2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)

Re-factoring based Program Repair applied to Programming Assignments

Yang Hu*
The University of Texas at Austin
huyang@utexas.edu

Umair Z. Ahmed†
National University of Singapore
umair@comp.nus.edu.sg

Sergey Mechtaev
University College London
s.mechtaev@ucl.ac.uk

Ben Leong
National University of Singapore
bleong@comp.nus.edu.sg

Abhik Roychoudhury
National University of Singapore
abhik@comp.nus.edu.sg

***Abstract*—Automated program repair has been used to provide feedback for incorrect student programming assignments, since program repair captures the code modification needed to make a given buggy program pass a given test-suite. Existing student feedback generation techniques are limited because they either require manual effort in the form of providing an error model, or require a large number of correct student submissions to learn from, or suffer from lack of scalability and accuracy.**

In this work, we propose a fully automated approach for generating student program repairs in real-time. This is achieved by first re-factoring all available correct solutions to semantically equivalent solutions. Given an incorrect program, we match the program with the closest matching refactored program based on its control flow structure. Subsequently, we infer the input-output specifications of the incorrect program's basic blocks from the executions of the correct program's aligned basic blocks. Finally, these specifications are used to modify the blocks of the incorrect program via search-based synthesis.

Our dataset consists of almost 1,800 real-life incorrect Python program submissions from 361 students for an introductory programming course at a large public university. Our experimental results suggest that our method is more effective and efficient than recently proposed feedback generation approaches. About 30% of the patches produced by our tool Refactory are smaller than those produced by the state-of-art tool Clara, and can be produced given fewer correct solutions (often a single correct solution) and in a shorter time. We opine that our method is applicable not only to programming assignments, and could be seen as a general-purpose program repair method that can achieve good results with just a single correct reference solution.

***Index Terms*—Program Repair, Programming Education, Software Refactoring**

I. Introduction

Program repair is an emerging technology that seeks to rectify program errors automatically, thereby meeting a correctness criterion, such as passing a test-suite. Besides improving programmer productivity, this technique can be applied to programming education. Particularly, program repair has been applied to automated grading [1], and providing hints about program errors [2]. In this work, we propose a repair method where an incorrect program can be repaired with the help of one or more correct reference solutions. While our approach is general-purpose, in our experiments, we focus on generating repair-based feedback for incorrect programming assignments.

Program repair has previously been used to provide feedback on incorrect student submissions for programming assignments [1]–[7]. The programming assignments are usually segments of code, so the limited scalability of existing program repair techniques is not a concern. However, it has been observed from a corpus of programming assignments that student submissions are often severely incorrect [1]. This is in stark contrast to the "competent programmer hypothesis" that assumes code bases are largely correct. Since programming assignments are written by novice programmers and can be substantially erroneous, they are a testbed to validate the effectiveness of program repair techniques. Since the submissions for programming assignments are often incorrect, the search space of edits to be navigated for program repair can be very large, even though the program might be small.

Existing systems that repair incorrect programming assignments have significant drawbacks because of the manual effort involved, underlying assumptions about the availability of correct solutions, and scalability or accuracy concerns. Approaches like Autograder [5] assume the availability of an error model that has to be provided manually. Efforts like sk_p [8] rely on neural networks to correct programs and suffer from low precision; a recent work has extended neural reasoning with symbolic analysis [6]. However, the accuracy of repairs typically remains low in such efforts. Refazer [7] learns program transformation schema from past submissions and its performance critically depends on the quality and quantity of corpus available. The recent works of *Clara* [3] and *Sarfgen* [4] compare an incorrect assignment with an available correct assignment. Such approaches assume the availability of a large number and diversity of correct solutions. However, this assumption often does not hold in practice, e.g. when a newly crafted assignment is given by an instructor.

Technical Contribution: The main technical contribution of this paper is a fully automated program repair method for repairing incorrect student submissions for programming assignments. While our technique can exploit the availability

*This work was done by the first author at National University of Singapore.
†Corresponding author.

978-1-7281-2508-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ASE.2019.00044

of a large number of correct solutions to perform better, we only assume and require one correct reference solution. Our approach is to use re-factoring rules to generate a correct solution with the same control flow as the incorrect program. Since the buggy program and the re-factored correct program possess the same (or similar) control flow, we compare their basic blocks and generate candidate variable mappings between the two programs based on dynamic observations over test executions and static analysis. Given such a variable mapping, we formulate the program repair problem as judiciously synthesizing expressions at selected basic blocks to meet the given correctness criterion (such as passing a test-suite). This synthesis problem is solved by efficient search-based synthesis where a large space of expressions is efficiently navigated to construct minimal repairs. The expressions considered for repairs of the basic blocks are obtained from expression templates or by mutating existing expressions. Our *Refactory* tool implementation of the above approach has been made available at https://github.com/githubhuyang/refactory

Conceptual Contribution and Results: If we envision the feedback generation problem through means of automated program repair as one of search space construction and traversal (with the search space capturing the possible edits of the buggy program), our solution enables a novel way to present and understand this search space. This is the main conceptual contribution of the work, and we believe this also leads to more superior experimental results as evidenced by our repair tool for actual Python programs from a real student submission data set. By separating the control flow matching (obtained via refactoring) from data-flow matching (achieved via search-based synthesis), we can construct small legible program repairs to be used as feedback to the students. We evaluate our approach on a large data set of $1,783$ buggy student programs, that was curated from five different Python assignments offered during a first-year university course credited by 361 students. Our tool *Refactory* achieves a higher repair rate, smaller patch size and less overfitting when compared to state-of-the-art tools such as *Clara* [3]. To verify the generality of our approach and crafted refactoring rules, we randomly sample an additional six assignments containing $7,290$ buggy student programs and observe similar results (Section VI). In addition to the practical utility of our technique in feedback generation, we believe that our viewpoint of cleanly partitioning the search-space of edits by separating control flow matching from expression synthesis can be useful for automated program repair.

II. Overview

Fig. 1 gives a high-level overview of our approach. Our approach takes three inputs: a test-suite T, a buggy program P_b and (one or more) correct programs C. Our approach includes three phases, which are elaborated in the following.

Phase 1. Refactoring: Given a set of refactoring rules, we conduct software refactoring on correct programs (C) to generate additional correct programs with new control flow structures. For example, Fig. 2a shows a correct program for the programming assignment `sequential search`, which

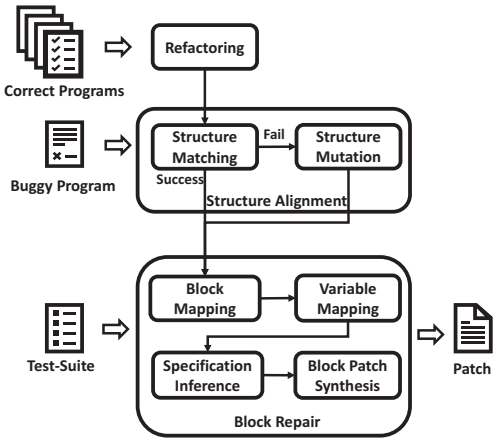


Fig. 1: Overview of our approach

outputs how many numbers in a sorted number sequence `seq` are smaller than `x`. To generate a correct program with new control flow, we mutate the control flow of the correct program by adding an empty `else` branch to an `if` branch. The refactored correct program is shown in Fig. 2b.

