the incorrect student submissions used in our user study is shown in Appendix A.

The tutors were asked three questions (questions 1–3 listed in Section 8.1) for each buggy student submission, followed by three questions (questions 4–6 listed in Section 8.1) as an overall summary at the end of the user study. The tutors were asked to provide their ratings on a numeric scale from 1 (very low) to 5 (very high) for each question.

8.3 User Study Results

The overall result of the 14 tutor responses is summarized using boxplots in Figure 10. From Figure 10 we note that the tutors responded with an overall positive rating for all six of our questionnaire (Q1–Q6), with a mean/median value of >= 3.8 in all the cases. We observe that the tutors rate the quality of Verifix generated repairs (Q1) highly, with a mean/median rating of ~4.0. Our tool's verification capability improved the tutors' confidence in accepting our generated repairs, with a mean/median rating of ~4.4. The tutors, on average, found Verifix's repair useful for providing feedback to students, both directly (Q2) and indirectly (Q3), giving a mean/median rating between 3.5–4.0. While a larger variation is observed in the case of direct usage of repair as feedback (Q2), this discrepancy reduces for indirect usage of repair as feedback (Q3), where tutors can quickly design customized feedback using the generated repair. The tutors agreed on the utility of Verifix's repair in grading (Q4) and in improving grading policy (Q5), giving a mean/median rating between 4–4.5.

9 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 increase the size of a generated patch. This however occurs rarely in practice, as demonstrated by our RQ4 (Section 7.4).

The arithmetic theory of SMT solvers is incomplete for non-linear expressions, which can affect our relative completeness. However, this issue affects 6.7% 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 success rate. We mitigate this risk by noting that such an evaluation has been undertaken earlier [16, 33], and would benefit both Verifix and our baseline tool Clara in terms of reduced structural mismatch rate. Furthermore, we cannot always assume the availability of a large number of reference solutions, in general.

10 RELATED WORK

10.1 General Purpose Program Repair

Automated Program Repair (APR) [11, 23] 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 [19], SemFix [26], Prophet [20] and Angelix [22], require an incomplete correctness specification typically in the form of a test-suite. These techniques achieve low repair success rate on student programs that suffer from multiple mistakes, since they can scale to large programs but not necessarily to large repair search spaces [34]. As student programs are substantially incorrect, the search space of repairs is typically large.

ITSP [34] 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 success rate and reliance on test-cases (overfitting) can be seen as a motivator for our work.

S3 [18] synthesizes a program using a generic grammar and user-defined test-cases. Semgraft [21] 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.

10.2 Repair of Programming Assignments

Autograder [30] 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 [13] 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 [33] 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 [16] 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 [17], 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, 5, 6, 14, 27, 32] for program repair. These approaches typically target syntactic/compilation errors, and the repair rate for semantic/logical errors is low [27]. Such machine learning based techniques do not offer any relative completeness guarantees, and the repair is evaluated against incomplete specification (e.g. tests).

There has been prior work on live deployment of APR tools for repairing student programs [2, 34]. The work of ITSP [34] shows negative results on providing semantic repair feedback to students on their 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 from 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.

10.3 Program Equivalence Verification

Verifix performs program equivalence verification which itself is a separate long-standing research area [4, 7, 25, 36]. In program equivalence verification, it is proved whether given two programs are semantically equivalent to each other. Program equivalence verification is usually performed by first constructing a product program (similar to our aligned CFA) where the loops of the two programs are aligned with each other [4, 36]. Aligning loops is considered as one of the major challenges in program equivalence verification [7]. In the traditional application areas of program equivalence verification such as optimized-code verification [8, 25], the original code and its optimized code often have different program structures, and thus alignment is challenging in those programs. This problem is much less severe in introductory programming assignments, as shown in our experiments, where Verifix fails to obtain a repair due to structural mismatch between the student and reference program in 27.2% of our dataset. The main difference of our work from program equivalence verification is that we add a CEGIS (counter-example-guided inductive synthesis) loop inside the verification procedure, so that repair and verification can take place hand in hand.

11 DISCUSSION

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 success rate than existing approaches on our common benchmark. The repairs produced by Verifix are of better quality than state-of-art techniques like Clara [13], since they are often smaller in size, while being verifiably equivalent to the instructor provided reference implementation.

We feel that technologies like Verifix have a place in intelligent tutoring systems of the future. Specifically, they may be used to give feedback to struggling students learning programming. Since Verifix generates verifiably correct repairs, it can used first for generating feedback. If Verifix is able to generate a feedback, it can used with confidence. For the cases where Verifix is unable to generate a feedback, other heuristic based student feedback generation approaches may then be used. We envision such a workflow for future intelligent tutoring systems for teaching programming.

ACKNOWLEDGMENTS

This work was partially supported by a Singapore Ministry of Education (MoE) Tier 3 program "Automated Program Repair". This work was also partly supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No.2021R1A2C1009819, No.2021R1A5A1021944) and the Institute for Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2021-0-01001).

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] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011. Relational verification using product programs. In *International Symposium on Formal Methods*. Springer, 200–214.

[5] 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.

[6] 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.

[7] 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.

[8] Berkeley Churchill, Rahul Sharma, JF Bastien, and Alex Aiken. 2017. Sound loop superoptimization for google native client. *ACM SIGPLAN Notices* 52, 4 (2017), 313–326.

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

[10] Pascal Fontaine, Mizuhito Ogawa, Thomas Sturm, Xuan Tung Vu, et al. 2018. Wrapping computer algebra is surprisingly successful for non-linear SMT. In *SC-square 2018-Third International Workshop on Satisfiability Checking and Symbolic Computation*.

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

[12] Sumit Gulwani, Ivan Radicek, and Florian Zuleger. 2014. Feedback generation for performance problems in introductory programming assignments. In *FSE*. 41–51.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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)*.

[22] 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.

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

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

[25] George C Necula. 2000. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. 83–94.

[26] 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.

[27] 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.

[28] 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)*.

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.

[34] 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.

[35] Jooyong Yi, Shin Hwei Tan, Sergey Mechtaev, Marcel Böhme, and Abhik Roychoudhury. 2018. A correlation study between automated program repair and test-suite metrics. *Empirical Software Engineering* 23, 5 (2018), 2948–2979.

[36] Anna Zaks and Amir Pnueli. 2008. Covac: Compiler validation by program analysis of the cross-product. In *International Symposium on Formal Methods*. Springer, 35–51.

[37] Kaizhong Zhang and Dennis Shasha. 1989. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing* 18, 6 (1989), 1245–1262.

A APPENDIX

Table 5. Example of questions we present in the user study

Assignment 7: Write a program to print pattern 1

Description: Given an integer N(N>0) as input, your program should output the following pattern.

Input: 5

Output: 5432*\n 543*1\n 54*21\n 5*321\n *4321

Student buggy submission

```
1  int main(){
2    int a,n,N,i,j;
3    scanf("%d",&N);
4    for(j=1;j<=N;j=j+1){
5      for(i=1;i<=N;i=i+1){
6        if(i+j==6)
7          printf("*");
8        else{
9          a=N+1-i;
10         printf("%d",a);
11       }
12     }
13     printf("\n");
14   }
15   return 0;
16 }
```

Repaired program

```
1  int main(){
2    int a,n,N,i,j;
3    scanf("%d",&N);
4    for(j=1;j<=N;j=j+1){
5      for(i=N;i>=1;i=i-1){
6        if(i==j)
7          printf("*");
8        else{
9          
10         printf("%d",a);
11       }
12     }
13     printf("\n");
14   }
15   return 0;
16 }
```

Question 1: Rate the quality of the generated repair (in terms of semantic correctness, size, etc).

Question 2: Rate the possibility that you would like to use the repair (either complete or partial) as feedback to the student.

Question 3: Rate the possibility that you would like to use the repair indirectly: to help formulate your own custom feedback to student.