Phase 2. Structure Alignment: We perform *structure matching*, for finding refactored correct programs which have the same control flow structure with the buggy program P_b. If we cannot find such programs, P_b may have bugs in its control flow. To fix such bugs, we conduct *structure mutation*, which edits the control flow structure of P_b to that of closest refactored correct program in terms of tree edit distance.

Phase 3. Block Repair: Among all correct programs which have the same control flow structure with the buggy program, we search for the correct programs which are the top-k closest to the buggy program P_b (we set $k = 5$ in our experimental evaluation). For any of these top-k closest programs, if we can construct a patch passing the given test-suite T, we have succeeded in repairing, and hence generating feedback.

Phase 3.1 Block Mapping: We build a mapping between basic blocks in a correct program P_c and those of P_b based on the graph isomorphism of the control flow graph of P_c and P_b. For example, consider the buggy program in Fig. 2c and the refactored correct program in Fig. 2b, where line 2,3,4,6,7 are different basic blocks (although line 7 in the buggy program is empty, we regard it as an empty basic block). Assume that B_i^c is the basic block in line i in the refactored correct program, and B_i^b is the basic block in line i in the buggy program. Based on graph isomorphism, we can get $\{B_c^i \mapsto B_b^i\}_{i \in 2,3,4,6,7}$.

Phase 3.2 Variable Mapping: We build a variable mapping between the correct program P_c and the buggy program P_b using *dynamic equivalence analysis* (DEA) [3] and *define/use analysis* (DUA). In DEA, we collect the trace of each variable when running the correct and the buggy programs, and then map two variables if they take the same values in the same order when running the same test. For variables that are not mapped by DEA, we apply DUA, which maps two variables if the blocks where the first variable is defined/used corresponds

```
1 def search(x, seq):
2   for i in range(len(
      seq)):
3     if x <= seq[i]:
4       return i
5
6
7   return len(seq)
```

(a) A correct program

```
1 def search(x, seq):
2   for i in range(len(
      seq)):
3     if x <= seq[i]:
4       return i
5     else:
6       pass
7   return len(seq)
```

(b) A refactored correct program

```
1 def search(e, lst):
2   for j in range(len(
      lst)):
3     if e < lst[j]:
4       return j
5     else:
6       return len(lst)
7
```

(c) A buggy program

```
1 def search(e, lst):
2   for j in range(len(
      lst)):
3     if e <= lst[j]:
4       return j
5     else:
6       pass
7   return len(lst)
```

(d) A fixed program

Fig. 2: Example programs of the `sequential search` programming assignment from our dataset.

to the blocks where the second variable is defined/used. To illustrate these approaches, consider building a variable mapping between the buggy program in Fig. 2c and the refactored correct program in Fig. 2b using the tests `search(2,[1,2,3])` and `search(3,[4,5,6])`. Table Ia and Table Ib show all the variable traces collected using DEA. Since the traces of `e` and `x` are the same, and the traces of `lst` and `seq` are the same, we get a variable mapping $\{e \mapsto x, lst \mapsto seq\}$. Note that `j` and `i` are not mapped by DEA because their traces are different. Then, we execute DUA that identifies that `j` and `i` are defined in line 2 and used in line 3 and line 4, and the basic blocks in line 2, 3, 4 in the buggy program correspond to the basic block in line 2, 3, 4 in the correct program. Thus, we map `j` to `i`, and finally obtain the variable mapping $\{e \mapsto x, lst \mapsto seq, j \mapsto i\}$.

Phase 3.3 Specification Inference: We generate a specification for each basic block in the buggy program P_b by (1) collecting inputs and outputs of each basic block in the correct program and (2) using the variable mapping to translate them into the inputs and expected outputs of each basic block in the buggy program. For instance, consider the buggy basic block `e<lst[j]` in the buggy program. We collect the inputs and outputs of its corresponding basic block `x<=seq[i]` in the refactored correct program (Table Ic). Then, we replace the variables using the variable mapping $\{e \mapsto x, lst \mapsto seq, j \mapsto i\}$ to generate the specification of `e<lst[j]`.

Phase 3.4 Block Patch Synthesis: We use the input-output specification derived in the previous step to check the correctness of each basic block in the buggy program. In the buggy program shown in Fig. 2c, the basic blocks in lines 3, 6 and 7 do not satisfy their input-output specifications, and hence we deem them to be in need for repair. We attempt to generate a patch for each incorrect basic block in the buggy program.

If the basic block in P_b is empty, we fix it based on the variable mapping and its corresponding basic block in the correct program P_c. For example, consider the buggy basic block in line 7, which is an empty basic block. Its corresponding basic block in the refactored correct program is `return len(seq)`. Based on the variable mapping $\{e \mapsto x, lst \mapsto seq, j \mapsto i\}$, we replace the empty basic block in line 7 of the buggy program with a fixed basic block `return len(lst)`.

If the basic block in P_b is not empty, while its corresponding basic block in P_c is empty, we fix it by making it empty. For example, consider the incorrect basic block in line 6. Its corresponding basic block in the refactored correct program is `pass`, which is a key word to show it is an empty basic block. We fix the basic block in line 6 of the buggy program to `pass`.

If the basic block in P_b and its corresponding basic block in P_c are both non-empty, then we synthesize a patch for the buggy basic block using its specification. Given a set of suspicious lines in a buggy basic block, we insert holes to produce a partial program. Then, we perform enumerative synthesis with test-equivalence analysis [9] to fill the holes in the partial program. We use two heuristics to generate expression candidates. First, we utilize *expression templates* (i.e., syntax patterns [10] of expressions) in correct programs. For example, given the expression `x<=seq[i]` in the refactored correct program, we can extract an expression template v_0`<=`v_1`[`v_2`]` where v_0, v_1, v_2 are free variables. Using this template, we can generate a candidate `e<=lst[j]`. We also generate expression candidates by mutating operators or variables of the expressions in the buggy program. For example, given the expression `e<lst[j]` in the buggy program, we generate candidates `e>lst[j]`, `e<=lst[j]`, `e>=lst[j]`, and `j<lst[j]`.

Once the search space of candidate expressions is constructed, we traverse them efficiently using an approach based on test-equivalence analysis [9]. In this approach, the candidate expressions are grouped together if they behave identically on the given input-output examples (these are the specification we derived earlier). Such an approach greatly contributes to the scalability of our technique, since it helps to avoid traversing and checking the candidate patches one by one.

After generating patches for each basic block, we combine them into a global patch and validate its correctness via the test-suite. Fig. 2d shows the fixed program.

III. Refactoring and Structure Mutation

In this section, we introduce refactoring rules for mutating the control flow structure of existing correct solutions to generate new semantically equivalent correct solutions with different control flow structures. This step is necessary since the accuracy of repairing a given buggy program depends on finding a correct program with similar control flow structure.

We designed generic rules based on the observation that the same algorithm can have syntactically different implementations. For example, although the two programs in Fig. 3 behave equivalently and contain the same basic blocks, the control flow structure of

Test Inputs	e	lst	j
`search(2,[1,2,3])`	2	[1, 2, 3]	0
`search(3,[4,5,6])`	3	[4, 5, 6]	0

(a) Variable traces of the buggy program

Test Case	x	seq	i
`search(2,[1,2,3])`	2	[1, 2, 3]	$0 \mapsto 1$
`search(3,[4,5,6])`	3	[4, 5, 6]	0

(b) Variable traces of the correct program

Input			Output
x	seq	i	
2	[1, 2, 3]	0	False
2	[1, 2, 3]	1	True
3	[4, 5, 6]	0	True

(c) Inputs and outputs of `x<=seq[i]`

TABLE I: Motivating example of our approach.

```
1 def func(x):
2     if x <= 0:
3         return -1
4
5     return 1
```

(a) Control Flow CF_x

```
1 def func(x):
2     if x <= 0:
3         return -1
4     else:
5         return 1
```

(b) Control Flow CF_y

Fig. 3: Semantically equivalent programs with different control flow structures.

Fig. 3a, given as $\{\texttt{Func}_{\texttt{start}}, \texttt{If}_{\texttt{start}}, \texttt{If}_{\texttt{end}}, \texttt{Func}_{\texttt{end}}\}$, differs from the control flow structure of Fig. 3b, given as $\{\texttt{Func}_{\texttt{start}}, \texttt{If}_{\texttt{start}}, \texttt{If}_{\texttt{end}}, \texttt{Else}_{\texttt{start}}, \texttt{Else}_{\texttt{end}}, \texttt{Func}_{\texttt{end}}\}$.

Since there is potential to generate infinitely many correct variants using our refactoring rules, we guide the search for closest refactored program using the following heuristic. The correct programs are ranked based on their edit distance from the given buggy program. Rules are repeatedly applied on the closest correct program to generate new correct programs until a correct program with a matching control flow is found, or a predefined maximum depth is reached, or a timeout occurs.

As observed in our dataset of 1783 incorrect programs (Section V), 35% of them do not have a matching control flow structure in the correct submissions. Thus, existing techniques such as *Sarfgen* [4], which rely on exact control-flow matches, cannot repair these programs. We define a total of 10 bi-directional refactoring rules divided into five different categories, which are listed in Fig. 4. These rules transform control flow structures of our 2442 correct student programs, after which only less than 20% of the incorrect programs lack a matching correct program. For these remaining incorrect programs, structure mutation is used to borrow the missing control flow structure from the closest matching correct program.

A. Existing conditional transformations

This category of rules transform existing conditional statements. The rules are presented in the form of abstract syntax tree (AST) transformations.

1) Successor statements to a conditional jump: The rule R_{A_1} in Fig 4 states that the block of statements S which succeeds a conditional jump (such as `Break`/`Continue`/`Return`), can occur either as a successor node to the `If` block (Fig. 4a), or inside an `Else` branch to the `If` block (Fig. 4b).

Consider our earlier example listed in Fig. 3. The program in Fig. 3a (respectively Fig. 3b) can be mutated to the program in Fig. 3b (resp. Fig. 3a) on application of refactoring rule R_{A_1}, since the control flow of program CF_x (resp. CF_y) matches with the control flow of rule CF_a (resp. CF_b).

2) Conditional statement with conjunction: An `If` statement with the condition being a conjunction of C_1 and C_2 (Fig. 4c) can be rewritten as a nested `If` structure, containing the conditions C_1 and C_2 individually (Fig. 4d), using rule R_{A_2}.

B. New conditional transformations

These set of rules introduce additional guards; either around arbitrary statements, or around existing conditionals.

1) Introduce new `If` conditionals: In this rule R_{B_1}, we introduce three types of `If` conditional blocks. Fig. 4f adds a trivially `True` conditional guard around an arbitrary node S. Fig. 4g introduces a trivially `False` conditional guard around an arbitrary block B_1^*. Fig. 4h introduces an arbitrary condition C_1^* around a `pass` (no-op) statement.

The arbitrary block B_1^* (respectively condition C_1^*) are placeholders which can match and copy any corresponding block (resp. condition) of incorrect program, during the *block mapping* phase of our approach described later in Section IV-A.

2) Introduce new `Elif`/`Else` branch: The rule R_{B_2} introduces `Elif` and `Else` branching statements to an existing `If` conditional statement. Fig. 4j adds a trivially `False` `Elif` branch containing arbitrary block B_1^*. Fig. 4k introduces an arbitrary C_1^* conditional `Elif` branch, around a `pass` (no-op) statement. Fig. 4l adds an `Else` branch containing `pass` statement.

C. Loop guards

These set of rules deal with introducing additional guards surrounding an existing loop structure.

1) Introduce guard around `For` loop: Programs containing `for` statement which loops over an iterator (such as list) can be mutated into a new program structure by introducing guards around the loop, targeting the case when iterator is empty (Fig 4m to Fig 4n), or non-empty (Fig 4m to Fig 4o).

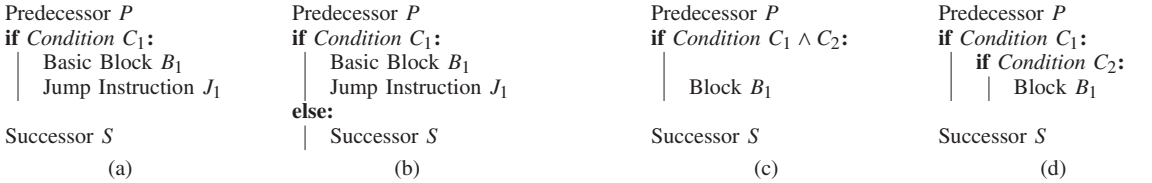
2) Introduce guard around `While` loop: Similar to previous rule, guards can be introduced in programs which loop over an iterator using `while` loop, targeting the case when iterator is empty (Fig 4p to Fig 4q), or non-empty (Fig 4p to Fig 4r).

D. While loop transformations

These set of rules replace `while` loop structure with an equivalent conditional jump statement, or vice-versa.

1) Conditional break inside `While` loop: A program which loops until a condition C_1 is satisfied (Fig. 4s) can be refactored into another program which loops indefinitely, with a C_1 conditional `break` instruction inside the loop's body (Fig. 4t).

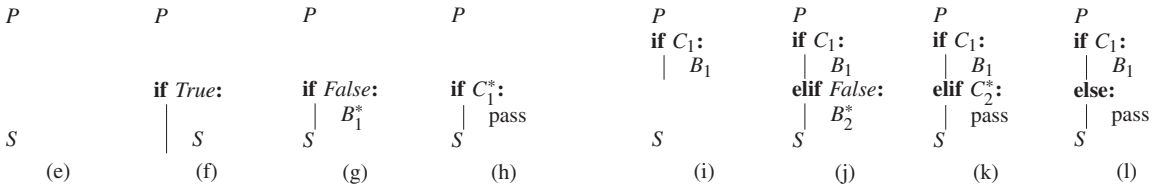
Category A: Existing conditional transformations



Rule R_{A_1}: Successor statements to a conditional jump.

Rule R_{A_2}: Conditional expression with conjunction.

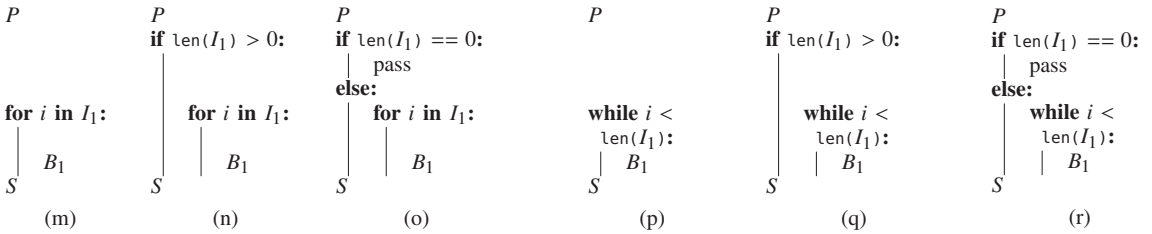
Category B: New conditional transformations



Rule R_{B_1}: Introduce new `If` conditionals.

Rule R_{B_2}: Introduce new `Elif`/`Else` branch.

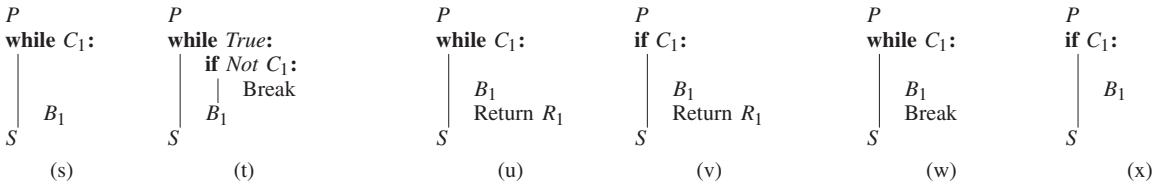
Category C: Loop guards



Rule R_{C_1}: Introduce guard around `For` loop.

Rule R_{C_2}: Introduce guard around `While` loop.

Category D: `While` loop transformations

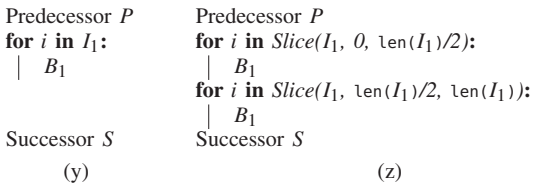


Rule R_{D_1}: Conditional break inside `While`.

Rule R_{D_2}: Unconditional return inside `While`.

Rule R_{D_3}: Unconditional break inside `While`.

Category E: Loop unrolling



Rule R_{E_1}: Split a `For` loop by iterator slicing.

Fig. 4: List of refactoring rules

2) Unconditional return inside `While` loop: A `While` loop which contains an unconditional `return` jump (Fig. 4u) can be replaced with an equivalent `If` conditional statement (Fig. 4v), since the block of statements inside the loop will be executed only once on successful satisfaction of the loop guard.

3) Unconditional break inside `While` loop: Similar to the previous rule, a `While` loop which contains an unconditional `break` jump (Fig. 4w) can be replaced with an equivalent `If` conditional statement without the `break` statement (Fig. 4v).

E. Loop unrolling

1) Iterator slicing: A `for` loop iterating over a sequence (Fig. 4y), can be split into two loops that iterate over two distinct sets of consecutive elements (Fig. 4z). The operator $\texttt{Slice}(I, i_1, i_2)$ (denoted as $I[i_1 : i_2]$) returns a subsequence of the elements starting of I at index i_1 until index i_2.

F. Structure Mutation

Given a buggy program P_b, we first search for a program P_c from the set of correct student submissions and their refactored variants, such that P_c has the same control-flow structure as P_b. If no such match is found, we attempt *structure mutation* that modifies the control-flow structure of the buggy program P_b. First, it searches for the closest program P'_c wrt control-flow structure from the set of correct programs and their refactored variants. Then, it borrows a minimal number of control-flow nodes (such as if-conditional or loop statements) from P'_c into P_b, in order to make their structure isomorphic.

Unlike refactoring rules, which mutates the control-flow structure of correct programs while preserving semantic equivalence, structure mutation does not offer such guarantee.

IV. Block Repair

Given a correct program P_c that has the same control flow structure as the buggy program P_b, we execute *block repair algorithm* to repair P_b. The algorithm consists of four stages. First, we construct a block mapping based on the isomorphism of the two control flow graphs (CFG). Second, we find a mapping between the variables of P_b and P_c. Third, we infer a correct specification for each basic block in P_b from P_c. Finally, we synthesize a patch for each basic block of P_b, and combine all block patches into a global patch.

A. Block Mapping

The goal of this stage is to find a mapping between the basic blocks of P_b and those of P_c. Since P_b and P_c have the same control-flow structure, their control flow graphs are isomorphic. Thus, a block mapping is effectively an isomorphism between the two control flow graphs.

Definition 1. ***Block Mapping.*** *Let $\mathcal{G}(P_c)$ be CFG of P_c with nodes $\{B^c_i\}_{i\in 1..n}$ and $\mathcal{G}(P_b)$ be CFG of P_b with nodes $\{B^b_i\}_{i\in 1..n}$. We define a block mapping $\mathcal{B}(P_c, P_b)$ as a CFG isomorphism $\{B^c_1 \mapsto B^b_{j_1}, ..., B^c_n \mapsto B^b_{j_n}\}$ between $\mathcal{G}(P_c)$ and $\mathcal{G}(P_b)$, where $\{j_1, ..., j_n\}$ are different indexes from 1 to n.*

B. Variable Mapping

The purpose of variable mapping is to identify how variables of P_c correspond to the variables of P_b.

Definition 2. ***Variable Mapping.*** *Let $x_1, ..., x_m$ be variables of the correct program P_c, and $y_1, ..., y_n$ be variables of the buggy program P_b. Then, $\{x_{i_1} \mapsto y_{j_1}, ..., x_{i_s} \mapsto y_{j_s}\}$ is a mapping of variables if $i_1, ..., i_s \in 1..n$ are different indices and $j_1, ..., j_s \in 1..m$ are different indices.*

Since there exist many possible mappings, we apply *Dynamic Equivalence Analysis* (*DEA*) and *Define/Use Analysis* (*DUA*) to filter out irrelevant mappings. In DEA, we collect the variable traces collected on P_b and P_c. The trace of a variable in a test refers to the sequence of values that the variable takes during the test execution.

The intuition behind DEA is that if a variable x in P_c takes the same values in the same order as a variable y in P_b during each test execution, then they represent the same user intent. In this case, we say y is a variable candidate of x.

Definition 3. ***Mapped Variable Candidates in DEA.*** *Let x be a variable in P_c, y be a variable in P_b, and T be a set of tests. $\mathcal{M}_{DEA}(x)$ represents a set of variable candidates in P_b that x can be mapped to. We define $y \in \mathcal{M}_{DEA}(x)$ iff for each test $t \in T$, the sequence of values that y takes during the execution of P_b with t is the same as the sequence of values that x takes during the execution of P_c with t.*

In DUA, we assume that variables that are defined and used in the same manner are more likely to have the same user intent. We get a set of variable candidates in P_b which a variable in P_c can be mapped to as follows.

Definition 4. ***Mapped Variable Candidates in DUA.*** *Let $\mathcal{D}(P, x)$ be the set of basic blocks in the program P where the variable x is defined, $\mathcal{U}(P, x)$ be the set of basic blocks in the program P where the variable x is used, r be a variable in P_c and s be a variable in P_b. $\mathcal{M}_{DUA}(r)$ represents a set of variable candidates in P_b that r can be mapped to in DUA. We define $s \in \mathcal{M}_{DUA}(r)$ iff (1) there exists a one-one block mapping from $\mathcal{D}(P_c, r)$ to $\mathcal{D}(P_b, s)$ and (2) there exists a one-one block mapping from $\mathcal{U}(P_c, r)$ to $\mathcal{U}(P_b, s)$.*

Finally, we rule out all invalid variable mappings that do not map the variable r in P_c to any variable candidates in P_b.

Definition 5. ***Valid Variable Mapping.*** *Let $\{c_{i_1} \mapsto b_{j_1}, ..., c_{i_n} \mapsto b_{j_n}\}$ be a variable mapping between P_c and P_b. We say that the variable mapping is invalid if and only if*

$$\exists e \in 1..n.(b_{j_e} \notin \mathcal{M}_{DEA}(c_{i_e}) \cup \mathcal{M}_{DUA}(c_{i_e}))$$

If no variable mapping is valid, the block repair algorithm will report a repair failure on P_b. Otherwise, we enumerate all valid variable mappings one-by-one until the algorithm successfully infers a specification and synthesizes a patch.

C. Specification Inference

First, we analyze the correct program to extract a specification for each basic block. This is done by running P_c

on our test-suite T to collect input-output state pairs of each basic block in P_c. Here, input state refers to the values of all variables before executing the basic block, and output state refers to these values after executing the basic block.

Definition 6. ***Specification.*** *Let B be a basic block in a program P, and T be the test-suite. The specification of B is defined as a set of input-output state pairs $\{\langle I_j, O_j\rangle\}_{j\in 1,\ldots,r}$, where I_j denotes the input state and O_j denotes the expected output state of B given I_j as the input state.*

Note that in the above definition we use a set of state pairs because each basic block can be executed multiple times during a test execution. Our algorithm infers a specification of a basic block B_b based on that of its corresponding basic block B_c and the valid variable mapping M.

Definition 7. ***Specification Inference.*** *Let B_c be a basic block in the correct program P_c, and $\{\langle I_j^c, O_j^c\rangle\}_{j\in 1..r}$ be a specification of B_c, and B_b be the corresponding basic block in the buggy program P_b, and M be a valid variable mapping between P_c and P_b. A specification of B_b is inferred as a set of input-output state pairs $\{\langle I_j^b, O_j^b\rangle\}_{j\in 1..r}$ such that*

$$\{y \mapsto u\} \in I_j^b \;\Leftrightarrow\; \exists x.\ \{x \mapsto u\} \in I_j^c \wedge \{x \mapsto y\} \in M$$
$$\{y \mapsto u\} \in O_j^b \;\Leftrightarrow\; \exists x.\ \{x \mapsto u\} \in O_j^c \wedge \{x \mapsto y\} \in M$$

D. Patch Synthesis

Before repairing a basic block B_b in P_b, we verify the correctness of B_b by collecting the inputs and their corresponding outputs of B_b and comparing them with the inputs and expected outputs in its inferred specification. Formally speaking, we run P_b on the test-suite T to collect a set of input-output pairs $\{\langle \hat{I}_j^b, \hat{O}_j^b\rangle\}_{j\in 1,\ldots,z}$ of B_b. We say B_b is incorrect if there exist $\langle \hat{I}_u^b, \hat{O}_u^b\rangle$ and $\langle I_v^b, O_v^b\rangle$ such that $I_v^b \subset \hat{I}_u^b \wedge O_v^b \not\subset \hat{O}_u^b$.

For B_b is incorrect, we attempt to repair it. If either B_c or B_b is an empty basic block, we fix it either by generating an empty block as a patch of B_b or using the valid variable mapping to translate B_c to a patch of B_b. In other words, we replace all variable names in B_c with their corresponding variable names according to the valid variable mapping.

If B_c and B_b are not empty, we use a program synthesis technique to generate a patch for B_b based on its specification. Given a set of suspicious lines, we produce a partial program with holes inserted in buggy lines. We generate expression candidates for each hole. Our goal is to fill holes with expressions that enable the block to satisfy the specification.

Definition 8. ***Block Patch Synthesis.*** *Let B be an incorrect basic block and L be a set of suspicious buggy lines in B. Let $\mathcal{P}(B, L)$ be a partial block that holes are inserted into all lines in L. Let S_l be a set of expressions candidates for the hole in line $l \in L$. Let $C : S_1 \times \ldots \times S_l \times L \to \mathbb{R}$ be a cost function. Our aim is to find a repair $\langle s_1, \ldots, s_l\rangle \in S_1 \times \ldots \times S_l$ that (i) can fill in $\mathcal{P}(B, L)$ to pass the correct specification and (ii) $C(s_1, \ldots, s_l, L)$ be minimal among all such basic blocks.*

Typically, program repair techniques identify suspicious lines via statistical fault localization. Considering such techniques may not be accurate on students' submissions which are usually severely incorrect, we enumerate the all subsets of lines in B_b as sets of suspicious buggy lines in the ascending order of the number of lines until we find a patch.

A simple approach to generate a patch is to enumerate all block candidates by filling in holes in the partial block with all combinations of expressions. However, the search space might be huge, suffering from a combinatorial explosion as the number of holes grows. To mitigate this issue, we perform *test-equivalence analysis* [9] when searching for a patch. In a nutshell, we partition candidates into test-equivalence classes. For each class, only one representative patch is executed and verified, thereby reducing the number of test executions.

Definition 9. ***Test-Equivalence Relation.*** *Let $\mathbb{B}$ be a set of block candidates, and α be an input-output pair in the correct specification of B. An test-equivalence relation on α is defined as an equivalence relation $\leftrightarrow_\alpha \subset \mathbb{B}\times\mathbb{B}$ that if $B_1 \leftrightarrow_\alpha B_2$, then B_1 and B_2 both pass or fail α.*

The search space of expression candidates for each hole is constructed based on expression templates and operator/variable mutation. An expression template is a *syntax pattern* [10] where variable names in the expression are abstracted away (i.e., use a set of wildcards instead of the variable names). Expression templates are extracted from expressions in correct programs. Formally, let $e = \langle e_1, \ldots, e_n\rangle$ be an expression, where e_i denotes i-th token of e. Let V be a set of variable names. An expression template of e can be defined as a sequence of tokens $\langle e'_1, \ldots, e'_n\rangle$, where $e'_i = *$ iff $e_i \in V$. Given a set of variable names from the buggy program, a space of candidate expressions is generated by assigning each wildcard with a unique variable name.

We also generate a space of candidate expressions by mutating operators or variable names of the suspicious expressions from the buggy program. Let $e = \langle e_1, \ldots, e_n\rangle$ be an expression, where e_i denotes i-th token of e. We construct a space of candidate expressions by generating $e' = \langle e'_1, \ldots, e'_n\rangle$, where $e_j = e'_j$ when $j \in 1, \ldots, k-1, k+1, \ldots, n$, and e'_k is such that $e_k \neq e'_k$ and if e_k is a variable, then e'_k is another variable, and if e_k an operator, then e'_k is another operator.

V. Dataset and Experimental Setup

We choose *Clara* [3], one of the most recent and related feedback generation approach with publicly available implementation[1], as the baseline to compare our approach against. *Clara* [3] was evaluated on a similar dataset as AutoGrader [5], which consists of student attempts from MITx MOOC [11]. However, this dataset is not publicly available.

Instead, we evaluate both *Clara* and our *Refactory* tool on real student submissions, collected from an introductory Python programming course offered at the author's university (National University of Singapore). This course was credited by 361 students, who had to attempt a large number of programming assignments throughout the entire semester.

[1] https://github.com/iradicek/CLARA

ID	Description	Avg. #Lines of Code	#Correct Attempt	#Incorrect Attempt	%*CFG* Match		Repair Rate	Avg. Time Taken (sec)	Relative Patch Size (RPS)
					W/O $\mathcal{R}$	W/ $\mathcal{R}$			
1	Sequential search	10	768	575	80.00%	86.78%	98.96% (81.91%)	3.5 (12.1)	0.39 (0.51)
2	Unique dates/months	28	291	435	33.33%	68.28%	78.16% (42.07%)	4.8 (17.4)	0.56 (0.31)
3	Duplicate elimination	7	546	308	87.34%	89.61%	97.40% (92.86%)	4.7 (8.5)	0.34 (0.58)
4	Sorting tuples	9	419	357	52.94%	81.23%	88.24% (64.43%)	8.7 (20.6)	0.31 (0.84)
5	Top-k elements	11	418	108	80.56%	83.33%	87.96% (93.52%)	13.1 (11.8)	0.32 (0.61)
1–5	overall	14	2442	1783	64.50%	81.44%	90.80% (71.28%)	5.5 (13.6)	0.40 (0.56)

TABLE II: Results on five programming assignments. "% CFG Match" is the percentage of incorrect submissions for which correct submissions with matching control-flow structure are found without refactoring (W/O $\mathcal{R}$) and with refactoring (W/ $\mathcal{R}$). Repair rate, average time-taken and relative patch size per assignment are shown for *Refactory* (and for *Clara* in brackets).

Students were allowed to submit multiple attempts, only the last of which was graded. On each attempt, students received the test-suite evaluation results as feedback.

From these assignments, we filter out those attempts that contain syntax errors, or contain a single basic block (trivial assignments), or utilize Python language features unsupported by implementation of *Refactory* or *Clara* (such as lambda functions, exception handling, Object-Oriented Programming concepts). After filtering, 19 assignments remain, from which we selected 5 assignments for an initial evaluation and crafting our refactoring rules. Totally. 2,442 correct submissions and 1,783 incorrect student attempts form our dataset, along with the instructor designed test-suite and reference program. This dataset is described in Table II. Our dataset and *Refactory* repair tool are publicly released to aid further research[2].

To test the generality of our refactoring rules and block repair algorithm, we report results on the remaining 14 assignments containing 6,448 correct submissions and 7,290 incorrect student attempts as well.

All experiments are conducted using Intel® Core™ i7-4770 CPU, 8GB RAM and Ubuntu 18.10. *Clara* has an offline phase for clustering the correct programs, for which we set a five-minute timeout per assignment. For the online phase, *Clara* and *Refactory* are configured to run in a single-thread mode with one-minute timeout to repair each incorrect submission.

VI. Evaluation

To evaluate the effectiveness of *Refactory*, we aim at answering the following research questions:

RQ1 Given a large number of correct submissions, how effectively can *Refactory* repair incorrect submissions?

RQ2 Given a small number of correct submissions, how effectively can *Refactory* repair incorrect submissions?

RQ1 and RQ2 investigate the applicability of our approach to assignments with different number of correct submissions. Existing data-driven approaches such as *Clara* and *Sarfgen* are designed for assignments with a large number of correct submissions. We use refactoring rules to generate new correct submissions, which makes our approach applicable when only a small number of correct submissions is available.

To answer RQ1, we evaluate *Refactory* and *Clara* on the entire dataset of correct programs. To answer RQ2, we evaluate them on downsampled dataset, where the number of correct submissions provided as input to these tools is varied from 100% to 0% (only the reference program is used).

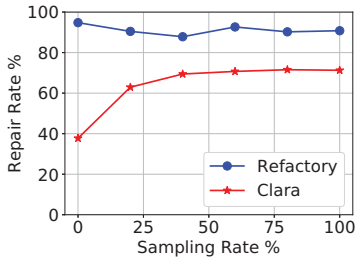
Explanation of Table II: Table II shows the results on our 5 assignments selected for initial evaluation. *Clara* can generate repairs for 71.28% of 1783 incorrect submissions consuming 13.6 seconds on average per repair. In comparison, *Refactory* can generate repairs for 90.8% of incorrect submissions requiring 5.5 seconds on average per repair. Which demonstrates that *Refactory* can repair a significantly larger % of incorrect submissions, while requiring lesser amount of time.

This high repair rate of 90.8% by *Refactory* is made possible by our refactoring and structure mutation phase. As seen from Table II, only 64.5% of incorrect programs have a matching correct submission with exactlt the same control flow structure. By applying our refactoring rules, we generate new correct programs, thereby increasing the %*CFG* match to 81.4%. The remaining incorrect programs which do not have a *CFG* match with correct programs undergo structure mutation during online phase, bringing our overall repair rate to 90.8%. In comparison, almost a half of *Clara*'s failures occur due to exceeding the running timeout of 1 min. The remaining occur when *Clara* is unable to find a matching correct submission with the same looping structure as incorrect submission.

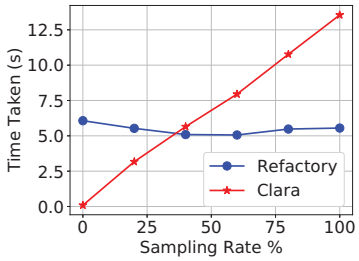
We also report on Relative Patch Size (*RPS*) metric, to further evaluate generated patches. Patch size is defined as the Tree-Edit-Distance (*TED*) between the Abstract Syntax Tree of given buggy program (AST_b) and repaired program (AST_r) generated by tool. Relative Patch Size (*RPS*), as defined by *Clara* [3], normalizes the patch size with the size of original buggy program's AST. $RPS = \text{TED}(AST_b, AST_r)/\text{Size}(AST_b)$ As shown in Table II, repairs generated by *Refactory* have a smaller average RPS compared to those generated by *Clara* (for majority of incorrect attempts), which indicates that our repairs are smaller and hence more likely to help students in rectifying bugs in their incorrect attempts.

Explanation of Figure 5: Fig. 5a shows the average repair rate achieved by both tools for various *sampling rate* of correct submissions provided as input to the tools. The repair rate of *Refactory* is relatively consistent when the sampling rate is reduced while *Clara*'s repair rate drops significantly with decrease in sampling rate. For example, when sampling

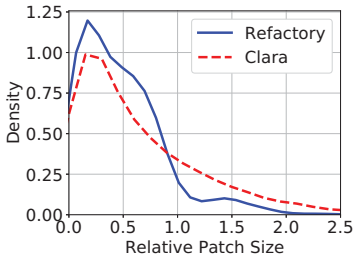
[2]https://github.com/githubhuyang/refactory



(a) Average repair rate across various sampling rates of correct solutions.



(b) Average time taken to repair incorrect program, across various sampling rates.



(c) Density function of relative patch size.

Fig. 5: Performance comparison between *Clara* and *Refactory*.

rate is 0% (only the reference program is provided as input), *Clara* achieves less than 40% repair accuracy due to its reliance on diverse correct programs, compared to more than 90% achieved by *Refactory* due to its ability to generate new correct programs using refactoring rules.

Fig. 5b shows the average time taken by both tools to repair a single program across different sampling rates. *Clara*'s time cost is proportional to the sampling rate, which highlights its scalability issue on dataset with a large number of correct submissions, due to the increase in complexity of *Integer Linear Programming* (*ILP*). Since *Refactory* selects the top-k closest refactored correct programs with same control flow structure for repair generation, it scales better than *Clara*.

Fig 5c compares the relative patch size (RPS) of *Refactory* against *Clara* [3], by plotting its density function. This is obtained by defining independent random variables which represent the RPS of a patch generated by *Refactory* and *Clara*, for each assignment for each sampling rate. Then Gaussian Kernel Density Estimator [12] is used to generate their probability density functions based on individual observation values. Formally, let $(x_1^t, \ldots, x_n^t)$ denote the n observation values of RPS of a patch generated by a tool t across all assignments and sampling rates. We estimate the density function as $f(x) = \frac{1}{nh}\sum_{i=1}^{n}\mathcal{K}(\frac{x-x_i^t}{h})$, where h is a smoothing parameter and $\mathcal{K}$ represents a Gaussian kernel function. From Fig. 5c, the estimated density (y-axis) of patches for *Refactory* tool is higher compared to *Clara*'s when RPS (x-axis) is smaller than 0.9. In other words, the patches generated by our tool are concentrated towards small RPS.

Results on Full Data-set: To demonstrate that our manually crafted refactoring rules do not over-fit our initially selected five assignments, reported in Table II, we report additional results on the 14 held out assignments (refer Section V). On these 14 new assignments, our *Refactory* tool achieves repair rate of 71.65% on 7290 incorrect submissions within 6.4 seconds on average, and the generated repair's average relative patch size is 0.44. In contrast *Clara* can repair 30.4% of 7290 incorrect submissions in 15 seconds on average, with a relative patch size of 0.82. Furthermore, our refactoring rules can improve the overall *CFG* match rate from 55.47% (W/O $\mathcal{R}$) to 67.08% (W/ $\mathcal{R}$). Overall, our tool achieves high accuracy with small patch size on the full set of 19 assignments.

VII. Related Work

In this section, we briefly review existing state-of-the-art approaches targeting introductory programming assignments, and clarify the novelties provided by our approach.

A. Automated Program Repair

The field of automated program repair [13], where changes are suggested to the program source code for fixing observable errors and vulnerabilities, has witnessed an explosive growth in recent years. GenProg [14] uses search-based techniques to navigate the space of edits, so as to automatically find an edit where the edited program passes a given test-suite. Learning or pattern-based approaches have been successfully applied in program repair, e.g. finding patterns of human patches and using them in program repair [15], or using machine learning techniques to rank patch candidates [16].

SemFix [17] and Angelix [18] extract specification from tests, and use it to synthesize a patch. *Refactory* also uses specification inference, however it differs in two key aspects. First, the specification is inferred from a correct solution that might have a different implementation and use different variable names. To tackle this, we apply automatic refactoring and variable matching. Second, *Refactory* is able to synthesize patches not only for program expressions, but also for basic blocks by transplanting fragments of code from the correct solution. Symbolic execution based patch synthesis algorithm of SemFix is orthogonal to our core contributions, and can be potentially used to improve expression synthesis in *Refactory*.

SimFix [19] mines repairs from similar code and past patches. In principle, it can be applied to correct student assignments when a history of previous corrections and a sufficient number of similar solutions are available. In contrast, our approach is designed to work when only a few correct solutions are available, without relying on the history of previous corrections. Instead, from a single correct solution we can generate several correct solutions one of which can match the control flow of the given buggy program, followed by which we resort to basic block synthesis. Thus, our approach

```
1 def swap(lst, i, j):
2     tmp    = lst[i]
3     lst[j] = lst[i]
4     lst[i] = tmp
```

(a) An incorrect program

```
1 def swap(lst, i, j):
2     tmp    = lst[i]
3     lst[i] = lst[j]
4     lst[j] = tmp
```

(b) A correct program

Fig. 6: Function to swap two elements in `list`.

is more applicable in pedagogical scenarios, e.g. when a newly crafted assignment is given by an instructor.

Our work on using a reference correct solution may appear superficially similar to the recent paper [20]. However, [20] employs simultaneous symbolic analysis of both buggy and correct programs to produce provably correct repairs. Similar to other recent works on repair in education [3], [4], we do not give formal guarantees about the repairs generated by our approach. Instead we use refactoring and synthesis to efficiently represent/navigate the space of patches.

B. Feedback Generation

Automated program repair tools, originally designed to work on large codebases targeting experienced developers, have been used to provide feedback to students on introductory programming assignments with limited success [1]. Hence, new tools have been proposed in literature, targeting the novice programmers and their mistakes specifically. AutoGrader [5] proposes a program synthesis based approach which takes a reference solution and manually provided error model to generate repairs on incorrect programs. Refazer [7] attempts to learn simple syntactic program transformations from historical edit examples, and applies AST rewrite rules on matching incorrect programs to automatically repair them.

Clara [3] and *Sarfgen* [4] are two recent approaches related to our work that generate complex patches on incorrect student programs automatically. Given an incorrect program attempt, *Clara* and *Sarfgen* rely on finding a correct solution having the same looping and control-flow structure, respectively. This assumption presents a serious challenge when there is lack of access to a diverse set of existing correct solutions, for example when a newly crafted assignment is given by instructor. To address this issue, our approach attempts to refactor one or more correct solutions to generate new semantically equivalent correct solutions, with different looping/control-flow structures. In addition, as noted in our experiments on running time, *Clara* suffers from a *scalability* problem due to the use of Integer Linear Programming. We are unable to compare our run-time/accuracy with *Sarfgen* since their implementation has not been publicly released; moreover *Sarfgen* is targeted towards C# while our tool works only for Python programs.

Consider the incorrect student attempt for `swap` function in Fig. 6a. Here, the student has made a mistake in swapping two elements of a list, `lst[i]` and `lst[j]`, through use of an intermediate `tmp` variable. Given the correct program shown in Fig. 6b as input, our *Refactory* approach generates the minimal repair of modifying a single line #2 from `tmp = lst[i]` to `tmp = lst[j]`, by replacing each line with holes and synthesizing expression candidates. While *Clara* generates a sub-optimal solution at the block level, by borrowing the two differing lines (3 and 4) from the correct program.

VIII. Threats to validity

Our choice of refactoring rules are by no means exhaustive, and primarily targets conditionals, looping structures, and their combinations; which constitutes majority of the control flow mistakes made by students in introductory programming classes. While we do report experimental results on 6 additional assignments unseen during refactoring rule crafting, in future we plan to rigorously test our tool on larger variety of programs collated from other publicly available datasets.

Our implementation currently supports only structured programming control flow structures. In future, we plan to extend our approach to handle Object-Oriented Programming concepts. Additional complex features available in Python, such as list comprehensions or lambda functions, are currently not handled since novice students rarely utilize advanced concepts.

Correctness of repairs is verified against instructor-provided test-suite, a manually designed incomplete specification.

Finally we note that while our implementation is targeted towards Python programs, our approach based on refactoring and block repair is not restricted to Python programs.

IX. Discussion

The recent past has witnessed an explosion of works on automated feedback generation for introductory programming assignments, through means of program repair [1]–[7]. At a general level, most of the works search in the space of program edits to either generate feedback for students or to help automatically grade assignments. Due to a large variety of coding errors in programming assignments written by novice programmers, the search space of edits between a given incorrect program and a correct program tends to be huge [1]. Many past works have contributed immensely in the navigation of this search space of edits which may enable feedback generation for students. In our work, we have focused first on the search space representation, thereby prompting our refactoring phase, and then attempted to systematize the navigation of possible patches of a basic block by partitioning the candidate patches using test-equivalence analysis. Such a representation and navigation of the search space also allows us to work in various set-ups including those where many correct solutions are not available.

Our efforts are embodied in the form of *Refactory*, a customized Python repair system. We have employed the repair system extensively over a large data-set of more than a thousand programming assignments collected from hundreds of students enrolled in an introductory programming course.

In future work, we plan to conduct detailed user studies where the feedback from our tool can be generated live during tutorial or recitation sessions, so as to gauge the possible improvement in meeting learning outcomes.

ACKNOWLEDGMENTS

This work was supported in part by Office of Naval Research grant ONRG-NICOP-N62909-18-1-2052. This work was partially supported by the National Satellite of Excellence in Trustworthy Software Systems, funded by NRF Singapore under National Cybersecurity R&D (NCR) programme. We would like to thank Tegawendé F. Bissyandé and the anonymous reviewers of ASE for their valuable feedback.

REFERENCES

[1] J. Yi, U. Z. Ahmed, A. Karkare, S. H. Tan, and A. Roychoudhury, "A feasibility study of using automated program repair for introductory programming assignments," in *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2017.

[2] S. Gulwani, "Example-based learning in computer-aided stem education," *Communications of the ACM*, vol. 57, 2014.

[3] S. Gulwani, I. Radicek, and F. Zuleger, "Automated clustering and program repair for introductory programming assignments," in *39th ACM SIGPLAN Conference on Programming Lnaguage Design and Implementation (PLDI)*, 2018.

[4] K. Wang, R. Singh, and Z. Su, "Search, align, and repair: data-driven feedback generation for introductory programming exercises," in *39th ACM SIGPLAN Conference on Programming Lnaguage Design and Implementation (PLDI)*, 2018.

[5] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," in *ACM SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*, 2013.

[6] S. Bhatia, P. Kohli, and R. Singh, "Neuro-symbolic program repair for correcting introductory programming assignments," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2018.

[7] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Suzki, and B. Hartmann, "Learning syntactic program transformations from examples," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2017.

[8] Y. Pu, K. Narasimhan, A. Solar-Lezama, and R. Barzilay, "Skp: A neural program corrector for moocs," in *ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications (SPLASH)*, 2013.

[9] S. Mechtaev, X. Gao, S. H. Tan, and A. Roychoudhury, "Test-equivalence analysis for automatic patch generation," *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 4, pp. 15:1–15:37, Oct. 2018.

[10] J. Lee, D. Song, S. So, and H. Oh, "Automatic diagnosis and correction of logical errors for functional programming assignments," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, p. 158, 2018.

[11] MITx MOOC. [Online]. Available: https://www.edx.org/school/mitx

[12] B. W. Silverman, *Density estimation for statistics and data analysis*. Routledge, 2018.

[13] C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, 2019.

[14] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE transactions on software engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[15] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings of International Conference on Software Engineering (ICSE)*, 2013.

[16] F. Long and M. Rinard, "Prophet: Automatic patch generation via learning from successful patches," 2015.

[17] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*, 2013, pp. 772–781.

[18] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th international conference on software engineering*. ACM, 2016, pp. 691–701.

[19] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2018, pp. 298–309.

[20] S. Mechtaev, M.-D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury, "Semantic program repair using a reference implementation," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 129–139.